Network Working Group
Request for Comments: 3119
Category: Standards Track

R. Finlayson
LIVE.COM

June 2001

A More Loss-Tolerant RTP Payload Format for MP3 Audio

Status of this Memo

Copyright Notice

Abstract

This document describes a RTP (Real-Time Protocol) payload format for transporting MPEG (Moving Picture Experts Group) 1 or 2, layer III audio (commonly known as "MP3"). This format is an alternative to that described in RFC 2250, and performs better if there is packet loss.

1. Introduction

While the RTP payload format defined in RFC 2250 [2] is generally applicable to all forms of MPEG audio or video, it is sub-optimal for MPEG 1 or 2, layer III audio (commonly known as "MP3"). The reason for this is that an MP3 frame is not a true "Application Data Unit" - it contains a back-pointer to data in earlier frames, and so cannot be decoded independently of these earlier frames. Because RFC 2250 defines that packet boundaries coincide with frame boundaries, it handles packet loss inefficiently when carrying MP3 data. The loss of an MP3 frame will render some data in previous (or future) frames useless, even if they are received without loss.

In this document we define an alternative RTP payload format for MP3 audio. This format uses a data-preserving rearrangement of the original MPEG frames, so that packet boundaries now coincide with true MP3 "Application Data Units", which can also (optionally) be rearranged in an interleaving pattern. This new format is therefore more data-efficient than RFC 2250 in the face of packet loss.

2. The Structure of MP3 Frames

In this section we give a brief overview of the structure of a MP3 frame. (For more detailed description, see the MPEG 1 audio [3] and MPEG 2 audio [4] specifications.)

Each MPEG audio frame begins with a 4-byte header. Information defined by this header includes:

- Whether the audio is MPEG 1 or MPEG 2.
- Whether the audio is layer I, II, or III.
  (The remainder of this document assumes layer III, i.e., "MP3" frames)
- Whether the audio is mono or stereo.
- Whether or not there is a 2-byte CRC field following the header.
- (indirectly) The size of the frame.

The following structures appear after the header:

- (optionally) A 2-byte CRC field
- A "side info" structure. This has the following length:
  - 32 bytes for MPEG 1 stereo
  - 17 bytes for MPEG 1 mono, or for MPEG 2 stereo
  - 9 bytes for MPEG 2 mono
- Encoded audio data, plus optional ancillary data (filling out the rest of the frame)

For the purpose of this document, the "side info" structure is the most important, because it defines the location and size of the "Application Data Unit" (ADU) that an MP3 decoder will process. In particular, the "side info" structure defines:

- "main_data_begin": This is a back-pointer (in bytes) to the start of the ADU. The back-pointer is counted from the beginning of the frame, and counts only encoded audio data and any ancillary data (i.e., ignoring any header, CRC, or "side info" fields).

An MP3 decoder processes each ADU independently. The ADUs will generally vary in length, but their average length will, of course, be that of the of the MP3 frames (minus the length of the header, CRC, and "side info" fields). (In MPEG literature, this ADU is sometimes referred to as a "bit reservoir".)

3. A New Payload Format

As noted in [5], a payload format should be designed so that packet boundaries coincide with "codec frame boundaries" - i.e., with ADUs. In the RFC 2250 payload format for MPEG audio [2], each RTP packet payload contains MP3 frames. In this new payload format for MP3 audio, however, each RTP packet payload contains "ADU frames", each preceded by an "ADU descriptor".

3.1 ADU frames

An "ADU frame" is defined as:

- The 4-byte MPEG header
  (the same as the original MP3 frame, except that the first 11 bits are (optionally) replaced by an "Interleaving Sequence Number", as described in section 6 below)
- The optional 2-byte CRC field
  (the same as the original MP3 frame)
- The "side info" structure
  (the same as the original MP3 frame)
- The complete sequence of encoded audio data (and any ancillary data) for the ADU (i.e., running from the start of this MP3 frame's "main_data_begin" back-pointer, up to the start of the next MP3 frame's back-pointer)

3.2 ADU descriptors

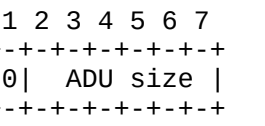
Within each RTP packet payload, each "ADU frame" is preceded by a 1 or 2-byte "ADU descriptor", which gives the size of the ADU, and indicates whether or not this packet's data is a continuation of the previous packet's data. (This occurs only when a single "ADU descriptor"+"ADU frame" is too large to fit within a RTP packet.)

