Ogg logical bitstream framing

Ogg bitstreams

The Ogg transport bitstream is designed to provide framing, error protection and seeking structure for higher-level codec streams that consist of raw, unencapsulated data packets, such as the Vorbis audio codec or Theora video codec.

Application example: Vorbis

Vorbis encodes short-time blocks of PCM data into raw packets of bit-packed data. These raw packets may be used directly by transport mechanisms that provide their own framing and packet-separation mechanisms (such as UDP datagrams). For stream based storage (such as files) and transport (such as TCP streams or pipes), Vorbis uses the Ogg bitstream format to provide framing/sync, sync recapture after error, landmarks during seeking, and enough information to properly separate data back into packets at the original packet boundaries without relying on decoding to find packet boundaries.

Design constraints for Ogg bitstreams

1. True streaming; we must not need to seek to build a 100% complete bitstream.
2. Use no more than approximately 1-2% of bitstream bandwidth for packet boundary marking, high-level framing, sync and seeking.
3. Specification of absolute position within the original sample stream.
4. Simple mechanism to ease limited editing, such as a simplified concatenation mechanism.
5. Detection of corruption, recapture after error and direct, random access to data at arbitrary positions in the bitstream.

Logical and Physical Bitstreams

A *logical* Ogg bitstream is a contiguous stream of sequential pages belonging only to the logical bitstream. A *physical* Ogg bitstream is constructed from one or more than one logical Ogg bitstream (the simplest physical bitstream is simply a single logical bitstream). We describe below the exact formatting of an Ogg logical bitstream. Combining logical bitstreams into more complex physical bitstreams is described in the Ogg bitstream overview. The exact mapping of raw Vorbis packets into a valid Ogg Vorbis physical bitstream is described in the Vorbis I Specification.

Bitstream structure

An Ogg stream is structured by dividing incoming packets into segments of up to 255 bytes and then wrapping a group of contiguous packet segments into a variable length page preceded by a page header. Both the header size and page size are variable; the page header contains sizing information and checksum data to determine header/page size and data integrity.

The bitstream is captured (or recaptured) by looking for the beginning of a page, specifically the capture pattern. Once the capture pattern is found, the decoder verifies page sync and

integrity by computing and comparing the checksum. At that point, the decoder can extract the packets themselves.

Packet segmentation

Packets are logically divided into multiple segments before encoding into a page. Note that the segmentation and fragmentation process is a logical one; it's used to compute page header values and the original page data need not be disturbed, even when a packet spans page boundaries.

The raw packet is logically divided into [n] 255 byte segments and a last fractional segment of < 255 bytes. A packet size may well consist only of the trailing fractional segment, and a fractional segment may be zero length. These values, called "lacing values" are then saved and placed into the header segment table.

An example should make the basic concept clear:

```
raw packet:
  ___________________________________________________
 |_________________packet data_____________________| 753 bytes

lacing values for page header segment table: 255,255,243
```

We simply add the lacing values for the total size; the last lacing value for a packet is always the value that is less than 255. Note that this encoding both avoids imposing a maximum packet size as well as imposing minimum overhead on small packets (as opposed to, eg, simply using two bytes at the head of every packet and having a max packet size of 32k. Small packets (<255, the typical case) are penalized with twice the segmentation overhead). Using the lacing values as suggested, small packets see the minimum possible byte-aligned overhead (1 byte) and large packets, over 512 bytes or so, see a fairly constant ~.5% overhead on encoding space.

Note that a lacing value of 255 implies that a second lacing value follows in the packet, and a value of < 255 marks the end of the packet after that many additional bytes. A packet of 255 bytes (or a multiple of 255 bytes) is terminated by a lacing value of 0:

```
raw packet:
  ___________________________________
 |________packet data_______________|          255 bytes

lacing values: 255, 0
```

Note also that a 'nil' (zero length) packet is not an error; it consists of nothing more than a lacing value of zero in the header.

Packets spanning pages

Packets are not restricted to beginning and ending within a page, although individual segments are, by definition, required to do so. Packets are not restricted to a maximum size, although excessively large packets in the data stream are discouraged.

After segmenting a packet, the encoder may decide not to place all the resulting segments into the current page; to do so, the encoder places the lacing values of the segments it wishes to belong to the current page into the current segment table, then finishes the page.

The next page is begun with the first value in the segment table belonging to the next packet segment, thus continuing the packet (data in the packet body must also correspond properly to the lacing values in the spanned pages. The segment data in the first packet corresponding to the lacing values of the first page belong in that page; packet segments listed in the segment table of the following page must begin the page body of the subsequent page).

The last mechanic to spanning a page boundary is to set the header flag in the new page to indicate that the first lacing value in the segment table continues rather than begins a packet; a header flag of 0x01 is set to indicate a continued packet. Although mandatory, it is not actually algorithmically necessary; one could inspect the preceding segment table to determine if the packet is new or continued. Adding the information to the packet_header flag allows a simpler design (with no overhead) that needs only inspect the current page header after frame capture. This also allows faster error recovery in the event that the packet originates in a corrupt preceding page, implying that the previous page's segment table cannot be trusted.

