Ogg Vorbis Specification and Decoding

Table of Contents

Ogg Specification 4
Introduction 4
Ogg Bitstream 4
Design constraints for Ogg bitstreams 4
Logical bitstream 4
Physical Bitstream 5
Bitstream structure 5
The encapsulation process 5
Multiplexing 7
The Ogg page format 8
1. capture_pattern 9
2. stream_structure_version 9
4. Absolute_granule_position 10
5. bitstream_serial_number 10
6. page_sequence_number 10
7. CRC_checksum 11
8. number_page_segments 11
9. segment_table 11
Ogg Decoding 12
Flowchart for the decoding process 13
Vorbis I Specification 14
Introduction 14
General Overview 14
Classification 14
Vorbis - Encoder and Decoder 14
Vorbis Encoder 15
Vorbis Decoder 15
Analysis and Synthesis 15
Coding and Decoding 15
Vorbis streaming and stream decomposition 15
Decode Overview 16
High-level Decode Process 17
Decode Setup 17
Decode Procedure 18
Explanation of the Decode Process 20
Packet Decode Process 20
Mode Decode 21
Window shape Decode (long Windows only) 21
Floor Decode 21
Residue Decode 21
Inverse Channel Coupling 22

Generate Floor Curve 22
Compute floor /residue dot product 22
Inverse Monolithic transform (MDCT) 22
Overlap/add data 22
Cache right hand data 23
Return finished audio data 23
Bitpacking Convention 23
Overview 23
Octets, bytes and words 23
Probability Model and Code Books 23
Huffman Decision tree representation 26
VQ lookup table vector representation 26
Codec Setup and Packet Decode 26
Header Decode and Decode Setup 26
Identification Header 27
Comment Header 27
Setup Header 28
Audio Packet decode and synthesis 28
Ogg Vorbis Decoder Programming Documentation using VorbisFile 29
Introduction 29
Programming with Vorbisfile 29
Base data structures 29
Setup/ Tear down 29
Decoding 30
Seeking 30
File Information 30
OggVorbis_File 30
Decoder Flowchart 33
Conclusion 43

Ogg Specification

Introduction

Ogg codecs use octet vectors of raw, compressed data or packets. These compressed packets do not have any high-level structure or boundary information; strung together, they appear to be streams of random bytes with no landmarks.

Ogg Vorbis uses the Ogg framework to encapsulate Vorbis-encoded audio data for stream-based storage, such as files, and transport, such as TCP streams or pipes.

Ogg Bitstream

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 Tarkin video codec.

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

- Framing for logical bitstreams.
- Interleaving of different logical bitstreams.
- Detection of corruption.
- Recapture after a parsing error.
- Position landmarks for direct random access of arbitrary positions in the bitstream.
- Streaming capability (no seeking is needed to build a 100% complete bitstream).
- Small overhead (using no more than approximately 1-2% of bitstream bandwidth for packet boundary marking, high-level framing, sync and seeking).
- Simplicity to enable fast parsing.
- Simple concatenation mechanism of several physical bitstreams.

Ogg provides a generic framework to perform encapsulation of time-continuous bitstreams. It does not know any specifics about the codec data that it encapsulates and is thus independent of any media codec.

Logical bitstream

A 'logical bitstream' consists of 'pages', in order, belonging to a single codec instance. Raw packets are grouped and encoded into contiguous pages of structured bitstream data. Thus, logical bitstream is a sequence of bits being the result of an encoded media stream. A stream always consists of an integer number of pages. Each page is a self contained entity, though it is possible that a packet may be split and encoded across one or more pages; that is, the page

decode mechanism is designed to recognize, verify and handle single pages at a time from the overall bitstream.

Physical Bitstream

Multiple logical bitstreams can be combined, with restrictions, into a single 'physical bitstream'. Physical bitstream is a sequence of bits resulting from an Ogg encapsulation of one or several logical bitstreams. Logical bitstreams may not be mapped or multiplexed into physical bitstreams without restriction. Each 'media' format defines its own restrictive mapping. A physical bitstream consists of a sequence of pages from the logical bitstreams with the restriction that the pages of each logical bitstream must come in their correct temporal order.