An ADU descriptor consists of the following fields

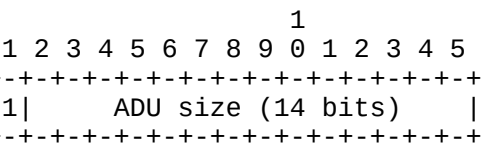
- "C": Continuation flag (1 bit): 1 if the data following the ADU descriptor is a continuation of an ADU frame that was too large to fit within a single RTP packet; 0 otherwise.
- "T": Descriptor Type flag (1 bit):
  0 if this is a 1-byte ADU descriptor;
  1 if this is a 2-byte ADU descriptor.
- "ADU size" (6 or 14 bits):
  The size (in bytes) of the ADU frame that will follow this ADU descriptor (i.e., NOT including the size of the descriptor itself). A 2-byte ADU descriptor (with a 14-bit "ADU size" field) is used for ADU frames sizes of 64 bytes or more. For smaller ADU frame sizes, senders MAY alternatively

use a 1-byte ADU descriptor (with a 6-bit "ADU size" field). Receivers MUST be able to accept an ADU descriptor of either size.

Thus, a 1-byte ADU descriptor is formatted as follows:



and a 2-byte ADU descriptor is formatted as follows:



3.3 Packing rules

Each RTP packet payload begins with a "ADU descriptor", followed by "ADU frame" data. Normally, this "ADU descriptor"+"ADU frame" will fit completely within the RTP packet. In this case, more than one successive "ADU descriptor"+"ADU frame" MAY be packed into a single RTP packet, provided that they all fit completely.

If, however, a single "ADU descriptor"+"ADU frame" is too large to fit within an RTP packet, then the "ADU frame" is split across two or more successive RTP packets. Each such packet begins with an ADU descriptor. The first packet's descriptor has a "C" (continuation) flag of 0; the following packets' descriptors each have a "C" flag of 1. Each descriptor, in this case, has the same "ADU size" value: the size of the entire "ADU frame" (not just the portion that will fit within a single RTP packet). Each such packet (even the last one) contains only one "ADU descriptor".

3.4 RTP header fields

Payload Type: The (static) payload type 14 that was defined for MPEG audio [6] MUST NOT be used. Instead, a different, dynamic payload type MUST be used - i.e., one in the range [96,127].

M bit: This payload format defines no use for this bit. Senders SHOULD set this bit to zero in each outgoing packet.

Timestamp: This is a 32-bit 90 kHz timestamp, representing the presentation time of the first ADU packed within the packet.

3.5 Handling received data

Note that no information is lost by converting a sequence of MP3 frames to a corresponding sequence of "ADU frames", so a receiving RTP implementation can either feed the ADU frames directly to an appropriately modified MP3 decoder, or convert them back into a sequence of MP3 frames, as described in appendix A.2 below.

4. Handling Multiple MPEG Audio Layers

The RTP payload format described here is intended only for MPEG 1 or 2, layer III audio ("MP3"). In contrast, layer I and layer II frames are self-contained, without a back-pointer to earlier frames. However, it is possible (although unusual) for a sequence of audio frames to consist of a mixture of layer III frames and layer I or II frames. When such a sequence is transmitted, only layer III frames are converted to ADUs; layer I or II frames are sent 'as is' (except for the prepending of an "ADU descriptor"). Similarly, the receiver of a sequence of frames - using this payload format - leaves layer I and II frames untouched (after removing the prepended "ADU descriptor), but converts layer III frames from "ADU frames" to regular MP3 frames. (Recall that each frame's layer is identified from its 4-byte MPEG header.)

If you are transmitting a stream consists *only* of layer I or layer II frames (i.e., without any MP3 data), then there is no benefit to using this payload format, *unless* you are using the interleaving mechanism.

