Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-royer-phoenix-00
Published:	8 February 2025
Intended Status:	Informational
Expires:	12 August 2025
Author:	DM. Royer
	RiverExplorer Games LLC

Phoenix: Lemonade Risen Again

Abstract

NOTE: This is just getting started, not ready for submission yet.

Email and MIME messages account for one the largest volumes of data on the internet. The transfer of these MIME message has not had a major updated in decades. Part of the reason is that it is very important data and altering it takes a great deal of care and planning.

This application transport can also transfer non-MIME data. It can be used as an XDR transport, or for opaue data (blobs of known or unknown data) transport.

Another major concern is security and authentication. This proposal allows for existing autnentication to continue to work.

This is a MIME message transport that can facilitate the transfer of any kind of MIME message. Including email, calendaring, and text, image, or multimedia MIME messages. It can transfer multipart and simple MIME messages.

The POP and IMAP protocols are overly chatty and now that the Internet can handle 8-bit transfers, there is no need for the overly complex text handling of messages.

This proposal includes a sample implementation. (https://github.com/RiverExplorer/Phoenix) Which also includes a gateway from this proposal to existing system. Thunderbid and Outlook plugins are part of the sample implementation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 August 2025.

Copyright Notice

Table of Contents

1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.1. ABNF additions

This specification adds some usage to ABNF [RFC5234] to deal with bit width in a binary number.

Terminals may specify a bit width. That is the number of bits in the value.

Section 2.3 of Terminal Values [RFC5234] is within this specification defined to be:

b = binary / binary:width

d = decimal / decimal:width

x = hexadecimal / hexadecimal:width

Where: with is the number of bits in the value. And must be an unsigned integer greater than zero.

And they will pack when applied to to a rule:

When the left side has a width: The number of bits on the left side must equal the number of bits on the right side.

In this example A Header is 32-bits in size and is composed of an 8-bit and 24-bit value. The most significant value are placed on the left of the rule:

```
      Header:32 = Offset:8 Length:24
```

The pseudo code could be:

```
// Header is a 32-bit unsigned integer.
// Offset is an 8-bit unsigned integer.
// Length is a 24-bit unsigned integer
//
Header = Offset << 24 | Length;
```

This shifts the 8-bit Offset over 24 bits to the left, then adds the 24-bit Length. The result would be:

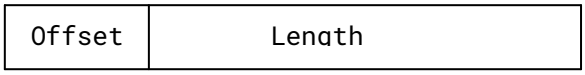


Figure 1: Packed Bit Example

2. Introduction

On the Internet, just about everything is a MIME object and there are many ways to transport MIME. This document specifies a new application level MIME transport mechanism and protocol. This document does not specify any new or changed MIME types.

Transporting MIME objects is generally done in one of two ways: (1) Broadcasting, (2) Polling. Both methods often require some form of authentication, registration, and selecting of the desired material. These selection processes are essentially a form of remote folder management. In some cases you can only select what is provided, and in others you have some or a lot of control over the remote folders.

In addition to other functions, this specification defines a remote and local folder management. This remote folder management is common with many type of very popular protocols. This design started by looking at the very popular IMAP and POP protocols.

An additional task is transporting the perhaps very large MIME objects. Some MIME objects are so large that some devices may default to looking at only at parts of the MIME object. An example is an email message with one or more very large attachments, where the device may default to not download the large attachment without a specific request from the user.

Some objects are transported as blocks of data with a known and fixed size. These are often transported with some kind of search, get, and put commands. In effect these are folder and file commands

Other MIME objects are transported in streams of data with an unspecified size, such as streaming music, audio, or video. This specification describes how to use existing protocols to facilitate the data streaming. And again, these are folder and file commands.

A MIME object can be a simple object, or it may contain many multipart sections of small to huge size. These sections can be viewed as files in the containing MIME object.

By implementing this specification application developers can use the techniques to manage local and remote files and folders. Remote email or files are the same thing in this specification. The sections of MIME object with multipart sections are viewed as files in the MIME object. You can interact with the entire folder, or just the files within it.

MIME objects have meta data, and they are called headers. Files and folders have meta data, and they are called file attributes. This specification does not mandate any meta data. It does define some that may be used by implementations. Other related specifications do define some meta data that is consistant with existing protocols. This protocol allows for a consistent transport of existing meta data and MIME objects.

File and folder meta data is a complex task that can involve access control lists and permissions. This specification defines a mechanism to transport this meta data, it does not define the meta data.

And this specification provides for the ability to define both protocol extensions and the creating of finer control for specific commands that may evolve over time.

This examples compares current folder and file manipulations to how it can be used in this protocol with email.

- You can search for file names. You can search email for: sender, subject, and more.
- You can search for file contents. You can search for email message contents.
- You can create, delete, and modify files. You can create, delete, and modify email messages.
- You can create, delete, and modify folders. You can create, delete, and modify email folders.

What this specification defines:

- How to use existing authentication implementations or use new ones.
- This specification describes a standard way to perform file and folder operations that are remote to the application and agnostic to purpose of data being transported.
- Specifies a way to migrate from some existing protocols to Phoenix. Provides links to sample implementations.

3. Terms and Definition used in this proposal

The following is a list of terms with their definitions as used in this specification.

AdminCmd
: A general term for any administrative command. Administrative and auditing operations. This list includes commands for authorized users to configure, query logs, errors, possibly user activity.

AuthCmd
: A general term for any authentication command. Authentication and authorization operations. These operations authenticate users and verity their authorization access.

Body Part ID
: A unique ID for a MIME Object. This is an unsigned 32-bit integer in network byte order that is assigned by the server and sent to the client on a successful folder open. This ID persists across connections. And as long as the MIME object does not get altered in any way, this ID is valid and persists across servers. It is the offset in octets from the beginning of the message to the start of the body part.

  See Index. (Section 3)

Command, CMD
: A specific protocol operation, or command. They are broken down into, AdminCmd, AuthCmd, FileCmd, and ProtoCmd. These are called a CMD or command.

FileCmd
: A general term for any file or folder command. This include creating, getting, modifying, deleting, moving, and renaming files.

Folder ID
: A unique ID for a MIME folder. This is an unsigned 32-bit integer in network byte order that is assigned by the server and sent to the client on a successful folder open. This ID persists across connections to the same server. Once a folder has an ID, it never changes on a server as described in Folders (Section 4.7).

  See Index. (Section 3)

IndexOP
: Header Index Operation. A command sent as part of a folder open command that tells the server which MIME headers it would like indexed.

  See Index. (Section 3)

HID
: Header Definition ID. And 8-bit unsigned integer the client has assiged to a specific header name.

  See Index. (Section 3)

HeaderName822
: A RFC822 or MIME header name. See Section 3.2 of [RFC0822]

HeaderID
: An offset into a MIME object where a specific header starts. As the position in a MIME object is unique, this value is also used as the ID to a specific header. As long as the MIME object does not change in any way this ID persisists across connections and servers.

See Index. (Section 3)

Header Value ID
: Related to Header ID. An offset into a MIME object where a specific header value starts. As the position in a MIME object is unique, this value is also used as the ID to a specific header value. As long as the MIME object does not change in any way this ID persisists across connections and servers.

  See Index. (Section 3)

  See Header ID. (Section 3)

Index
: This wire protocol transmits all or part of MIME objects. Various parts can be referenced by an offset into the object. This is an index into the MIME objects. A client may request an index be used when opening a folder.

  Note: None of these index values are guaranteed to persist across re-connections to the server, as other clients may have altered the contents.

LID
: List ID. In operations that require a list or set of data. This ID uniquely identifies which list or set is in context.

Media Type
: Each MIME object has a media type that identifies the content of the object. This specification does not add, remove, or alter any MIME media type;

MIME
: This protocol transports MIME objects. This specification does not remove or alter any MIME objects;

Offset
: Unless otherwise specified, an offset is an unsigned 32-bit integer in network byte order.

Packet
: A packet is a blob of data that has a header (its length) followed by a Phoenix command with all of its values and parameters. Packets flow in both directions and asynchronously. Commands can be sent while still waiting for other replies. Each endpoint may send commands to the other endpoint without having to be prompted to send information.

Parameter
: Most commands have values that are associated with them. These values are called parameters. For example, the create folder command has the name of the new folder to be created as a parameter.

ProtoCmd
: A general term for all protocol commands. This also includes commands that do not fall into one of the other categories described here in this definitions section.

SEQ, Command Sequence, CMD_SEQ
: Each command has a unique identifier, a sequence number. All replies to a command include the same sequence number as the original command. In this way replies can be matched up with their original command.

SSL
: For the purpose of this specification, SSL is interchangeable with TLS. This document uses the term TLS. The sample implementation uses both SSL and TLS because the legacy UNIX, Linux, Windows, and OpenSSL code uses the term SSL in cases where is it TLS.

TLS
: A way of securely transporting data over the Internet.

  See [RFC8446]

XDR
: RFC-4506 specifies a standard and compatible way to transfer binary information. This protocol uses XDR to transmit a command, its values and any parameters and replies. The MIME data, the payload, is transported as XDR opaque, and is unmodified.

  Note: XDR transmits data in 32-bit chunks. An 8-bit value is transmitted with the lower 8-bits valid and the upper 24 bits set to zero. A 16-bit value is transmitted with the lower 16-bits valid and the upper 16 bits set to zero.

  So many of these protocol elements pack one or more of its parameters into one 32-bit value. As defined in each section. In many cases pseudo code is shown on how to pack the data and create the protocol element.

  See Section 3 of [RFC4506]

4. Commands Summary

The endpoint that initiates the connection is called the client. The endpoint that is connected to, is called the server. The client is the protocol authority, and the server responds to client commands as configured or instructed by the client.

This section provides an overview of the basic commands. Each command has a detailed section in this specification.

When a command is sent to the remote endpoint and received, the remote endpoint determines if the connection is authenticated or authorized to perform the command. If not supported, or not authorized, a NotSupported command is send as a reply. The NotSupported command sent back has the same Sequence number that was in the original command.

Many commands are only valid after authentication.

When the client connects to a server it immediately sends its pre authentication capabilities to the server. Or an Auth command.

When the server gets a new connection followed by a pre authentication capability command, it immediately sends its pre authentication capabilities to the client.

When the client and server have had a relationship, the client may send an Auth Command to initiate the authorization and does not send its pre authentication capability list to the server. It then waits for the Auth reply from the server.

- If the client gets an Auth reply that is positive, it sends its post authentication capability list to the server.
- If the client gets an Auth reply that is negative, it sends its pre authentication capability list to the server.

When a servers first received packet is a Auth command, It processes the Auth command and sends the Auth reply.

- If the Auth reply is positive, then it also sends it post authentication capability list.
- If the Auth reply is negative, then it sends its pre authentication capability list to the client.

A server may automatically send its pre authentication capability list to the client upon initial connection. Or it may wait to see if it gets a pre authentication capability list, or an Auth command.

If the client sends an Auth command as its first packet, it may get the pre authentication capability from the server before the Auth reply. Simply process both.

4.1. Protocol Commands Summary

In addition to the protocols listed in this specification. Additional protocols and commands can be added in the future. They must follow the same framework listed here.

This protocol connects two endpoints over a network and facilitates the secure and authorized transfer of MIME objects.

This is a binary protocol. The payload can be anything, text or binary. This protocol was designed to reduce the number of back and forth requests and replies between the client and server. By using XDR as the format for transferring binary control information it is portable to any computer architecture. Appendix XXX has the rpcgen definition for the protocol defined in this specification.

After the connection is successful and authenticated, ether endpoint may send commands to the other endpoint. When the server initiates an unsolicited command, it could be a any kind of notification or message for the client side application or the user. It could be reporting errors or updates to previous client initiated commands.