Thus, a physical bitstream consists of multiple logical bitstreams interleaved in pages and may include a 'meta-header' at the beginning of the multiplexed logical stream that serves as identification magic. The simplest physical bitstream is a single, unmultiplexed logical bitstream with no meta-header; this is referred to as a 'degenerate stream'.

The logical bitstreams are identified by a unique serial number in the header of each page of the physical bitstream. This unique serial number is created randomly and does not have any connection to the content or encoder of the logical bitstream it represents. Pages of all logical bitstreams are concurrently interleaved, but they need not be in a regular order - they are only required to be consecutive within the logical bitstream.

Bitstream structure

The bitstreams provided by encoders are handed over to Ogg as so-called 'packets' with packet boundaries dependent on the encoding format. From Ogg's perspective, packets can be of any arbitrary size. Ogg divides each packet into 255 byte long chunks plus a final shorter chunk. These chunks are called 'Ogg segments'. 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. They do not have a header for themselves. A group of contiguous segments is wrapped 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.

Each Ogg page contains only one type of data as it belongs to one logical bitstream only. Pages are of variable size and have a page header containing encapsulation and error recovery information. The encapsulation specification for one or more logical bitstreams is called a 'media mapping'. A specific media mapping will define how to group or break up packets from a specific media encoder.

The encapsulation process

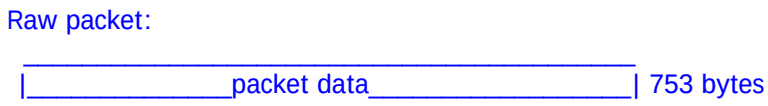
As Ogg pages have a maximum size of about 64 kB, sometimes a packet has to be distributed over several pages. As mentioned earlier, the raw packet is logically divided into [n] 255 byte segments and a last fractional segment of < 255 bytes. A packet size may well consist only of the trailing fractional segment, and a fractional segment may be zero length. These values, called 'lacing values' are then saved and placed into the header segment table.

Packets are not restricted to beginning and ending within a page, though individual segments are required to do so. The Ogg bitstream specification strongly recommends nominal page size of

approximately 4-8kB. 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, and 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 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. The above spanning process is repeated for each spanned page boundary. 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.

A segment table in the page header tells about the lacing values of the segments included in the page. A flag in the page header tells whether a page contains a packet continued from a previous page or not. Note that a lacing value of 255 implies that a second lacing value follows in the packet, and a value of less than 255 marks the end of the packet after that many additional bytes.



Lacing values for page header segment table: 255,255,243

A packet of 255 bytes (or a multiple of 255 bytes) is terminated by a lacing value of 0. A 'nil' (zero length) packet is not an error; it consists of nothing more than a lacing value of zero in the header.