5. Frame Packetizing and Depacketizing

The transmission of a sequence of MP3 frames takes the following steps:

```
      MP3 frames
              -1-> ADU frames
                  -2-> interleaved ADU frames
                        -3-> RTP packets
```

Step 1, the conversion of a sequence of MP3 frames to a corresponding sequence of ADU frames, takes place as described in sections 2 and 3.1 above. (Note also the pseudo-code in appendix A.1.)

Step 2 is the reordering of the sequence of ADU frames in an (optional) interleaving pattern, prior to packetization, as described in section 6 below. (Note also the pseudo-code in appendix B.1.) Interleaving helps reduce the effect of packet loss, by distributing consecutive ADU frames over non-consecutive packets. (Note that

because of the back-pointer in MP3 frames, interleaving can be applied - in general - only to ADU frames. Thus, interleaving was not possible for RFC 2250.)

Step 3 is the packetizing of a sequence of (interleaved) ADU frames into RTP packets - as described in section 3.3 above. Each packet's RTP timestamp is the presentation time of the first ADU that is packed within it. Note that, if interleaving was done in step 2, the RTP timestamps on outgoing packets will not necessarily be monotonically nondecreasing.

Similarly, a sequence of received RTP packets is handled as follows:

```
      RTP packets
            -4-> RTP packets ordered by RTP sequence number
                  -5-> interleaved ADU frames
                        -6-> ADU frames
                              -7-> MP3 frames
```

Step 4 is the usual sorting of incoming RTP packets using the RTP sequence number.

Step 5 is the depacketizing of ADU frames from RTP packets - i.e., the reverse of step 3. As part of this process, a receiver uses the "C" (continuation) flag in the ADU descriptor to notice when an ADU frame is split over more than one packet (and to discard the ADU frame entirely if one of these packets is lost).

Step 6 is the rearranging of the sequence of ADU frames back to its original order (except for ADU frames missing due to packet loss), as described in section 6 below. (Note also the pseudo-code in appendix B.2.)

Step 7 is the conversion of the sequence of ADU frames into a corresponding sequence of MP3 frames - i.e., the reverse of step 1. (Note also the pseudo-code in appendix A.2.) With an appropriately modified MP3 decoder, an implementation may omit this step; instead, it could feed ADU frames directly to the (modified) MP3 decoder.

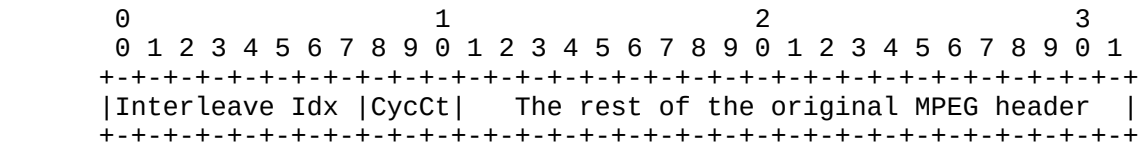
6. ADU Frame Interleaving

In MPEG audio frames (MPEG 1 or 2; all layers) the high-order 11 bits of the 4-byte MPEG header ('syncword') are always all-one (i.e., 0xFFE). When reordering a sequence of ADU frames for transmission, we reuse these 11 bits as an "Interleaving Sequence Number" (ISN). (Upon reception, they are replaced with 0xFFE once again.)

The structure of the ISN is (a,b), where:

- a == bits 0-7: 8-bit Interleave Index (within Cycle)
- b == bits 8-10: 3-bit Interleave Cycle Count

I.e., the 4-byte MPEG header is reused as follows:



Example: Consider the following interleave cycle (of size 8):
1,3,5,7,0,2,4,6
(This particular pattern has the property that any loss of up to four consecutive ADUs in the interleaved stream will lead to a deinterleaved stream with no gaps greater than one [7].) This produces the following sequence of ISNs:

(1,0) (3,0) (5,0) (7,0) (0,0) (2,0) (4,0) (6,0) (1,1) (3,1) (5,1) etc.

So, in this example, a sequence of ADU frames

f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 (etc.)

would get reordered, in step 2, into:

(1,0)f1 (3,0)f3 (5,0)f5 (7,0)f7 (0,0)f0 (2,0)f2 (4,0)f4 (6,0)f6 (1,1)f9 (3,1)f11 (5,1)f13 (etc.)

and the reverse reordering (along with replacement of the 0xFFE) would occur upon reception.