All commands initiated from the client have even numbered command sequence numbers. All commands initiated from the server have odd numbered command sequence numbers.

Some commands expect a command reply. Other commands do not expect a command reply. An example of a command that expects a reply is the ping command. An example of a command that does not expect a reply is the keep-alive command. Conceptually there are two kinds of commands:

Directive commands: A directive type command expects the other endpoint to process the command and possibly reply with some results. An example could be: Send me an index of my emails in my InBox. The client would expect a result. Another example is a bye command, once sent, no reply is expected.

Request commands: A request type command may or might not have any reply. For example, a keep-alive command is a request to not timeout and has no reply. And a send new email notifications command would expect zero or more replies and it would not require them, as they might not happen.

These are not specific protocol entities, these concepts will be used to describe the expected behavior when one of these are transmitted.

4.1.1. Packet Summary

All commands are sent in a packet. A packet has two parts:

1. The packet header.
2. The packet body.

The packet header has one value, the total length of the packet body, and payload sent as an unsigned 64-bit integer in network byte order. The length does not include its own length. It is the total length that follows the length value.

The packet body is divided into three parts:

1. Command sequence (SEQ).
2. The Command (CMD).
3. The command specific data (Payload).

4.1.1.1. Command Sequence Number (SEQ)

The Command SEQ is a 32-bit unsigned integer sent in network byte order. This SEQ is an even number when initiated from the client, and an odd number when initiated from the server.

The first SEQ value sent from the client is zero (0) and is incremented by two each time.

The first SEQ value sent from the server is one (1) and is incremented by two each time.

In the event an endpoint command SEQ reaches its maximum value, then its numbering starts over at zero (0) for the client and one (1) for the server. An implementation must keep track of outstanding commands and not accidentally re-issue the same SEQ that may still get replies from the other endpoint.

4.1.1.2. The Command (CMD)

The command is a predefined enumerated 32-bit unsigned integer sent in network byte order. The value (in hex) 0xFFFFFFFF is reserved for extensions if the 32-bit range is exhausted.

4.1.1.3. The Payload (Payload)

The payload has no predefined length, other what what is specified for the CMD in the packet. It could be zero to vary large in size. It could be opaque data, or it could be data that is XDR encoded. The contents are specific to the CMD specified in the in the packet body.

4.1.2. Packet Reply Summary

All replies to a command are also a command packet. They contain the same command SEQ and command as the original packet. The endpoint recognizes it is a reply because:

- The command SEQ matches one that is waiting a reply.
- When the client gets an even numbered SEQ, it can only be a reply.
- When the server gets an odd numbered SEQ, it can only be a reply.

Some commands have zero to many replies. Each of these multiple replies contains the same SEQ as the original command. An example, the client sends a request to be notified when new email arrives and uses command SEQ 20. Each time a new email arrives, a reply will be sent from the server with a command SEQ of 20. And over time, the client may get many with a SEQ of 20 as new emails arrive on the server.

4.2. Administration Commands Summary

Implementations are not required to implement any ADMIN command. A client will know the server supports one or more ADMIN commands when it gets its post authentication capability command from the server.

Administrative command can be used to configure, audit, and manage the remote endpoint. Administrative command can be used to configure, audit, and manage user access.

4.2.1. Administration Capability Definitions

Implementations MUST NOT send the ADMIN capability in the pre authorization CAPABILITY list.

Implementations that support any administration command MAY include ADMIN capability in the post authentication CAPABILITY list. An implementation may decide that only specified and authorized users may issue administrative commands and send only those authenticated users the ADMIN capability.

The ADMIN capability include the list of ADMIN commands the user is allowed to perform. For example, if a user only has permission to only view user lists, then only the USER_LIST ADMIN capability will be provided.

The capability name is also the command name to use when invoking that capability.

When a user attempts to send a commmand they are not authorized to send, the remote endpoint will reply with a NotSupported command with its sequence number set to the sequence number from offending command.

4.2.2. Administration of users.

The following operations are defined for administration.

Command and Capability Name	Brief Description.
USER_CREATE	May create a new user. And also the command to create a user.
USER_DELETE	May delete a user. And the command to delete a user.
USER_RENAME	May rename a user. And the command to rename a user.
USER_LIST	May list users and their capabilities. And the command to list users.
USER_PERMISSIONS	May update other users permissions. And the command to view and set user permissions.
SERVER_SHUTDOWN	May shutdown the server. And the command to shutdown the server.
SERVER_LOGS	May view the server logs. And the command to view server logs.
SERVER_KICK_USER	May logout a user. And limit when they can use the server again. And the command to kick and limit a user.
SERVER_MANAGE_BANS	May manage IP and user bans. And the command to manage ban users an IP addresses.
SERVER_VIEW_STATS	May view server statistics. And the command to view statistics.
SERVER_CONFIGURE	May configure the server. If sent with a VIEW_ONLY parameter, then the user may only view the configuration information. And the command to view and alter the server configuration information.

Table 1

4.3. Authentication Commands Summary

TODO

4.4. Calendar Commands Summary

These command are based on iCalendar and iTIp.

4.5. Capability Commands Summary

This section ...

4.6. EMail Commands Summary

These commands allow for the fetching and submission of EMail messages

4.7. File and Folder Commands Summary

The file operations (FileOp) have protocol names. Here are their protocol names and a breif description.

Implementations are not required to support any or all of these commands.

Op Name	Brief Description.
FOLDER_CAPABILITY	When sent as a command, request the list of folder commands supported. When sent as a reply, includes the list of folder commands supported.
FOLDER_CREATE	Create a new folder. Also the name of the capability for this permission.
FOLDER_COPY	Copy a folder. Also the name of the capability for this permission.
FOLDER_DELETE	Delete a folder. Also the name of the capability for this permission.
FOLDER_RENAME	Rename a folder. Also the name of the capability for this permission.
FOLDER_METADATA	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.
FOLDER_MOVE	Move a folder. Also the name of the capability for this permission.
FOLDER_OPEN	Open a folder and get information about the folder and files in the folder.
FOLDER_SHARE	Share a folder. Also the name of the capability for this permission.
FOLDER_LIST	List folders and files. Also the name of the capability for this permission.
FILE_CREATE	Create a new file. Also the name of the capability for this permission.

Op Name	Brief Description.
FILE_COPY	Copy a file. Also the name of the capability for this permission.
FILE_DELETE	Delete a file. Also the name of the capability for this permission.
FILE_RENAME	Rename a file. Also the name of the capability for this permission.
FILE_METADATA	Get, set, and update information associated with the folder. File meta data is also returned with the FOLDER_OPEN command.
FILE_MOVE	Move a file. Also the name of the capability for this permission.
FILE_SHARE	Share a file. Also the name of the capability for this permission.
FILE_GET	Get a file. Also the name of the capability for this permission.
FILE_MODIFY	Modify the contents of an existing file. Also the name of the capability for this permission.

Table 2

4.8. KeepAlive Command Summary

The KeepAlive command is sent to the server from the client. It requests the server not time out. The server may honor or ignore the request.

The Phoenix protocol is designed to transfer data and a server may handle a small subsets of what is possible. Which is why the server decides what is an important command while determining idle timeout.

When the server sends the post authentication capabilities to the client, it includes an IdleTimeout capability that includes the number of seconds it allows for idle time. If no significant action has been taken by the client, as determined by the server, in that time the server may timeout and close the connection.

The KeepAlive command tells the server that the client wishes the server not to time out as long as a KeepAlive or other command is sent to the server before IdleTimeout seconds have passed.

An IdleTimeout capability can be a positive number, zero, or a negative number.

- A positive number is the maximum idle time in seconds before the server terminates the connection.
- When the IdleTimeout is zero (0), the server does not timeout.
- When the IdleTimeout is less than zero (< 0), it means it ignores KeepAlive and it will idle out in the absolute value of the IdleTimeout value in seconds. For example, a value of (-300) means it will ignore KeepAlive and timeout when the server determines nothing significant has happened in 5 minutes (300 seconds).

Servers that are not threaded or can not reply to simultaneous or overlapping commands, MUST set their IdleTimeout to zero (0) or a negative number.

Clients MUST NOT send KeepAlive commands to a server that has an IdleTimeout of zero (0) or negative (< 0).

Clients MUST NOT send KeepAlive commands to the server until at least 75% of the idle time has passed since the last command has been sent to the server.

A server may terminate a connection if the server implementation determines that KeepAlive commands are arriving to quickly.

4.9. Ping Command Summary

The ping command only sent when the client implementation has determined it has waited too long for a command reply. The ping command is only initiated from the client. It is not valid for the server to send a ping command to a client.

The ping command MUST NOT be the first command sent to the server. It should only be sent when the client implementation determines it has waited too long for a reply.

If the server supports the ping command, then a PING capability is sent in the pre authentication capability command.

Sometimes servers are unavailable and can go down. A server could be down for maintenance, or in a shutdown mode. It might limit the number of simultaneous connections. It might be very busy. The packets might not be making it to the server because of network issues.

When a ping command is received by the server:

- When the server did not send PING capability to the client. Then the server replies with a NotSupported packet with the sequence number the same as in the ping command.
- When the server has not yet received an authentication command, the server replies with a NotSupported packet with the sequence number the same as in the ping command.
- When the server has received an authentication command, and has not yet replied to an authentication command. Then the server sends a ping reply, with the same sequence number that was in the ping command. This could happen when the client implementation had determined it has waited too long for an authentication reply.
- When the client is authenticated, and when the server is available for processing commands. Then the server replies with a ping reply with the same sequence number. This could happen when the client implementation had determined it has waited too long for an expected reply.

If the server is alive and not available, the server will reply with a NotSupported command, with its sequence number set to the sequence number in the ping command.

If a connected and authenticated client has been waiting for a reply or for some other reason needs to determine if the server is still available. It can send a ping command. If the server is still available, it sends a ping reply. If it is no longer available for any reason, it sends a NotSupported reply.

Endpoints MUST NOT send a ping command if they are awaiting the results of a previously sent ping command.

Endpoints MUST NOT send more than two ping commands per minute.

Clients and servers must give priority to ping commands. If possible, reply as soon as it receives the command.

The server MAY consider too many ping commands as a malfunctioning or malicious client and terminate the connection.

Servers that are not threaded or can not reply to simultaneous or overlapping commands, MUST NOT include PING in their capability command.

4.10. S/MIME Commands Summary

ToDo

5. Meta Data

In this specification a file and a MIME object are used interchangeably. Meta Data is data that is associated with the MIME object and not contained within the MIME object. Meta Data should never be stored in the MIME object as altering the MIME object would invalidate the index information and can invalidate digital signature and encryption information.

Meta Data for the folder and MIME objects is returned in a FOLDER_OPEN, FILE_METADATA, or FOLDER_METADATA command. Meta Data can be set and updated by the client using FILE_METADATA or FOLDER_METADATA commands.

Most are 8-bit boolean values that are set to false (0x00) or true (0x01). A value that does not exists is the same as a false.

Meta data can be global to the object. That is once tagged (or not tagged) the attribute shows up for all users. Or it can be user specific meta data. User specific meta data does not show up for other users.

Many have the same or similar name and meaning as they do in IMAP [RFC9051].

5.1. Meta Data - Answered

When true, the object has been replied to by the client. This has the same meaning as \Answered does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.2. Meta Data - Attributes

This object has been tagged with special attributes. It is a list of strings with matching values.

User defined attributes MUST start with "X-". These are not portable between implementations and no attempt should be made to copy these between implementations.

Non user defined attributes are described in other sections or specifications.

This can be user specific meta data or global meta data. See the specific attribute documentation.

5.3. Meta Data - Deleted

When true, this object has been marked as deleted and has not yet been expunged. This has the same meaning as \Deleted does in IMAP.