Note that a packet can span an arbitrary number of pages; the above spanning process is repeated for each spanned page boundary. Also a 'zero termination' on a packet size that is an even multiple of 255 must appear even if the lacing value appears in the next page as a zero-length continuation of the current packet. The header flag should be set to 0x01 to indicate that the packet spanned, even though the span is a nil case as far as data is concerned.

The encoding looks odd, but is properly optimized for speed and the expected case of the majority of packets being between 50 and 200 bytes (note that it is designed such that packets of wildly different sizes can be handled within the model; placing packet size restrictions on the encoder would have only slightly simplified design in page generation and increased overall encoder complexity).

The main point behind tracking individual packets (and packet segments) is to allow more flexible encoding tricks that requiring explicit knowledge of packet size. An example is simple bandwidth limiting, implemented by simply truncating packets in the nominal case if the packet is arranged so that the least sensitive portion of the data comes last.

Page header

The headering mechanism is designed to avoid copying and re-assembly of the packet data (ie, making the packet segmentation process a logical one); the header can be generated directly from incoming packet data. The encoder buffers packet data until it finishes a complete page at which point it writes the header followed by the buffered packet segments.

capture_pattern

A header begins with a capture pattern that simplifies identifying pages; once the decoder has found the capture pattern it can do a more intensive job of verifying that it has in fact found a page boundary (as opposed to an inadvertent coincidence in the byte stream).

```
byte value

 0  0x4f 'O'
 1  0x67 'g'
 2  0x67 'g'
 3  0x53 'S'
```

stream_structure_version

The capture pattern is followed by the stream structure revision:

```
byte value

 4  0x00
```

header_type_flag

The header type flag identifies this page's context in the bitstream:

```
byte value

 5  bitflags: 0x01: unset = fresh packet
                    set = continued packet
              0x02: unset = not first page of logical bitstream
                      set = first page of logical bitstream (bos)
              0x04: unset = not last page of logical bitstream
                      set = last page of logical bitstream (eos)
```

absolute granule position

(This is packed in the same way the rest of Ogg data is packed; LSb of LSB first. Note that the 'position' data specifies a 'sample' number (eg, in a CD quality sample is four octets, 16 bits for left and 16 bits for right; in video it would likely be the frame number. It is up to the specific codec in use to define the semantic meaning of the granule position value). The position specified is the total samples encoded after including all packets finished on this page (packets begun on this page but continuing on to the next page do not count). The rationale here is that the position specified in the frame header of the last page tells how long the data coded by the bitstream is. A truncated stream will still return the proper number of samples that can be decoded fully.

A special value of '-1' (in two's complement) indicates that no packets finish on this page.

```
byte value

 6  0xXX LSB
 7  0xXX
 8  0xXX
 9  0xXX
10  0xXX
11  0xXX
12  0xXX
13  0xXX MSB
```

stream serial number

Ogg allows for separate logical bitstreams to be mixed at page granularity in a physical bitstream. The most common case would be sequential arrangement, but it is possible to interleave pages for two separate bitstreams to be decoded concurrently. The serial number is the means by which pages physical pages are associated with a particular logical stream. Each logical stream must have a unique serial number within a physical stream:

```
byte value

14  0xXX LSB
15  0xXX
16  0xXX
17  0xXX MSB
```

page sequence no

Page counter; lets us know if a page is lost (useful where packets span page boundaries).

```
byte value

18  0xXX LSB
19  0xXX
20  0xXX
21  0xXX MSB
```

page checksum

32 bit CRC value (direct algorithm, initial val and final XOR = 0, generator polynomial=0x04c11db7). The value is computed over the entire header (with the CRC field in the header set to zero) and then continued over the page. The CRC field is then filled with the computed value.

(A thorough discussion of CRC algorithms can be found in "A Painless Guide to CRC Error Detection Algorithms" by Ross Williams ross@ross.net.)

```
byte value

22  0xXX LSB
23  0xXX
24  0xXX
25  0xXX MSB
```

page_segments

The number of segment entries to appear in the segment table. The maximum number of 255 segments (255 bytes each) sets the maximum possible physical page size at 65307 bytes or just under 64kB (thus we know that a header corrupted so as destroy sizing/alignment information will not cause a runaway bitstream. We'll read in the page according to the corrupted size information that's guaranteed to be a reasonable size regardless, notice the checksum mismatch, drop sync and then look for recapture).

```
byte value

26 0x00-0xff (0-255)
```

segment_table (containing packet lacing values)

The lacing values for each packet segment physically appearing in this page are listed in contiguous order.

```
byte value

27 0x00-0xff (0-255)
[...]
n  0x00-0xff (0-255, n=page_segments+26)
```

Total page size is calculated directly from the known header size and lacing values in the segment table. Packet data segments follow immediately after the header.

Page headers typically impose a flat .25-.5% space overhead assuming nominal ~8k page sizes. The segmentation table needed for exact packet recovery in the streaming layer adds approximately .5-1% nominal assuming expected encoder behavior in the 44.1kHz, 128kbps stereo encodings.