The reason for breaking the ISN into "Interleave Cycle Count" and "Interleave Index" (rather than just treating it as a single 11-bit counter) is to give receivers a way of knowing when an ADU frame should be 'released' to the ADU->MP3 conversion process (step 7 above), rather than waiting for more interleaved ADU frames to arrive. E.g., in the example above, when the receiver sees a frame with ISN (<something>,1), it knows that it can release all previously-seen frames with ISN (<something>,0), even if some other (<something>,0) frames remain missing due to packet loss. A 8-bit Interleave Index allows interleave cycles of size up to 256.

The choice of an interleaving order can be made independently of RTP packetization. Thus, a simple implementation could choose an interleaving order first, reorder the ADU frames accordingly (step 2), then simply pack them sequentially into RTP packets (step 3). However, the size of ADU frames - and thus the number of ADU frames that will fit in each RTP packet - will typically vary in size, so a more optimal implementation would combine steps 2 and 3, by choosing an interleaving order that better reflected the number of ADU frames packed within each RTP packet.

Each receiving implementation of this payload format MUST recognize the ISN and be able to perform deinterleaving of incoming ADU frames (step 6). However, a sending implementation of this payload format MAY choose not to perform interleaving - i.e., by omitting step 2. In this case, the high-order 11 bits in each 4-byte MPEG header would remain at 0xFFE. Receiving implementations would thus see a sequence of identical ISNs (all 0xFFE). They would handle this in the same way as if the Interleave Cycle Count changed with each ADU frame, by simply releasing the sequence of incoming ADU frames sequentially to the ADU->MP3 conversion process (step 7), without reordering. (Note also the pseudo-code in appendix B.2.)

7. MIME registration

MIME media type name: audio

MIME subtype: mpa-robust

Required parameters: none

Optional parameters: none

Encoding considerations:
This type is defined only for transfer via RTP as specified in "RFC 3119".

Security considerations:
See the "Security Considerations" section of "RFC 3119".

Interoperability considerations:
This encoding is incompatible with both the "audio/mpa" and "audio/mpeg" mime types.

Published specification:
The ISO/IEC MPEG-1 [3] and MPEG-2 [4] audio specifications, and "RFC 3119".

Applications which use this media type:
Audio streaming tools (transmitting and receiving)

Additional information: none

Person & email address to contact for further information:
Ross Finlayson
finlayson@live.com

Intended usage: COMMON

Author/Change controller:
Author: Ross Finlayson
Change controller: IETF AVT Working Group

8. SDP usage

When conveying information by SDP [8], the encoding name SHALL be "mp3" (the same as the MIME subtype). An example of the media representation in SDP is:

```
m=audio 49000 RTP/AVP 121
a=rtpmap:121 mpa-robust/90000
```

9. Security Considerations

If a session using this payload format is being encrypted, and interleaving is being used, then the sender SHOULD ensure that any change of encryption key coincides with a start of a new interleave cycle. Apart from this, the security considerations for this payload format are identical to those noted for RFC 2250 [2].

10. Acknowledgements

The suggestion of adding an interleaving option (using the first bits of the MPEG 'syncword' - which would otherwise be all-ones - as an interleaving index) is due to Dave Singer and Stefan Gewinner. In addition, Dave Singer provided valuable feedback that helped clarify and improve the description of this payload format. Feedback from Chris Sloan led to the addition of an "ADU descriptor" preceding each ADU frame in the RTP packet.

11. References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Hoffman, D., Fernando, G., Goyal, V. and M. Civanlar, "RTP Payload Format for MPEG1/MPEG2 Video", RFC 2250, January 1998.

[3] ISO/IEC International Standard 11172-3; "Coding of moving pictures and associated audio for digital storage media up to about 1,5 Mbits/s - Part 3: Audio", 1993.

[4] ISO/IEC International Standard 13818-3; "Generic coding of moving pictures and associated audio information - Part 3: Audio", 1998.