For shared objects, an expunge removes the user from shared access to the file. And the actual expunge is only processed when all shared users have expunged the object.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.4. Meta Data - Draft

When true, this object is incomplete and not ready.

This has the same meaning as \Draft does in IMAP.

This value can be set and unset. This is user specific meta data.

5.5. Meta Data - Flagged

An object has been tagged as important. This is the same as the IMAP \Flagged value.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.6. Meta Data - Forwarded

This has the same meaning as $Forwarded does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.7. Meta Data - Hide

With NotExpungable objects, the user may wish to not view the object. In these cases the attribute Hide can be set. The attribute does not effect the view of other users.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.8. Meta Data - Junk

This has the same meaning as $Junk does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.9. Meta Data - MDNSent

This value can be set and unset. This has the same meaning as $MDNSent does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.10. Meta Data - NoCopy

When true, this MIME object can not be copied.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

5.11. Meta Data - NotJunk

This has the same meaning as $NotJunk does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.12. Meta Data - NotExpungable

The mime object can not be marked for delete or expunged. It could be because it is an historical record that will never be expunged, or other reason.

A client implementation could use the Hide attribute to not show the object to the user.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

5.13. Meta Data - Phishing

This has the same meaning as $Phishing does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.14. Meta Data - ReadOnly

The MIME object associated with this attribute can not be altered, deleted, moved, or renamed. It can be copied, unless the NoCopy meta tag is also applied.

This value can be set and unset by the owner of the file or folder. This value can not be unset by non owners. This is global meta data.

Setting of this to false may fail if the file or folder is stored on read-only media. When the file or folder is stored on read-only media, this MUST BE set to true.

5.15. Meta Data - Shared

The MIME object associated with this attribute is shared and is also often tagged with the ReadOnly meta data tag.

This value can not be set and unset by the owner.

If copying of the file or folder is allowed, then the shared attribute is removed when copied.

This file or folder will only be expunged when all of the users with shared access have deleted and expunged it.

5.16. Meta Data - Seen

This has the same meaning as \Seen does in IMAP.

This value can be set and unset. This is user specific meta data because it also applies to shared folders and files.

5.17. Meta Data - MDNData Attribute

This Meta Data Attribute is only visible to the owner of the object for which MDN has been set.

This is a list of recipients email address that that are on the distribution list effected by the MDN.

The format of an MDNData record is a header, that has:

- COUNT: A 32-bit unsigned integer in network byte order indicating how many were on the distribution list for this MDN.

- SENT: The UTC timestamp as a 64-bit unsigned integer in network byte order of when the MDN reply was sent.

Followed by COUNT detail records:

- UTC: The UTC timestamp as a 64-bit unsigned integer in network byte order of when the MDN reply was received. Set to zero if not received.
- Email-Address: A PhoenixString with the associated email address of the user that has or has not returned the MDN.

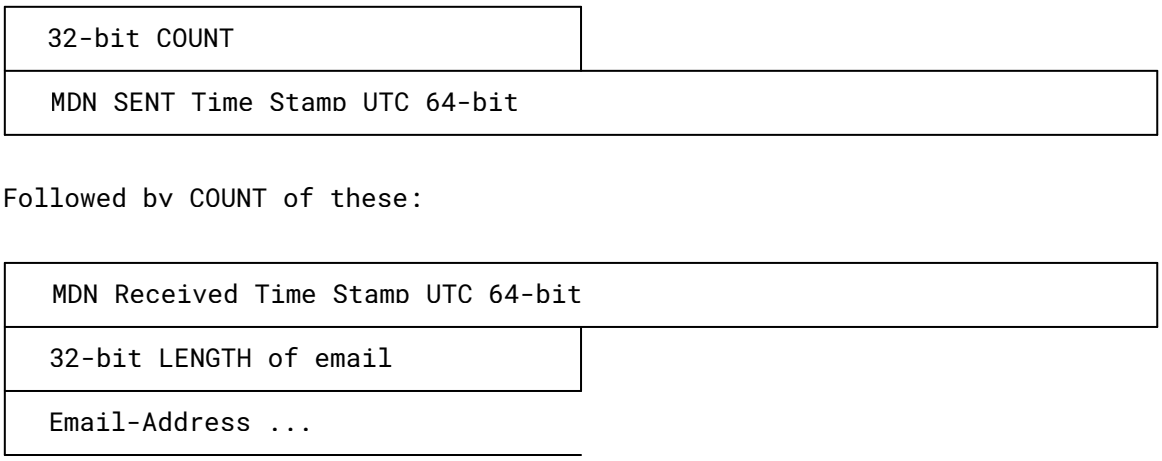


Figure 2: MDNData Attribute

5.18. Meta Data with Shared Objects

When a server implementation allows shared objects, the meta data returned to the client may be different depending on the authenticated user. Some users may have read only copies, other may be able to delete the object.

When a shared object is deleted, it is marked as deleted for only the user that issued the delete.

When a shared object is expunged, its access is removed for the user that issued the expunge. After all users have expunged the object, then it is removed by the server.

There are two kinds of expunge for shared objects. Force and delayed.

- Forced:

  A forced expunge can be the result of security policies at the server, site, or administrators discretion. This also is how timed messages are deleted.

  In order for a shared object that is expunged to not force an immediate re-index for all clients, the server sends an expunge to all clients, where the client MUST immediately make the object not show to the user and MUST invalidate any cached or memory copy of the data.

Then when convenient, the client can do a re-index of the folder. When a user is viewing the object when an expunge arrives, the client must inform the user that the data is no longer available and replace the user view of the data with an empty object or move the view to another object.

Server implementations must prioritize forced expunge notices to the clients and immediately reject all attempts to read, view, copy, or access meta data.

- Delayed:

The user is informed the MIME object is no longer available. The client implementation may continue to show the object. The client may copy the MIME object, unless tagged as NoCopy.

The next time the client does an expunge the object will be expunged from the client.

When a client application closes, all delayed expunges MUST occur at exit.

When a client applications starts the client MUST check for delayed expunges that have not been processed and expunge them and not allow the user to see them.

Server implementations must reject attempts to fetch or view a folder or file or any of its meta data when an expunge has started, and not yet completed.

6. Over the Wire Protocol Detail

This section specifies the details of what is transmitted over the network.

All protocol data transmitted between the endpoints is sent in network byte order.

All payload data transmitted between the endpoints is sent in original format. The payload consent is seen as an opaque blob of data within a command packet.

When a command packet is received by ether endpoint it: (1) Checks the command sequence number to determine if it is a reply or not. (2) If it is a reply, it looks at the command and dispatches it to the implementations commands reply code. (3) If is not a reply, it looks at the command and dispatches it to the implementations command code.

A command and all of its replies, use the same format as described here.

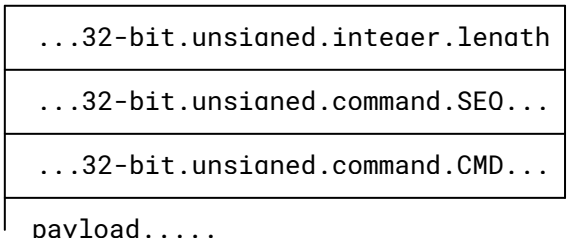
A packet has a 32-bit unsigned integer in network byte order that is set to the octet count of all of the data that follows this length value. The shortest packet is 16 octets in size, with a length value set to 8. With 8 octets for the length, and 8 octets for the packet.

Followed by a 32-bit unsigned integer in network byte order that is the command sequence number.

Followed by a 32-bit unsigned integer in network byte order that is the command.

Followed by zero or more octets of payload data.

There is no space, padding, or line endings between the parts of the packet. The payload is sent without any modification and is not encoded or transformed in any way. A packet is shown here vertically only to aid in readability.



ABNF:

```
CommandHeader : Length:32 Sequence:32 CMD:32

Command : CommandHeader Payload
```

The payload size and format varies for each command. The details of the payload content, and the format of that content, is described in each specific CMD section.

An implementation can send, receive, and dispatch packets within its implementation by looking at the length, SEQ, and CMD, then passing the payload to code that can handle that payload.

- Read in a 64-bit value. - Convert the value from network byte order, to host byte order. This is the total length of the data that follows. - Read in length octets into the packet payload. - Get the 32-bit value in the payload, it is the SEQ in network byte order. - Convert the SEQ from network byte order, to host byte order. - Get another 32-bit value in the payload, it is the CMD in network byte order. - Convert the CMD from network byte order, to host byte order. - Dispatch the CMD with SEQ and all of the data that follows to implementation

The following is pseudo code that explains how processing incoming XDR data can be handled:

```
// Where:
//  uint32_t, is a 32-bit unsigned integer.
//  uint8_t *, is a pointer to 8-bit data.
//  XDR, is an XDR object.
//
//  CmdPacket, is an object that represents all commands
//  and replies.
//
//  NOTE: See the sample implementation.
//
uint32_t   NetLength;
uint32_t   PacketLength;
uint8_t *  Data;
```

```
uint8_t *  DataPointer;
XDR        Xdr;
CmdPacket  Packet;

// Read the length and convert to host byte order.
//
read(FromClientSocket, &NetLength, sizeof(uint32_t));
PacketLength = ntohll(NetLength);

// Allocate PacketLength data, and read it.
//
Data = new uint8_t[PacketLength]
DataPointer = Data;

// Initialize the XDR deserializer.
//
xdrmem_create(&Xdr, Data, PacketLength, XDR_DECODE);

// Decode the received data into a Packet.
//
if (xdr_CmdPacket(&Xdr, &Packet)) {

  // If the lowest bit is set, it is an odd number.
  //
  if (Packet.Sequence & 0x01) {
    SequenceIsEvenNumber = false;
  } else {
    SequenceIsEvenNumber = true;
  }

  // The client sends even numbered sequences, and the server
  // sends the same even numbers sequence in the reply to
  // the command.
  //
  // If a client gets an odd numbered sequence, it is a command
  // from the other endpoint.
  //
  // The server sends odd numbered sequences, and the client
  // sends the same odd numbers sequence in the reply to
  // the command.
  //
  // If a server gets an even numbered sequence, it is a
  // command from the other endpoint.
  //
  if (WeAreTheClient) {
    if (SequenceIsEvenNumber) {
      DispatchReply(Packet);
    } else {
      DispatchCommandFromOtherEndpoint(Packet);
    }
  } else {
    if (SequenceIsEvenNumber) {
      DispatchCommandFromOtherEndpoint(Packet);
    } else {
      DispatchReply(Packet);
    }
  }
}
```

7. Index

7.1. Interested Headers

Some implementation may wish to specify which MIME headers it wants to get in the index supplied by the server. This is done as part of the folder selection command which can supply a list of desired headers. Or it can specify a list ID that has already been transmitted. When none are supplied, no header index values will be returned.

This list can be the same for all folders, or unique to specific folders. The client generates a list of interested headers and sends an Interested Headers list or list ID to the server when selecting a folder.

7.1.1. Index Operation (IndexOP)

An Index Operation (IndexOP) is an 8-bit unsigned integer in network byte order. There are only two (2) values, set list (0), use existing list (1).

There are two (2) types of index operations (IndexOP) requests. These are 8-bit unsigned integer in network byte order. And the IndexOP values are:

- IndexOP = 0

  Used to set a list of body MIME object, and Body Part, interesting headers the client cares about.

- IndexOP = 1

  This indicates that LID is an existing list number to use.

7.1.2. Header ID (HID)

A Header ID (HID) is an 8-bit unsigned integer in network byte order. The client assigned this 8-bit number to a header name, then the client and server references it by ID in packets and replies. A client can have up to 254 interested headers per connection. The value 255 is reserved for expansion.

7.1.3. List ID (LID)

A List ID (LID) it an 8-bit unsigned integer in network byte order. It is used in requests and replies to refer to the interested headers list. A client can have up to 254 lists per connection. The value 255 is reserved for expansion.

Restrictions:

- The list IDs are unique to the connection and do not persist across connections.
- No two lists can have the same ID during a connection.

7.1.4. Protocol Format

When a folder is opened with the FOLDER_OPEN command, it is followed by a list of interested headers it wants back in the folder open reply. The list can be empty (a single 32-bit unsigned integer set to zero), or contain what follows:

The client sets these 3 values into one 32-bit unsigned integer in network byte order as shown in Figure 3, where:

- IndexOP is used to specify headers the client wishes the server to index in MIME objects. It must be a zero (0) to set the list definition.
- LID is the list ID of the list that client is defining. An 8-bit value.
- HdrCnt is set by the client to the number of headers in the list. This is an 8-bit value from 0 to 254 with 255 reserved for expansion.

  This is a 32-bit unsigned integer in network byte order.

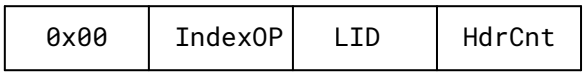


Figure 3: Setting the Interest List - Header

ABNF:

```
Interest Header:32 = 0x00:8 IndexOP:8 LID:8 HdrCnt:8
```

7.1.4.1. Interested Headers - Single Entry

Followed by the header request information is one or more of these single header entries. One sent for each unique header to be placed into the header index. As shown in Figure 4, where:

- HID is the client assigned unique header ID for the named header. This is an 8-bit unsigned integer.
- STRING LENGTH is the octet count of the string. Any terminating zero (0) is not counted in the length.

  This is a 24-bit unsigned integer in network byte order.
- THE HEADER NAME is the characters that make up the MIME header name that is interesting without including any terminating zero (0). These are a sequence of 8-bit unsigned integer values in network byte order.

  They are packed into a 32-bit unsigned integer in network byte order.

  When THE HEADER NAME is not a multiple of 32-bits in length, the remaining bits are unused and shown as zero (0) in this specification.

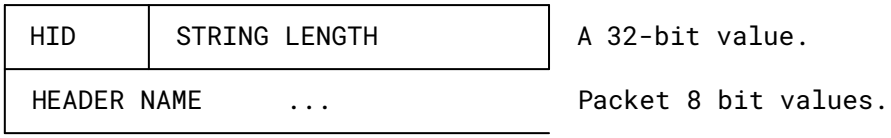


Figure 4: Setting the Interest List - Contents

ABNF:

```
SingleEntry     = SingleHeader:32 HeaderName822
SingleHeader:32 = HID:8 StringLength:24
```

7.1.4.2. Interested Headers - Use Existing List

When the IndexOP flag is set to one (1) then it is followed by an existing list ID number.

LID, the list ID of an already transmitted list to be used.

This is sent as a 32-bit unsigned integer in network byte order.

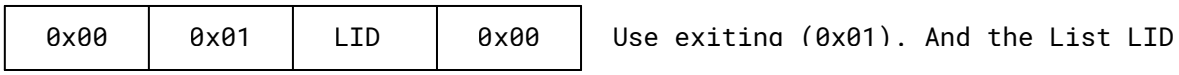


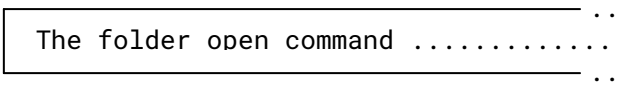
Figure 5: Using Existing Header Index by List ID (LID)

ABNF:

```
UseExistingOp:8 = 0x01:8
UseExistingList:32 = 0x00:8 UseExistingOp:8 LID:8 0x00:8
```

7.1.4.3. Example: Setting the Interested Header List

This is an example of the client sending an interesting header list to the server. The client is asking for the index values for the following MIME headers (1) From, and (2) Subject. And for the following Body part headers (1) Content-Type.



Immediately followed by:

	IndexOP	LID		2 MIME Object Headers included.
(a)	0x00	0x00	0x00	0x02	Start of MIME object List
(b)	0x00	0x000004			Header (0), Length (4)
(c)	0x46	0x72	0x6f	0x6d	"From"
(d)	0x01	0x000007			Header (1), Length (7)
(e)	0x53	0x75	0x62	0x6a	"Subject"
	0x65	0x63	0x74	0x00	
(f)	0x00	0x00	0x00	0x01	Start of Body Part List
(g)	0x02	0x00000c			Body Header (2), Length (12)
(h)	0x43	0x6f	0x6e	0x74	"Content-Type"
	0x65	0x6e	0x74	0x2d	
	0x74	0x79	0x70	0x74	

Figure 6: Example Setting a List

Where:

- (a): A 32-bit unsigned integer in network byte order as described in Figure 3.

  The first 8-bits are zero.

  The IndexOP of zero, which means defining a list.

  And in this example two (0x02) MIME object headers are requested to be indexed, "From", and "Subject".

- (b): A 32-bit unsigned integer in network byte order as described in Figure 4.

  The header ID that the client and server will use to to identify the "From" header name will be zero (0) in this example.

  The length of the string "From" is four (4) and its length is the lower 24 bits of this entry.

- (c) A series of 8-bit unsigned values packed into one or more 32-bit unsigned integers in network byte order.

Each 8-bit value is the value of the letters in "From". As "From" is a multiple of 32-bits, no padding is done.

- (d): A 32-bit unsigned integer in network byte order as described in Figure 4.

  The header ID that the client and server will use to to identify the "Subject" header name will be one (1) in this example.

  The length of the string "Subject" is seven (7) and its length is the lower 24 bits of this entry.

- (e) A series of 8-bit unsigned values packed into one or more 32-bit unsigned integers in network byte order.

  Each 8-bit value is the value of the letters in "Subject".

  As The length of "Subject" is not a multiple of 32-bits, the remaining bits are ignored. Shown as zero in this example.

- (f) The two MIME objects headers are done, start of Body Part headers, and there is one (1) of them. IndexOP and LID are not used here.
- (g) The second header will be identified as three (3). The first body part header is 12 octets long (0xc): 'Content-Type'.
- (h) The value of the characters for 'Content-Type'.
- (i) The rest of the value of the characters for 'Content-Type'.

7.2. PhoenixString

This protocol references the strings by octet offset into the entire message. All strings can be referenced by using a total of 8 octets. A Phoenix string consists of two parts:

- Offset: A 32-bit unsigned integer in network byte order. The octet count to the start of the string with zero being the first octet in the message.
- Length: The length in octets of the string. A 32-bit unsigned integer in network byte order.

A Phoenix string over the wire is 8 octets in size.

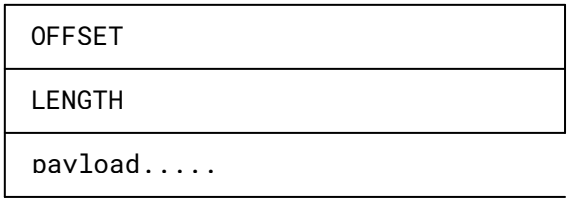


Figure 7: Phoenix String Format

ABNF:

```
PhoenixString = Offset:32 Length:32 String
```

7.3. MIME Folder Index

In this specification, a MIME folder is also called a folder. And can be files containing MIME objects on a disk that have a defined order, or sequence of MIME objects in one file.

A folder index is a summary of the contents of a MIME folder. It may include the basic header information. It does include location information provided as the octet count to the start of the beginning of the related target data.

- An index is an unsigned 32-bit integer in network byte order.
- A length is an unsigned 32-bit integer in network byte order.

For example, if a MIME folder contains 100 MIME messages, then the folder index will have 100 message indexes. Each message will have header indexes for the interested headers. Each message index will contain 1 or more body part indexes. Each body part will have header indexes with zero (0) or more entries.

7.3.1. Folder Index Header

A folder index consists of:

- The entire length of the index as a 32-bit unsigned integer in network byte order of what follows this value. Allowing the recipient of this index to do one read and process later.
- The number of message indexes in this folder index. As an unsigned 32-bit integer in network byte order.

The index header is 8 octets, that is followed by the each message index:

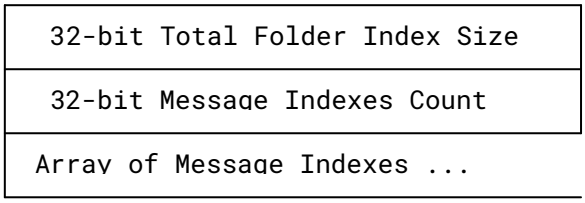


Figure 8: Folder Index

ABNF:

```
FolderIndexHeader = FolderIndexSize:32 MessageCount:32 ArrayOfMsgIndex
```

The header is followed by an array of message indexes. They are an ordered list of references to each message. In the order they appear in the folder:

7.3.2. Message Index

- A 32-bit unsigned integer in network byte order that is the offset into the folder of the message. A Message offset is unique in a MIME folder, it is used both as an offset into the MIME folder, and as a unique ID within a MIME folder for a message.
- An a length of the message as a 32-bit unsigned integer in network byte order.

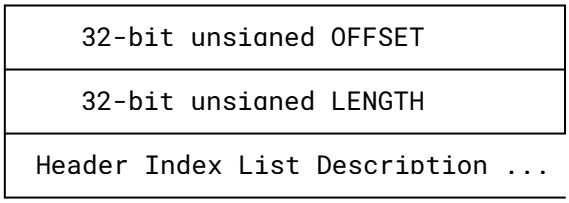


Figure 9: Message Index

ABNF:

```
MessageIndex = OfffsetIntoFolder:32 MessageLength:32 ArrayOfHeaderIndex

ArrayOfMsgIndex = MessageIndex*
```

For each message index is an ordered list of interested headers. The interested header list is assignable by the client and body part indexes. It consists of offsets to the interested headers and associated value. Each interested header can be indexed with nine (9) octets. and consists of:

7.3.3. Header Index

- ID-CNT: A count of matched headers. Only matched headers will be included. If they are not included, no such header existed in the object.
- The number of body parts in this object. An unsigned 8-bit number. With MIME, body parts may contain body parts.

  Any MIME preamble and epilogue are not counted as body parts A preamble, if it exists, can be easily be calculated as it starts as the first octet after the header area. And the epilogue, if it exists, can be calculated as starting as the first octet after the last MIME boundary.
- Followed by an array of ID-CNT 8-bit client assigned HID values that matched. Padded to round up to 32-bits. The unused bits are ignored and shown as zero in this specification.

A single header index consists of the list description, followed by the index values. There are two header indexes in each Message index.

1. The first is for the MIME object itself.

2. The second is for the objects Body Parts. This part will not exists exist when it is an RFC-822 style message or has no body parts. Followed by the header index. This second part also include an offset to the start of the body part itself in the MIME object.

A list description is one 8-bit result count, followed by the list of matching header ID's (HID).