[5] Handley, M., "Guidelines for Writers of RTP Payload Format Specifications", BCP 36, RFC 2736, December 1999.

[6] Schulzrinne, H., "RTP Profile for Audio and Video Conferences with Minimal Control", RFC 1890, January 1996.

[7] Marshall Eubanks, personal communication, December 2000.

[8] Handley, M. and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

11. Author’s Address

Ross Finlayson,
Live Networks, Inc. (LIVE.COM)

EMail: finlayson@live.com
WWW: http://www.live.com/

Appendix A. Translating Between "MP3 Frames" and "ADU Frames"

The following 'pseudo code' describes how a sender using this payload format can translate a sequence of regular "MP3 Frames" to "ADU Frames", and how a receiver can perform the reverse translation: from "ADU Frames" to "MP3 Frames".

We first define the following abstract data structures:

- "Segment": A record that represents either a "MP3 Frame" or an "ADU Frame". It consists of the following fields:
  - "header": the 4-byte MPEG header
  - "headerSize": a constant (== 4)
  - "sideInfo": the 'side info' structure, *including* the optional 2-byte CRC field, if present
  - "sideInfoSize": the size (in bytes) of the above structure
  - "frameData": the remaining data in this frame
  - "frameDataSize": the size (in bytes) of the above data
  - "backpointer": the size (in bytes) of the backpointer for this frame
  - "aduDataSize": the size (in bytes) of the ADU associated with this frame. (If the frame is already an "ADU Frame", then aduDataSize == frameDataSize)
  - "mp3FrameSize": the total size (in bytes) that this frame would have if it were a regular "MP3 Frame". (If it is already a "MP3 Frame", then mp3FrameSize == headerSize + sideInfoSize + frameDataSize) Note that this size can be derived completely from "header".

- "SegmentQueue": A FIFO queue of "Segment"s, with operations
  - void enqueue(Segment)
  - Segment dequeue()
  - Boolean isEmpty()
  - Segment head()
  - Segment tail()
  - Segment previous(Segment): returns the segment prior to a given one
  - Segment next(Segment): returns the segment after a given one
  - unsigned totalDataSize(): returns the sum of the "frameDataSize" fields of each entry in the queue

A.1 Converting a sequence of "MP3 Frames" to a sequence of "ADU Frames":

```
SegmentQueue pendingMP3Frames; // initially empty
while (1) {
        // Enqueue new MP3 Frames, until we have enough data to generate
        // the ADU for a frame:
        do {
                int totalDataSizeBefore
                        = pendingMP3Frames.totalDataSize();

                Segment newFrame = 'the next MP3 Frame';
                pendingMP3Frames.enqueue(newFrame);

                int totalDataSizeAfter
                        = pendingMP3Frames.totalDataSize();
        } while (totalDataSizeBefore < newFrame.backpointer ||
                  totalDataSizeAfter < newFrame.aduDataSize);

        // We now have enough data to generate the ADU for the most
        // recently enqueued frame (i.e., the tail of the queue).
        // (The earlier frames in the queue - if any - must be
        // discarded, as we don't have enough data to generate
        // their ADUs.)
        Segment tailFrame = pendingMP3Frames.tail();

        // Output the header and side info:
        output(tailFrame.header);
        output(tailFrame.sideInfo);

        // Go back to the frame that contains the start of our ADU data:
        int offset = 0;
        Segment curFrame = tailFrame;
        int prevBytes = tailFrame.backpointer;
        while (prevBytes > 0) {
                curFrame = pendingMP3Frames.previous(curFrame);
                int dataHere = curFrame.frameDataSize;
                if (dataHere < prevBytes) {
                        prevBytes -= dataHere;
                } else {
                        offset = dataHere - prevBytes;
                        break;
                }
        }

        // Dequeue any frames that we no longer need:
        while (pendingMP3Frames.head() != curFrame) {
                pendingMP3Frames.dequeue();
        }
```

```
        // Output, from the remaining frames, the ADU data that we want:
        int bytesToUse = tailFrame.aduDataSize;
        while (bytesToUse > 0) {
                int dataHere = curFrame.frameDataSize - offset;
                int bytesUsedHere
                        = dataHere < bytesToUse ? dataHere : bytesToUse;

                output("bytesUsedHere" bytes from curFrame.frameData,
                        starting from "offset");

                bytesToUse -= bytesUsedHere;
                offset = 0;
                curFrame = pendingMP3Frames.next(curFrame);
        }
}
```