If the list description is not a multiple of 32-bits then padding is added and the extra are ignored and shown as zero in this specification.

```
-Meta-Data-: Seen,Answered,$NotJunk
```

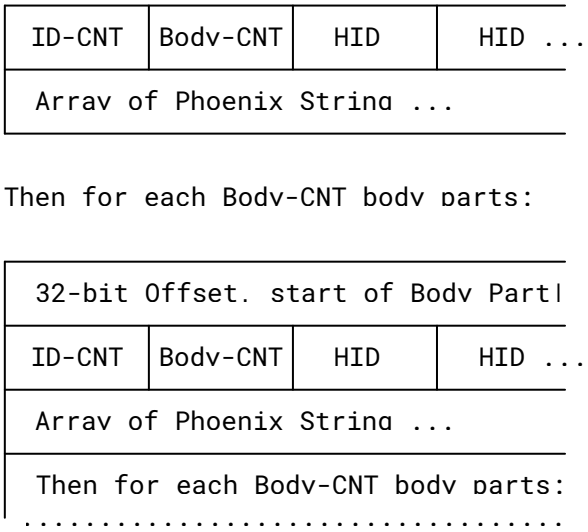


Figure 10: Header Index

ABNF:

```
HeaderIndex          = HeaderIndexHeader:32 ArrayOfHID*
                       PhoenixString* BodyPartIndex*

                       ; One HID (HeaderID) for each match header
                       ; in the LID provided. Padded out to multiples
                       ; of 32-bits.
HeaderIndexHeader:32 = ID-CNT:8 Body-Count:8
                       / (HID HID)
                       / (HID 0x00:8)
                       / (0x00:8 0x00:8)

ArrayOfHid           = HIDEntry*

BodyPartIndex        = BodyPartOffset:32 HeaderIndexHeader:32 PhoenixString*
                       BodyPartIndex*

ID-CNT:8               ; The number of headers found in the
                       ; MIME object and requested in the interested
                       ; header list.

Body-CNT:8             ; The number of body parts in the object

                       ; Padded out to multiples of 32-bits.
HidEntry             = (HID:8 HID:8 HID:8 HID:8)
                       / (HID:8 HID:8 HID:8 0x00:8)
                       / (HID:8 HID:8 0x00:8 0x00:8)
                       / (HID:8 0x00:8 0x00:8 0x00:8)
```

Where:

HeaderIndex: The header index starts with a 32-bit unsigned integer in network byte order, the HeaderIndex:32.

HeaderIndex:32: Contains 0, 1, or 2 HID values. They are in the order found in the object.

ArrayIfHID: Keeps repeating until all of the headers in the list have been found in the message. The last one pads with zeros when needed.

BodyPartIndex: When the object has body parts, there will be a BodyPartIndex for each body part, in the order they are in the object. The first 32-bits are the offset to the start of the body part. This does not include any boundary.

Body parts themselves may contain body parts, they are recursively included as needed.

7.3.4. Header Index Example 1

For example, if the client requested MIME object indexes for the "From", "Subject", "To", "Message-ID", "Content-Type", "MIME-Version", and "Date" header values.

Assume this is an RFC-822 message with no body parts. So the body part header index has a count of zero (0). And the HID values assigned by the client when opening the folder are:

- From: 0

- Subject: 1
- To: 2
- Message-ID: 3
- Content-Type: 4
- Data: 5
- MIME-Version: 6

In the Message each line is terminated with a carriage return and line feed:

```
From: Doug@example.com
To: Notices@example.com, Supervisors@example.com, Dave@example.com
Date: Thu, 06 Feb 2025 20:29:35 +0000
MessageID: <7324e0b9-f6dc-3c9b-a02f-0b2b824e863c@example.com>
Subject: A new draft of Phoenix has been published.
Content-Type: text/plain

A new draft has been published.
```

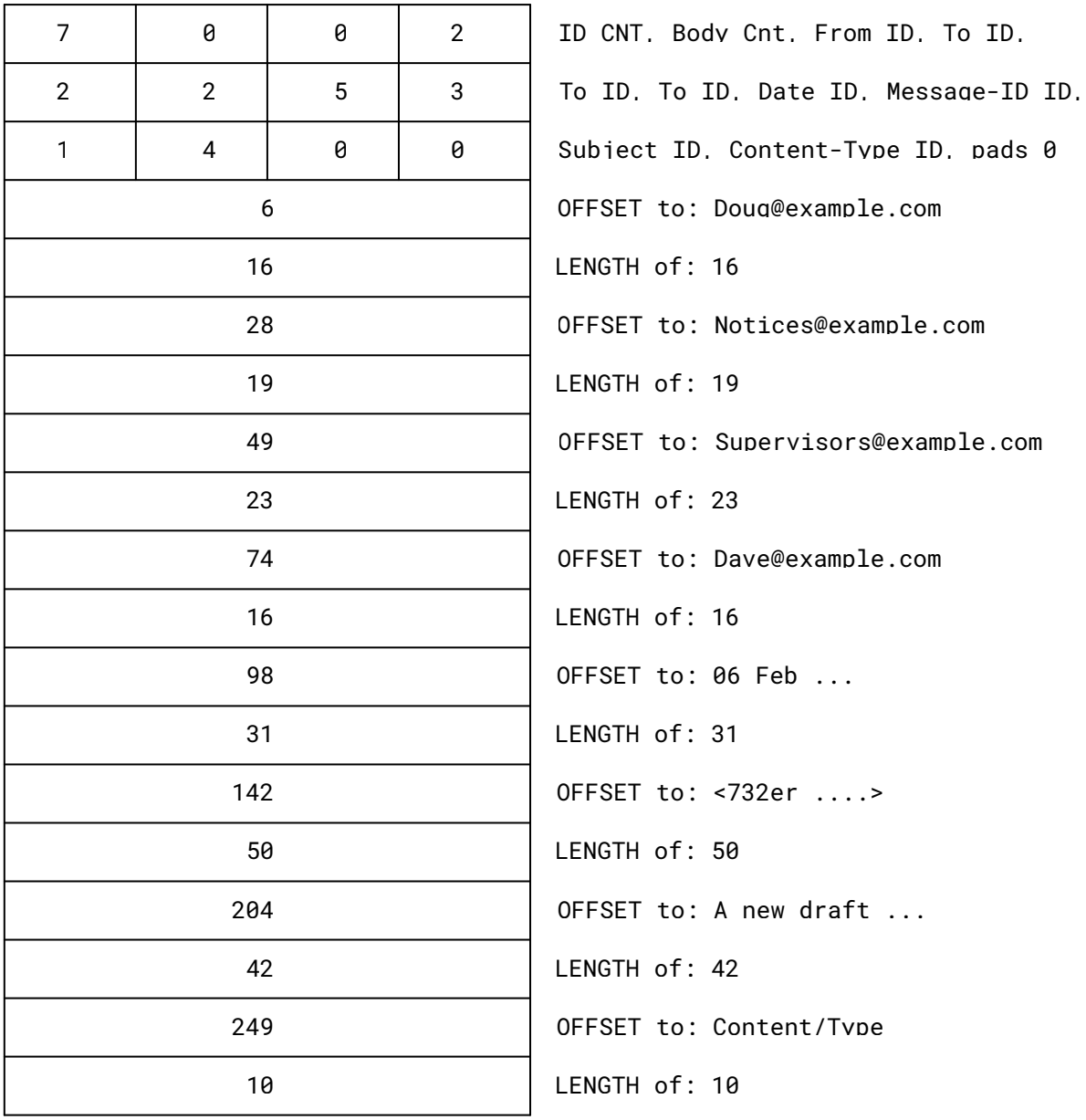


Figure 11: Header Index

7.3.5. Header Index Example 2

For example, if the client requested MIME object indexes for the "From", "Subject", "To", "MIME-Version", and "Content-Type". header values.

And when the folder was opened, the client asked for the "Content-Type" header.

Assume this is a MIME message with two body parts. So the body part header index has a count of two (2). And the HID values assigned by the client when opening the folder are:

- From: 8
- Subject: 12
- To: 4
- Content-Type: 3
- MIME-Version: 9

In the Message each line is terminated with a carriage return and line feed:

```
From: User@example.com
To: User2@example.com
Subject: This is the subject of a sample message
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="XXXXboundary text"

--XXXXboundary text
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: quoted-printable

This is the body text of a sample message

--XXXXboundary text
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: quoted-printable

This is the body text of a sample message.
--XXXXboundary text--
```

5	2	8	4	ID CNT. Body Cnt. From ID. To ID.
12	9	3	0	Subject ID. MIME ID. Content-Type ID
6				OFFSET to: User@example.com
16				LENGTH of: 16
28				OFFSET to: User2@example.com
17				LENGTH of: 17
56				OFFSET to: This is the subject ...
39				LENGTH of: 39
111				OFFSET to: 1.0
3				LENGTH of: 3
130				OFFSET to: multiplar/alternative...
50				LENGTH of: 50

Next is the data for the first body part.
This one body part has no body parts, so its Body Cnt is zero.

206				Offset to start of Body Part
1	0	3	0	ID CNT. Body Cnt. Content-Type ID. pad
220				OFFSET to: Content/Type
27				LENGTH of: 27

Then the second body part:

361				Offset to start of Body Part
1	0	3	0	ID CNT. Body Cnt. Content-Type ID. pad
376				OFFSET to: Content/Type
26				LENGTH of: 26

Figure 12: Header And Body Part Index

8. IANA Considerations

This memo includes no request to IANA. [CHECK]

9. Security Considerations

This document should not affect the security of the Internet. [CHECK]

10. References

10.1. Normative References

[RFC0822] Crocker, D., "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES", STD 11, RFC 822, DOI 10.17487/RFC0822, August 1982, <https://www.rfc-editor.org/info/rfc822>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9051] Melnikov, A., Ed. and B. Leiba, Ed., "Internet Message Access Protocol (IMAP) - Version 4rev2", RFC 9051, DOI 10.17487/RFC9051, August 2021, <https://www.rfc-editor.org/info/rfc9051>.

10.2. Informative References

[exampleRefMin] Surname [REPLACE], Initials [REPLACE]., "Title [REPLACE]", 2006.

[exampleRefOrg] Organization [REPLACE], "Title [REPLACE]", 1984, <http://www.example.com/>.

Appendix A. Administrative Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	USER_CERT	Manage a users certificate.
x	USER_CREATE	When sent in a capability reply USER_CREATE informs the user that they have permission to create users. When sent as a command the USER_CREATE instructs the other endpoint to create a named user.
x	USER_DELETE	Delete a user.
x	USER_LIST	List users and their capabilities.
x	USER_PERMISSIONS	Update user permissions.
x	USER_RENAME	Rename a user.
x	USER_RESET	Used to coordinate resetting a users authentication information.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 3

Appendix B. Authentication Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	AUTH_TODO	xxx.
xxx	AUTH_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 4

Appendix C. File and Folder Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	FILE_TODO	xxx.

Decimal Value	Command / Capability Name	Brief Description.
xxx	FILE_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 5

Appendix D. Protocol Enumerated Binary Values

Phoenix is a binary protocol. Each value is sent as an unsigned 32-bit integer in xdr format.

The values for the commands are arbitrary and were assigned as created. There is no plan or origination to the numbers. There is no priority or superiority to any value. The table is sorted by name, not value.

The values are not unique. They are only unique within the context in which they are used.

Some of these values are reused for other commands. For example USER_CREATE is both an (a) AUTH capability reply informing the user that they have permission to create a user with the (b) USER_CREATE command.

Some values may be reused if they are parameter arguments to other commands. For example xxxxxx.

Decimal Value	Command / Capability Name	Brief Description.
x	PROTO_TODO	xxx.
xxx	PROTO_xxx	xxx.
4294967296	Reserved for future expansion.	4294967296 has a hex value of: 0xffffffff

Table 6

Appendix E. RPCGEN protocol specification

The following is the extendable RPCGEN specification for the Phoenix protocol defined in this document.

E.1. RPCGEN - Acl

```
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * The Acl_Cmd ...
% */
#endif
class Acl
{
        int Todo;
};
```

E.2. RPCGEN - Administration

```
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * The Administratoin Commands
% */
#endif
enum AdministrativeCommands_e
{
        USER_CREATE = 0,
        USER_DELETE = 1,
        USER_RENAME = 2,
        USER_LIST = 3,
        USER_PERMISSIONS = 4,
        SERVER_SHUTDOWN = 5,
        SERVER_LOGS = 6,
        SERVER_KICK_USER = 7,
        SERVER_MANAGE_BANS = 8,
        SERVER_VIEW_STATS = 9
};

#ifdef RPC_HDR
%/**
% * The Administratoin Command ...
% */
#endif
class Administration
{
        int Todo;
};

#ifdef RPC_HDR
%/**
% * The Administratoin Reply Command ...
% */
#endif
class AdministrationReply
{
        int Todo;
};
```

E.3. RPCGEN - Authenticate

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
```

```
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * Authentication is initiated by the endpoing doing the inital
% * connection, to the endpoint it is connecting to.
% *
% * The start of the authentication process take one of two
% * directions:
% *
% * (1) New account, new client, or new authentication mechanism being
% * attempted.
% *
% * (2) - Existing account from a previously used client using a
% * previously known to work (from the clients point of view) authentication
% * mechanism.
% *
% * New is divided up into (1.a) unknown or new account or (1.b) known
account.
% *
% * (1.a) New, known account:
% * Just after the network connection is made, the client sends its
% * pre authentication capabilities to the server, then waits
% * for the pre authentication capabilities packet to arrive from
% * the server. The server supplied pre authentication packet includes
% * the authentication mechanisms it suppots.
% *
% * (1.b) New, unknown account:
% * Starts off like (a), then checks the servers capability set for
% * "allow-new-accounts". If provided, then the sign-up procedure is
% * followed. (sign-up documentation below). Followed by an authentication
% * to verify it worked.
% *
% * If "allow-new-accounts" is not provided, then the account information
% * must be aquired by methods external to this protocol.
% *
% * @note
% * Allowing new users to sign up using this protocol can be site
% * specific and may include procedures external to this protocol
% * such as visiting web sites or other external verification processes.
% *
% * @note
% * For security and anti-junk user accounts, many site may choose not enable
% * "allow-new-accounts". For servers internal to orginazations or on secure
% * networks this might be enabled.
% *
% * (2) Existing or known:
% * Just after the network connection is made, the client sends its
% * pre authentication capabilities to the server. Then without
% * waiting sends any first step authentication data to the server.
% * These are two separate packets each unique and independent.
% *
% * In both cases:
% * Authentication proceeds.
% */
#endif
struct AuthMD5
```

```
{
        string Account<>;
        string Md5Password<>;
};

struct Authenticate
{
        AuthMD5 Md5;
};

struct AuthenticateReply
{
        bool_t  Accepted;
};
```

E.4. RPCGEN - Capability

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * The Capability command informs the other endpoint about
% * its capabilities.
% *
% * This is done once at connection time.
% * And once after authentication is successful.
% * All other attempts will get a NotSupported_Cmd reply.
% *
% * A single capability is a string key, and a string value.
% * The value for each capability is described in its own
% * section.
% *
% * A capability value may contain one or more values, each separated
% * as defined in the capability description.
% * It is suggested that a comma (HEX 2C) be used. Except when
% * that would complicate the capability value.
% * In all cases, the capability value is defined separately for
% * each capability.
% *
% * If the capability list does not include the capability name,
% * then it is not supported.
% *
% * If the reply contains the capability name, then
% * its associated value will be used to determine the
% * extent of its support.
% *
% * Capability keys MUST BE processed in all lower case.
% * If a capability arrives in upper or mixed case,
% * the receiver MUST covert then check to see if that is
% * valid. This prevents capabilities with the same name
```

```
% * and varying case from being used.
% *
% * Capability values SHOULD BE in all lower case.
% * Except when the capability itself contains values
% * that must be upper or mixed case.
% *
% * Capabilties that have a boolean value, SHOULD use
% * 'true' or 'false' and not 'yes', 'no' or other variations of true or
false.
% *
% * Capabilties that have a enabled or disabled value, SHOULD use
% * 'enabled' or 'disabled' and not 'yes', 'no' or other variations of
% * enabled or disabled.
% *
% */
#endif

class Capability
{
        ArrayOfStrings  Supported<>;
};

#ifdef RPC_HDR
%/**
% * Capability FolderCapabilities_e Capabilities that
% * apply to folder or directories.
% *
% * Folder capabilites should not be provided in pre-authentication
% * capability packets.
% *
% * @note
% * Not call capabilities are applicable to all endpoints, files,
% * folders, users, or specific commands. Read the associated
% * command and capability specifications to understand each
% * operation.
% *
% * The capability specifies the endpoing is capable of the
% * function. It may be further restriced by who is authenticated
% * or acess control lists (ACLs) on specific associated items.
% *
% * <ul>
% * <li>CanList: Has a key of "canlist" and has a boolean value.
% * -When true, then a list of files and folders can be accessed.
% * -When false, then only already known named files and folders can be
accessed.
% * </li>
% *
% * <li>CanSubscribe: Has a key value of "cansubscribe" and has
% * a boolean value.
% * -When true, then this server supports unsolicited push notifcations.
% * -When false, then no unsolicited push notifcations can not be sent
% * from this endpoint.
% * </li>
% *
% * <li>CanCreate: Has a key value of "cancreate" and has a boolean value.
% * -When true, then this endpoint supports creating folders.
% * -When false, this endpoint does not support creating folders.
% * <li>
```

```
% *
% * <li>CanRemove: Has a key value of "canremove" and has a boolean value.
% * -When true, this endpoint supports removing folders.
% * -When fales, this endpoint does not support removing foldres.
% * <li>
% *
% * <li>CanRename: Has a key value of "canrename" and has a boolean value.
% * When true, this endpoint supports renaming folders.
% * When false, this endpoint does not support renaming folders.
% * <li>
% *
% * <li>CanCopy: Has a key value of "cancopy" and has a boolean value.
% * When true, this endpoint supports copying folders.
% * When false, this endpoint does not support copying folders.
% * <li>
% *
% * <li>CanSearch: Has a key value of "cansearch" and has a boolean value.
% * When true, this endpoint supports searching folders names using posix
regex expressions
% * When false, this endpoint does not support searching folders.
% * <li>
% *
% * <li>Acls: Has a key value of 'acls' and has a string value.
% * When supplied, the endpoint supports at least one acl type.
% * The types currely defined in the Acls_e enumeration.
% */
#endif

enum FolderCapabilities_e
{
        CanList,
        CanSubscribe,
        CanCreate,
        CanRemove,
        CanAppend,
        CanRename,
        CanUpdate,
        CanCopy,
        CanSearch,
        Acls
};

enum Acls_e
{
        OwnerRW_t,
        OwnerRO_t,
        GroupRW_t,
        GroupRO_t,
        OtherRW_t,
        OtherRO_t,
        NamedListRW_t,
        NamedListRO_t
};
```

E.5. RPCGEN - Folder

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#include "MetaData.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#include <RiverExplorer/Phoenix/MetaData.hpp>
%#endif

#ifdef RPC_HDR
%/**
% */
#endif

enum Folder_e
{
        FolderCreate_Cmd = 0,
        FolderCreateReply_Cmd = 1,
        FolderCopy_Cmd = 2,
        FolderCopyReply_Cmd = 3,
        FolderDelete_Cmd = 4,
        FolderDeleteReply_Cmd = 5,
        FolderRename_Cmd = 6,
        FolderRenameReply_Cmd = 7,
        FolderMove_Cmd = 8,
        FolderMoveReply_Cmd = 9,
        FolderShare_Cmd = 10,
        FolderShareReply_Cmd = 11,
        FolderList_Cmd = 12,
        FolderListReply_Cmd = 13,
        FileCreate_Cmd = 14,
        FileCreateReply_Cmd = 15,
        FileCopy_Cmd = 16,
        FileCopyReply_Cmd = 17,
        FileDelete_Cmd = 18,
        FileDeleteReply_Cmd = 19,
        FileRename_Cmd = 20,
        FileRenameReply_Cmd = 21,
        FileMove_Cmd = 22,
        FileMoveReply_Cmd = 23,
        FileShare_Cmd = 24,
        FileShareReply_Cmd = 25,
        FileGet_Cmd = 26,
        FileGetReply_Cmd = 27
};

#ifdef RPC_HDR
%/**
% * @class FolderCreate CmdFolder.hpp <RiverExplorer/Phoenix/CmdFolder.hpp>
% * @addtogroup Folder
% *
% * The full path is the path from the top of the users virtual
% * home folder, and includes the folder name.
% *
```

```
% * The path separator is the '/' (UTF-8 value 0x2f) character.
% *
% * The root folder is "/".
% */
#endif
class FolderCreate
{
        string  FullPath<>;
};

#ifdef RPC_HDR
%/**
% * @class FolderCreateReply CmdFolder.hpp <RiverExplorer/Phoenix/
CmdFolder.hpp>
% * @addtogroup Folder
% * The reply for CreateFolder, Success is set to true if the
% * folder was created. Otherwise it is set to false.
% */
#endif
class FolderCreateReply
{
        bool_t  Success;
};

#ifdef RPC_HDR
%/**
% * @class FolderCopyr CmdFolder.hpp <RiverExplorer/Phoenix/CmdFolder.hpp>
% * @addtogroup Folder
% *
% * The full original path is the path from the top of the users virtual
% * home folder, and includes the folder name. And names
% * the folder to copy from.
% *
% * The full destination path is the path from the top of the users virtual
% * home folder, and includes the folder name.
% * And names the folder that will have a copy of FullOriginalPath
% * placed into.
% *
% * When recursive is false, only the contents are copied
% * and not any directories.
% *
% * When recursive is true, the contents are copied
% * and recursively copies all directories in the original path directories
% * to the new destination.
% *
% * The path separator is the '/' (UTF-8 value 0x2f) character.
% *
% * The root folder is "/".
% */
#endif
class FolderCopy
{
        string  FullOriginalPath<>;
        string  FullDestinationPath<>;
        bool_t  Recursive;
};

#ifdef RPC_HDR
```

```
%/**
% * @class FolderCopyReply CmdFolder.hpp <RiverExplorer/Phoenix/
CmdFolder.hpp>
% * @addtogroup Folder
% *
% * The reply for CopyFolder, Success is set to true if the
% * folder was copied into FillToPath. Otherwise it is set to false.
% *
% * A false indicates that nothing was copied.
% * A true indicates that everything was copied.
% *
% * Must fail and leave the original structure in place
% * on any failure, and return false.
% */
#endif
class FolderCopyReply
{
        bool_t  Success;
};

class FolderDelete
{
        int todo;
};

class FolderDeleteReply
{
        int todo;
};

class FolderRename
{
        int todo;
};

class FolderRenameReply
{
        int todo;
};

class FolderMove
{
        int todo;
};

class FolderMoveReply
{
        int todo;
};

class FolderShare
{
        int todo;
};

class FolderShareReply
{
        int todo;
```

```
};

class FolderList
{
        int todo;
};

class FolderListReply
{
        int todo;
};

class FileCreate
{
        int todo;
};

class FileCreateReply
{
        int todo;
};

class FileCopy
{
        int todo;
};

class FileCopyReply
{
        int todo;
};

class FileDelete
{
        int todo;
};

class FileDeleteReply
{
        int todo;
};

class FileRename
{
        int todo;
};

class FileRenameReply
{
        int todo;
};

class FileMove
{
        int todo;
};

class FileMoveReply
```

```
{
        int todo;
};

class FileShare
{
        int todo;
};

class FileShareReply
{
        int todo;
};

class FileGet
{
        int todo;
};

class FileGetReply
{
        int todo;
};

union FolderCmdData switch (Folder_e FCmd)
{
 case FolderCreate_Cmd:
         FolderCreate * FolderCreateFolder;

 case FolderCopy_Cmd:
         FolderCopy * FolderCopyData;

 case FolderDelete_Cmd:
         FolderDelete * FolderDeleteData;

 case FolderRename_Cmd:
         FolderRename * FolderRenameData;

 case FolderMove_Cmd:
         FolderMove * FolderMoveData;

 case FolderShare_Cmd:
         FolderShare * FolderShareData;

 case FolderList_Cmd:
         FolderList * FolderListData;

 case FileCreate_Cmd:
         FileCreate * FileCreateData;

 case FileCopy_Cmd:
         FileCopy * FileCopyData;

 case FileDelete_Cmd:
         FileDelete * FileDeleteData;

 case FileRename_Cmd:
         FileRename * FileRenameData;
```

```
 case FileMove_Cmd:
         FileMove * FileMoveData;

 case FileShare_Cmd:
         FileShare * FileShareData;

 case FileGet_Cmd:
         FileGet * FileGetData;