A.2 Converting a sequence of "ADU Frames" to a sequence of "MP3 Frames":

```
SegmentQueue pendingADUFrames; // initially empty
while (1) {
        while (needToGetAnADU()) {
                Segment newADU = 'the next ADU Frame';
                pendingADUFrames.enqueue(newADU);

                insertDummyADUsIfNecessary();
        }

        generateFrameFromHeadADU();
}

Boolean needToGetAnADU() {
        // Checks whether we need to enqueue one or more new ADUs before
        // we have enough data to generate a frame for the head ADU.
        Boolean needToEnqueue = True;

        if (!pendingADUFrames.isEmpty()) {
                Segment curADU = pendingADUFrames.head();
                int endOfHeadFrame = curADU.mp3FrameSize
                        - curADU.headerSize - curADU.sideInfoSize;
                int frameOffset = 0;

                while (1) {
                        int endOfData = frameOffset
                                - curADU.backpointer +
                                  curADU.aduDataSize;
                        if (endOfData >= endOfHeadFrame) {
                                // We have enough data to generate a
                                // frame.
```

```
                                needToEnqueue = False;
                                break;
                        }

                        frameOffset += curADU.mp3FrameSize
                                - curADU.headerSize
                                - curADU.sideInfoSize;
                        if (curADU == pendingADUFrames.tail()) break;
                        curADU = pendingADUFrames.next(curADU);
                }
        }

    return needToEnqueue;
}

void generateFrameFromHeadADU() {
        Segment curADU = pendingADUFrames.head();

        // Output the header and side info:
        output(curADU.header);
        output(curADU.sideInfo);

        // Begin by zeroing out the rest of the frame, in case the ADU
        // data doesn't fill it in completely:
        int endOfHeadFrame = curADU.mp3FrameSize
                - curADU.headerSize - curADU.sideInfoSize;
        output("endOfHeadFrame" zero bytes);

        // Fill in the frame with appropriate ADU data from this and
        // subsequent ADUs:
        int frameOffset = 0;
        int toOffset = 0;

        while (toOffset < endOfHeadFrame) {
                int startOfData = frameOffset - curADU.backpointer;
                if (startOfData > endOfHeadFrame) {
                        break; // no more ADUs are needed
                }
                int endOfData = startOfData + curADU.aduDataSize;
                if (endOfData > endOfHeadFrame) {
                        endOfData = endOfHeadFrame;
                }

                int fromOffset;
                if (startOfData <= toOffset) {
                        fromOffset = toOffset - startOfData;
                        startOfData = toOffset;
                        if (endOfData < startOfData) {
```

```
                                endOfData = startOfData;
                        }
                } else {
                        fromOffset = 0;

                        // leave some zero bytes beforehand:
                        toOffset = startOfData;
                }

                int bytesUsedHere = endOfData - startOfData;
                output(starting at offset "toOffset, "bytesUsedHere"
                        bytes from "&curADU.frameData[fromOffset]");
                toOffset += bytesUsedHere;

                frameOffset += curADU.mp3FrameSize
                        - curADU.headerSize - curADU.sideInfoSize;
                curADU = pendingADUFrames.next(curADU);
        }

        pendingADUFrames.dequeue();
}

void insertDummyADUsIfNecessary() {
        // The tail segment (ADU) is assumed to have been recently
        // enqueued.  If its backpointer would overlap the data
        // of the previous ADU, then we need to insert one or more
        // empty, 'dummy' ADUs ahead of it.  (This situation should
        // occur only if an intermediate ADU was missing - e.g., due
        // to packet loss.)
        while (1) {
                Segment tailADU = pendingADUFrames.tail();
                int prevADUend; // relative to the start of the tail ADU

                if (pendingADUFrames.head() != tailADU) {
                        // there is a previous ADU
                        Segment prevADU
                                = pendingADUFrames.previous(tailADU);
                        prevADUend
                                = prevADU.mp3FrameSize +
                                  prevADU.backpointer
                                  - prevADU.headerSize
                                  - curADU.sideInfoSize;
                        if (prevADU.aduDataSize > prevADUend) {
                                // this shouldn't happen if the previous
                                // ADU was well-formed
                                prevADUend = 0;
                        } else {
                                prevADUend -= prevADU.aduDataSize;
```

```
                        }
                } else {
                        prevADUend = 0;
                }

                if (tailADU.backpointer > prevADUend) {
                        // Insert a 'dummy' ADU in front of the tail.
                        // This ADU can have the same "header" (and thus
                        // "mp3FrameSize") as the tail ADU, but should
                        // have an "aduDataSize" of zero.  The simplest
                        // way to do this is to copy the "sideInfo" from
                        // the tail ADU, and zero out the
                        // "main_data_begin" and all of the
                        // "part2_3_length" fields.
                } else {
                        break; // no more dummy ADUs need to be inserted
                }
        }
}
```