};

union FolderReplyData switch (Folder_e FCmd)
{
 case FolderCreateReply_Cmd:
         FolderCreateReply * FolderCreateReplyData;

 case FolderCopyReply_Cmd:
         FolderCopyReply * FolderCopyReplyData;

 case FolderDeleteReply_Cmd:
         FolderDeleteReply * FolderDeleteReplyData;

 case FolderRenameReply_Cmd:
         FolderRenameReply * FolderRenameReplyData;

 case FolderMoveReply_Cmd:
         FolderMoveReply * FolderMoveReplyData;

 case FolderShareReply_Cmd:
         FolderShareReply * FolderShareReplyData;

 case FolderListReply_Cmd:
         FolderListReply * FolderListReplyData;

 case FileCreateReply_Cmd:
         FileCreateReply * FileCreateReplyData;

 case FileCopyReply_Cmd:
         FileCopyReply * FileCopyReplyData;

 case FileDeleteReply_Cmd:
         FileDeleteReply * FileDeleteReplyData;

 case FileRenameReply_Cmd:
         FileRenameReply * FileRenameReplyData;

 case FileMoveReply_Cmd:
         FileMoveReply * FileMoveReplyData;

 case FileShareReply_Cmd:
         FileShareReply * FileShareReplyData;

 case FileGetReply_Cmd:
         FileGetReply * FileGetReplyData;

};
```

```
#ifdef RPC_HDR
%/**
% * An XML representation of a file, not the
% * contents, but the information about the file.
% *
% * - Meta: Is an array of MetaData.
% * - Name: This is the name excluding the path.
% * - Size: This is the full size of the file.
% * The FLAG_... symbols are symbolic names
% */
#endif

class FileInformation
{
        MetaData        Meta<>;
        string          Name<>;
        uint32_t        Size;
        uint32_t        FlagBits;
};

#ifdef RPC_HDR
%/**
% * An XML representation of a folder
% *
% * - Meta: Is an array of MetaData.
% * - Name: This is the name excluding the path.
% * - Folders: An array of information about folders within this folder.
% * - Files: An array of information about files in this folder.
% */
#endif
class FolderInformation
{
        MetaData                Meta<>;
        string                  Name<>;
        FolderInformation       Folders<>;
        FileInformation         Files<>;
};

class Folder
{
        FolderCmdData Data;
};

class FolderReply
{
        FolderReplyData Data;
};
```

E.6. RPCGEN - KeepAlive

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * The KeepAlive command sends a packet to the remote endpoint.
% *
% * There is no reply to a KeepAlive command.
% */
#endif
struct KeepAlive
{
        int foo;
};
```

E.7. RPCGEN - NotSupported

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * Not supported. This is sent back to the initiating endpoint
% * when this endpoint does not support the command sent.
% *
% * @note
% * There is no data associated with a NotSupported_Cmd, only the
% * CmdPacket is sent.
% */
#endif

struct NotSupported
{
        int foo;
};

struct NotSupportedReply
{
        int foo;
};
```

E.8. RPCGEN - Ping

```
%
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#endif

#ifdef RPC_HDR
%/**
% * The Ping command sends a packet to the remote endpoint.
% * The other endpoint does a PingReply with no data.
% *
% * The reply is required.
% */
#endif
class Ping
{
        int foo;
};

#ifdef RPC_HDR
%/**
% * The Ping Reply command sends a packet to the remote endpoint.
% * The other endpoint does a PingReply with no data.
% *
% * The reply is required.
% */
#endif
class PingReply
{
        int foo;
};

#ifdef RPC_HDR
%/**
% * @return a new Ping CmdPacket.
% */
#endif
%namespace RiverExplorer::Phoenix
%{
%class CmdPacket;
%extern CmdPacket * NewPing(CommandSequence Seq);
%}
```

E.9. RPCGEN - Commands

```
%#ifdef BUILDING_LIBPHOENIX
%#include "CppTypes.hpp"
%#else
%#include <RiverExplorer/Phoenix/CppTypes.hpp>
```

```
%#endif

#ifdef RPC_HDR
%/**
% * Command_e: An enumerated list of fetch commands.
% * <ul>
% * <li>
% *     Admin_Cmd - The packet contains an administrative command.
% * </li>
% * <li>
% *     AdminReply_Cmd - The packet contains an administrative command reply.
% * </li>
% * <li>
% *     Auth_Cmd - The packet contains an authentication command.
% * </li>
% * <li>
% *     AuthReply_Cmd - The packet contains an authentication command reply.
% * </li>
% * <li>
% *     Capability_Cmd - The packet contains a capability command.
% * A Capability_Cmd has no reply.
% * </li>
% * <li>
% *     Folder_Cmd - The packet contains a folder command.
% * </li>
% * <li>
% *     FolderReply_Cmd - The packet contains a folder command reply.
% * </li>
% * <li>
% * The NotSupported_Cmd is sent back to
% * the remote endpoint when it sends a command
% * that is not supported.
% * It can be because of access control list,
% * out of sequence, or other error.
% * A Capability_Cmd has no reply.
% * </li>
% * <li>
% *     Ping_Cmd - The packet contains a pint command reply.
% * The reply to a ping is the same packet back.
% * </li>
% * <li>
% *     Proto_Cmd - The packet contains a protocol command.
% * A protocol command is an extension command not built int
% * the core Phoenix protocol.
% * </li>
% * <li>
% *     ProtoReply_Cmd - The packet contains a protocol command reply.
% * </li>
% * <li>
% *     Reserved_Cmd - In the unlikely event that 2^32 commands
% * are ever created, this is an escape to allow more.
% * </li>
% * </ul>
% *
% */
#endif
enum Command_e
{
```

```
        Admin_Cmd = 1,
        AdminReply_Cmd = 2,
        Auth_Cmd = 3,
        AuthReply_Cmd = 4,
        Capability_Cmd = 5,
        Folder_Cmd = 6,
        FolderReply_Cmd = 7,
        NotSupported_Cmd = 8,
        Ping_Cmd = 9,
        Proto_Cmd = 10,
        ProtoReply_Cmd = 11,
        Reserved_Cmd = 0xffffffff
};

#ifdef RPC_HDR
%/**
% * The transport top level is simple.
% * You can Send() a packet and get a packet back.
% * Or you send a notification that receives nothing back.
% * Or you send a broadcast message to all interested participants, with no
% * reply expected.
% */
#endif

#ifdef RPC_HDR

%#ifdef BUILDING_LIBPHOENIX
%#include "CmdAcl.hpp"
%#include "CmdAddMessage.hpp"
%#include "CmdAuthenticate.hpp"
%#include "CmdAdministration.hpp"
%#include "CmdCapability.hpp"
%#include "CmdCopyMessage.hpp"
%#include "CmdFolder.hpp"
%#include "CmdExpunge.hpp"
%#include "CmdGetMessage.hpp"
%#include "CmdKeepAlive.hpp"
%#include "CmdNotSupported.hpp"
%#include "CmdPing.hpp"
%#include "CmdSearch.hpp"
%#include "CmdSubscribe.hpp"
%#include "CmdTimeout.hpp"
%#include "CmdUpdateMessage.hpp"
%#include "Commands.hpp"
%#else
%#include <RiverExplorer/Phoenix/CmdAcl.hpp>
%#include <RiverExplorer/Phoenix/CmdAddMessage.hpp>
%#include <RiverExplorer/Phoenix/CmdAuthenticate.hpp>
%#include <RiverExplorer/Phoenix/CmdAdministration.hpp>
%#include <RiverExplorer/Phoenix/CmdCapability.hpp>
%#include <RiverExplorer/Phoenix/CmdCopyMessage.hpp>
%#include <RiverExplorer/Phoenix/CmdFolder.hpp>
%#include <RiverExplorer/Phoenix/CmdExpunge.hpp>
%#include <RiverExplorer/Phoenix/CmdGetMessage.hpp>
%#include <RiverExplorer/Phoenix/CmdKeepAlive.hpp>
%#include <RiverExplorer/Phoenix/CmdNotSupported.hpp>
%#include <RiverExplorer/Phoenix/CmdPing.hpp>
%#include <RiverExplorer/Phoenix/CmdSearch.hpp>
```

```
%#include <RiverExplorer/Phoenix/CmdSubscribe.hpp>
%#include <RiverExplorer/Phoenix/CmdTimeout.hpp>
%#include <RiverExplorer/Phoenix/CmdUpdateMessage.hpp>
%#include <RiverExplorer/Phoenix/Commands.hpp>
%#endif
#endif

union CmdData switch (Command_e Cmd)
{
 case Admin_Cmd:
        Administration *       AdminData;

 case AdminReply_Cmd:
        AdministrationReply *  AdminReplyData;

 case Auth_Cmd:
        Authenticate           *       AuthData;

 case AuthReply_Cmd:
        AuthenticateReply              *       AuthReplyData;

 case Capability_Cmd:
        Capability             *       CapabilityData;

 case Folder_Cmd:
        Folder *       FolderData;

 case FolderReply_Cmd:
        FolderReply    *       FolderReplyData;

 case NotSupported_Cmd:
        void;

 case Ping_Cmd:
        void;

};

#ifdef RPC_HDR
%/**
% * Set the callback for a command.
% *
% * @param Cmd The command being registered.
% *
% * @param Callback The user supplied callback function.
% *
% * @note
% * It is is recommended that each registered callback be thread safe.
% */
%namespace RiverExplorer::Phoenix
%{
%extern bool Register_Cmd(Command_e Cmd, CommandCallback * Callback);
%}
#endif

#ifdef RPC_HDR
%/**
% * Set the callback for a command.
```

```
% *
% * @param Cmd The command being registered.
% *
% * @param Callback The user supplied callback function.
% *
% * @note
% * It is is recommended that each registered callback be thread safe.
% */
%namespace RiverExplorer::Phoenix
%{
%extern bool Register_Cmd(Command_e Cmd, CommandCallback * Callback);
%}
#endif

#ifdef RPC_HDR
%/**
% * A command consists of:
% * - The enumerated command (Command_e),
% * - An XDR opaque object (length + data).
% * The XDR opaque data was prepared by the Cmd specific code
% * and is set into place for transport.
% */
#endif
struct  CmdPacket
{
        CommandSequence Sequence;
  CmdData Data;
};
```

E.10. RPCGEN - EMail

```
const FLAG_SEEN                         = 0x0001;
const FLAG_ANSWERED             = 0x0002;
const FLAG_FLAGGED              = 0x0004;
const FLAG_DELETED              = 0x0008;
const FLAG_DRAFT                        = 0x0010;
const FLAG_FORWARDED    = 0x0040;
const FLAG_MDNSENT              = 0x0080;
const FLAG_JUNK                         = 0x0100;
const FLAG_NOTJUNK              = 0x0200;
const FLAG_PHISHING             = 0x0400;

#ifdef RPC_HDR
%/**
% * @note
% * The "C" routines return a bool_t, the C++ API return a bool.
% */
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#include "MetaData.hpp"
%#include "Mime.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#include <RiverExplorer/Phoenix/MetaData.hpp>
%#include <RiverExplorer/Phoenix/Mime.hpp>
```