Appendix B: Interleaving and Deinterleaving

The following 'pseudo code' describes how a sender can reorder a sequence of "ADU Frames" according to an interleaving pattern (step 2), and how a receiver can perform the reverse reordering (step 6).

B.1 Interleaving a sequence of "ADU Frames":

We first define the following abstract data structures:

- "interleaveCycleSize": an integer in the range [1,256] -
  "interleaveCycle": an array, of size "interleaveCycleSize", containing some permutation of the integers from the set [0 .. interleaveCycleSize-1]
  e.g., if "interleaveCycleSize" == 8, "interleaveCycle" might contain: 1,3,5,7,0,2,4,6
- "inverseInterleaveCycle": an array containing the inverse of the permutation in "interleaveCycle" - i.e., such that interleaveCycle[inverseInterleaveCycle[i]] == i
- "ii": the current Interleave Index (initially 0)
- "icc": the current Interleave Cycle Count (initially 0)
- "aduFrameBuffer": an array, of size "interleaveCycleSize", of ADU Frames that are awaiting packetization

```
while (1) {
        int positionOfNextFrame = inverseInterleaveCycle[ii];
        aduFrameBuffer[positionOfNextFrame] = the next ADU frame;
        replace the high-order 11 bits of this frame's MPEG header
```

```
            with (ii,icc);
                // Note: Be sure to leave the remaining 21 bits as is
        if (++ii == interleaveCycleSize) {
                // We've finished this cycle, so pass all
                // pending frames to the packetizing step
                for (int i = 0; i < interleaveCycleSize; ++i) {
                        pass aduFrameBuffer[i] to the packetizing step;
                }

                ii = 0;
                icc = (icc+1)%8;
        }
}
```

B.2 Deinterleaving a sequence of (interleaved) "ADU Frames":

We first define the following abstract data structures:

- "ii": the Interleave Index from the current incoming ADU frame
- "icc": the Interleave Cycle Count from the current incoming ADU frame
- "iiLastSeen": the most recently seen Interleave Index (initially, some integer *not* in the range [0,255])
- "iccLastSeen": the most recently seen Interleave Cycle Count (initially, some integer *not* in the range [0,7])
- "aduFrameBuffer": an array, of size 32, of (pointers to) ADU Frames that have just been depacketized (initially, all entries are NULL)

```
while (1) {
        aduFrame = the next ADU frame from the depacketizing step;
        (ii,icc) = "the high-order 11 bits of aduFrame's MPEG header";
        "the high-order 11 bits of aduFrame's MPEG header" = 0xFFE;
                // Note: Be sure to leave the remaining 21 bits as is

        if (icc != iccLastSeen || ii == iiLastSeen) {
                // We've started a new interleave cycle
                // (or interleaving was not used).  Release all
                // pending ADU frames to the ADU->MP3 conversion step:
                for (int i = 0; i < 32; ++i) {
                        if (aduFrameBuffer[i] != NULL) {
                                release aduFrameBuffer[i];
                                aduFrameBuffer[i] = NULL;
                        }
                }
        }

        iiLastSeen = ii;
```

```
        iccLastSeen = icc;
        aduFrameBuffer[ii] = aduFrame;
}
```

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.