```
%#endif
%#include <string>
%#include <vector>

#endif

#ifdef RPC_HDR
%/**
% * EMail headers are a vector of MetaData.
% *
% * Some examples:
% * @verbatim
% *
% * From: RiverExplorer.USexample.com
% * To: CEO@example.com
% * Subject: The holidays!
% *
% * @endverbatim
% *
% * @note
% * This does not define new or alter existing header types.
% * This is a container to transport them.
% */
#endif
typedef MetaData EMailHeader;
typedef EMailHeader EMailHeaders<>;

#ifdef RPC_HDR
%
%/**
% * An email is a vector of MimeBodyPart
% *
% * @note
% * This does not define new or alter existing mime or email headers.
% * This is a container to transport them.
% */
#endif
typedef MimeBodyPart EMailBodyParts<>;

#ifdef RPC_HDR
%
%/**
% * Check for the existence of a specific Header.
% *
% * @param Headers The Headers that is being processed.
% *
% * @param Key The header name being looked for.
% *
% * @return The number of matches.
% * Returns zero (0) when none are found.
% * Returns ((uint32_t)-1) when Obj or Key are NULL.
% */
%namespace RiverExplorer::Phoenix{
%extern uint32_t * EMail_HasHeader(EMailHeaders * Headers, const char * Key);
%}
#endif

#ifdef RPC_HDR
```

```
%
%/**
% * Get a specific Header.
% *
% * @param Headers The Headers that is being processed.
% *
% * @param Key The header name being looked for.
% *
% * @return A vector of the matches.
% * Will return zero (0) or more matches, in the original order.
% * Will return NULL of Obj or Key are NULL.
% * Will return NULL when Key is not found.
% */
%namespace RiverExplorer::Phoenix{
%extern EMailHeader  * EMail_GetHeader(EMailHeaders * Headers, const char *
Key);
%}
#endif

#ifdef RPC_HDR
%
%/**
% * Headers count.
% *
% * @param Headers The Headers that is being processed.
% *
% * @return The number of EMailHeader in Headers.
% * Returns ((uint32_t)-1) when Headers is NULL.
% */
%namespace RiverExplorer::Phoenix{
%extern uint32_t EMail_GetHeaderCount(EMailHeaders * Headers);
%}
#endif

#ifdef RPC_HDR
%
%/**
% * Headers iterator.
% *
% * @param Headers The Headers that is being processed.
% *
% * @param Which The header to get, the first is zero (0).
% *
% * @return A pointer to the EMailHeader.
% * Returns NULL when Headers or Key is NULL.
% * Returns NULL when Which is not a valid index.
% */
%namespace RiverExplorer::Phoenix{
%extern EMailHeader * EMail_GetHeaderByIndex(EMailHeaders * Headers, uint32_t
Which);
%}
#endif

#ifdef RPC_HDR
%
%/**
% * Common headers names.
% */
```

```
%namespace RiverExplorer::Phoenix{
%extern const char * Date_s;                    /** "Date" */
%extern const char * From_s;                    /** "From" */
%extern const char * Subject_s;         /** "Subject" */
%extern const char * To_s;                              /** "To" */
%}
#endif

#ifdef RPC_HDR
#ifdef __cplusplus
} // End extern "C"
%
#endif
%#ifdef __cplusplus
%namespace RiverExplorer::Phoenix::EMail {
%class Message
%{
% public:
%
%       /**
%        * Message - Default Destructor.
%        */
%       Message();
%
%       /**
%  * Message - Destructor.
%  */
%
%       /**
%  * Get the headers.
%        *
%        * @return All of the headers.
%  */
%       EMailHeaders * Headers() const;
%
% /**
%  * Add a header.
%  *
%  * @param NewHeader The new header to add.
%  */
%  void Add(EMailHeader & NewHeader);
%
% /**
%  * Add a header.
%  *
%  * @param Key The new header to add.
%  *
%  * @param Value The new header value to add.
%  */
%  void Add(std::string Key, std::string Value);
%
% /**
%  * Get the named header.
%  *
%  * @param Key The new header to add.
%  *
%        * @return A vector of all the matches. It will contain
%        * zero or more entries.
```

```
%  *
%  * @note
%  * You MUST delete result when finished or you will have
%        * a memory leak. You MUST NOT delete the
%        * entries in the vector, they are still in the email message.
%  */
%  std::vector<const EMailHeader*>      * Header(std::string Key);
%
% /**
%  * Get the number of body parts.
%  *
%  * @return The number of body parts.
%        */
% const uint32_t Count() const;
%
% /**
%  * Get the body parts.
%  *
%  * @return The body parts.
%  *
%  * @note
%  * Do not delete the results.
%        */
% const MimeBodyPart * Body() const;
%
% /**
%  * Get the nTh body part.
%  *
%  * @param nTh Which to get, first is zero (0).
%  *
%  * @return The body part.
%  * Returns nullptr when nTh does not exist.
%  *
%  * @note
%  * Do not delete the results.
%        */
% const MimeBodyPart * Body(uint32_t nTh) const;
%
% /**
%        * This ID is used as a transport ID between
%        * endpoints to uniquely identify this message for this
%        * account on this server.
%        *
%        * @return The message ID For this message.
%        * A return value of zero (0) indicates that this message
%  * does not have an ID.
%        *
%        * @note
%        * This is NOT the 'Message-ID' in email headers.
%        * This is the ID that uniquely identifies this message to endpoints.
%        *
%        * @note
%        * This ID is unique to the server or message provider and might
%        * not be unique on the client side.
%        * Implementations may wish to provide their own local ID
%        * to uniquely identify this message in their system that is
%        * separate from this ID.
%        */
```

```
% uint32_t ID() const;
%
% /**
%  * Messages in a message store have an ID.
%  * This sets the message ID for this message.
%        *
%        * @param TransportID The transport ID to associate with this
message.
%        * A value of zero (0) indicates that this message
%  * does not have an ID.
%        *
%        * @note
%        * This is NOT the 'Message-ID' in email headers.
%        * This is the ID that uniquely identifies this message to endpoints.
%        *
%        * @note
%        * This ID is unique to the server or message provider and might
%        * not be unique on the client side.
%        * Implementations may wish to provide their own local ID
%        * to uniquely identify this message in their system that is
%        * separate from this ID.
%        */
% void ID(uint32_t StoreID) const;
%};
%} // End namespace EMail
%#endif // class
#endif // RPC_HDR
```

E.11. RPCGEN - MIME

```
#ifdef RPC_HDR
%/**
% * @note
% * The "C" routines return a bool_t, the C++ API return a bool.
% */
%#ifdef BUILDING_LIBPHOENIX
%#include "Types.hpp"
%#include "MetaData.hpp"
%#else
%#include <RiverExplorer/Phoenix/Types.hpp>
%#include <RiverExplorer/Phoenix/MetaData.hpp>
%#endif
%
%/**
% * A MediaType is a string pair as defined
% * at https://www.iana.org/assignments/media-types/media-types.xhtml
% *
% * Some examples:
% * @verbatim
% *
% * text/html
% * application/pdf
% * audio/ogg
% * image/png
% * multipart/mixed
% *
```

```
% * @endverbatim
% *
% * Would have a Key of 'From' and a value of 'RiverExplorer.US@gmail.com'.
% *
% * @note
% * This does not define new or alter existing IANA media types.
% * This is a containter to transport them.
% */
#endif
typedef MetaData MediaType;

#ifdef RPC_HDR
%/**
% * A MIME header is a key/string value.
% */
#endif
typedef MetaData MimeHeader;

#ifdef RPC_HDR
%/**
% * Headers is an array of MimeHeader.
% *
% * Example:
% * @verbatim
% *
% * From: RiverExplorer.US@example.com
% * To: CEO@example.com
% *
% * @endverbatim
% *
% * @note
% * This does not define new or alter existing mime or email headers.
% * This is a containter to transport them.
% */
#endif
typedef MimeHeader MimeHeaders<>;

#ifdef RPC_HDR
%
%/**
% * A MIME object. Has a MediaType, MimeHeaders and data.
% *
% * To use:
% *
% * @code
% *
% * MimeBodyPart        * TheMimeBodyPart;
% *
% * // ...however you fill in or get the MimeBodyPart object data ...
% * //
% * TheMimeBodyPart = GetTheData();
% *
% * uint32_t            Length;
% * const char* TheMediaType = MimeBodyPart_GetMediaType(TheMimeBodyPart);
% * uint8_t             * Data = MimeBodyPart_GetData(TheMimeBodyPart,
&Length);
% * ...
% *
```

```
% * // At this point, you have the media-type, Data, and length
% * // of the data.
% *
% * @endcode
% */
#endif

#ifdef RPC_HDR
%/**
%       * - Type: The media-type of the object.
% * @note
% * Media-type is not duplicated in Headers.
% *
%       * - Headers: The MIME headers.
% *
% * - Data: The data itself.
% * The data is a binary blob without any encoding.
% */
#endif
struct MimeBodyPart
{
        MediaType               Type;
        MimeHeaders Headers;
        IoVec                           Data;
};
```

E.12. RPCGEN - Phoenix

```
/**
 * These are just used below to make the 'program' values
 * more readable.
 */
#define PhoenixProgramNumber 1
#define PhoenixProgramVersion 1
#define PhoenixSendPacket 1
#define PhoenixNotifyPacket 2
#define PhoenixBroadcastPacket 3

program PhoenixProgram
{
        version PhoenixVersion
        {
                CmdPacket Send(CmdPacket) = PhoenixSendPacket;
                void Notifiy(CmdPacket) = PhoenixNotifyPacket;
                void Broadcast(CmdPacket) = PhoenixBoradcastPacket;

        } = PhoenixProgramVersion;
} = PhoenixProgramNumber;
```

E.13. RPCGEN - Types

```
#ifdef RPC_HDR
/**
 * The orignal RPCGEN was "C" only, RPCGEN++ is "C++".
```

```
 * C useed bool_t, and C++ uses bool.
 * So this is a wrapper.
 *
 * @param xdrs An initialized XDR object.
 *
 * @param BValue The address of the bool_t object.
 *
 * @return false if failed.
 */
%namespace RiverExplorer::Phoenix
%{
%extern bool xdr_bool_t(XDR * xdrs, bool_t * BValue);
%}
#endif

#ifdef RPC_HDR
%/**
% * Each command sent, has a command identifier, and a sequence.
% * Each reply to a command has the same CommandID and sequence.
% */
#endif
typedef uint64_t CommandSequence;

#ifdef RPC_HDR
%/**
% * An array of strings.
% */
#endif
typedef string StringType<>;

#ifdef RPC_HDR
%/**
% * An array of strings.
% */
#endif
typedef StringType ArrayOfStrings<>;


#ifdef RPC_HDR
%/**
% * An object to data objects, some of which might
% * be memory mapped.
% *
% * - IsMMapped: When true, Data was memory mapped and not allocated.
% * - IsAllocated When true, Data was allocated.
% *
% * @note
% * Both IsMMapped and IsAllocated can be false when it is pointing
% * to a subset of an allocated or mmapped data.
% *
% * - Len The number of octets in Data.
% *
% * - Data A pointer to the data.
% */
#endif
class IoVec
{
        bool_t          IsMMapped;
```

```
        bool_t          IsAllocated;
        uint32_t        Length;
        uint8_t * Data;
};
```

Acknowledgments

Contributors

Thanks to all of the contributors. [REPLACE]

Author's Address

Doug Royer
RiverExplorer Games LLC
848 N. Rainbow Blvd #1120
Las Vegas, Nevada 89107
United States of America
Phone: 1+714-989-6135
Email: DouglasRoyer@gmail.com
URI: https://RiverExplorer.games