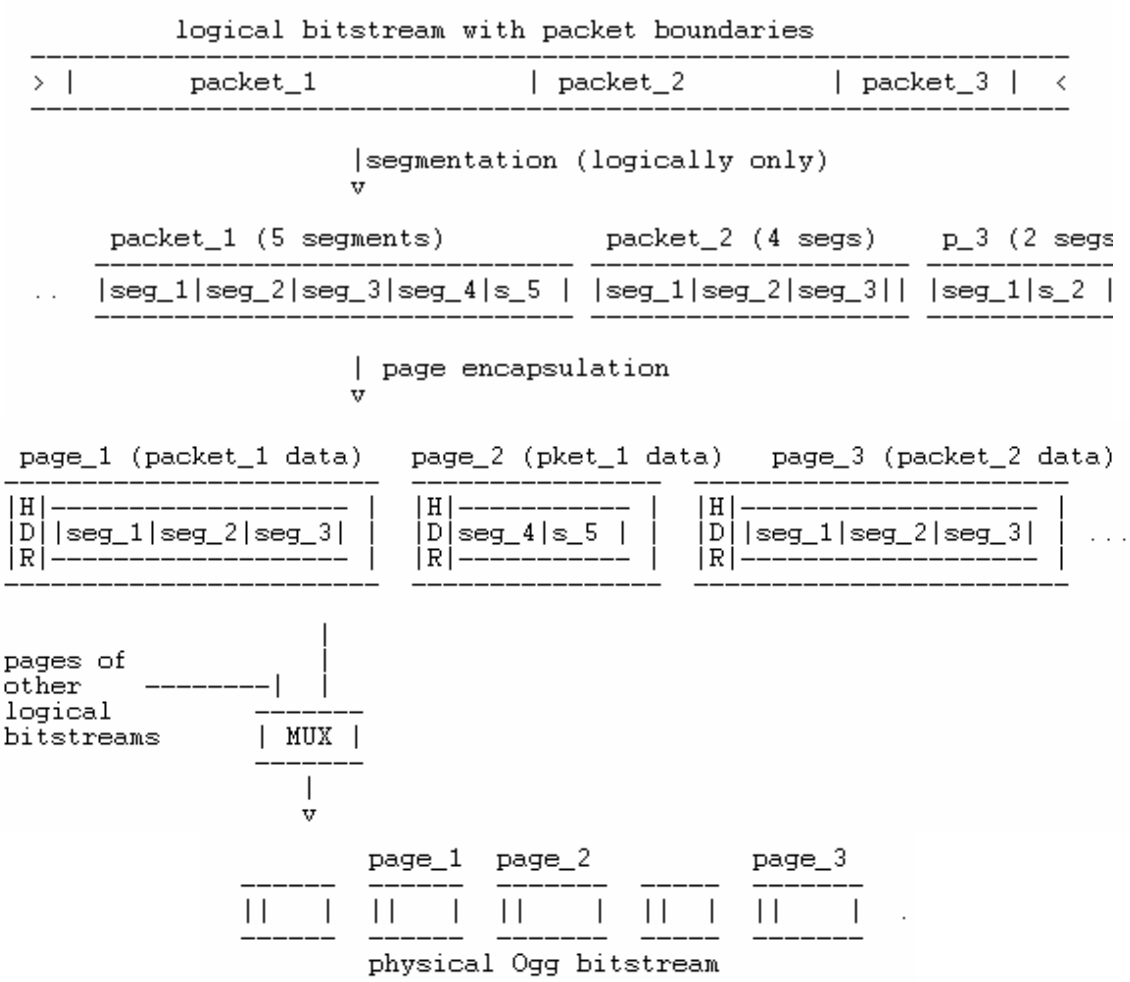
Lacing values for page header segment table: 255,255,243

Each logical bitstream in a physical Ogg bitstream starts with a special page (BOS=beginning of stream) and ends with a special page (EOS=end of stream). The first page of a logical Ogg bitstream consists of a single, small 'initial header' packet that includes sufficient information to identify the exact codec type and media requirements of the logical bitstream. BOS page is the initial page (beginning of stream) of a logical bitstream, which contains information to identify the codec type and other decoding-relevant information. It should contain the sample rate and number of channels. By convention, the first bytes of the BOS page contain magic data that

uniquely identifies the required codec. Ogg Vorbis provides the name and revision of the Vorbis codec, the audio rate and the audio quality in the Ogg Vorbis BOS page. There is no fixed way to detect the end of the codec-identifying marker. The format of the BOS page is dependent on the codec and therefore must be given in the encapsulation specification of that logical bitstream type.

Ogg also allows, but, does not require, secondary header packets after the BOS page for logical bitstreams and these must also precede any data packets in any logical bitstream. These subsequent header packets are framed into an integral number of pages, which will not contain any data packets. Ogg Vorbis uses two additional header pages per logical bitstream. The Ogg Vorbis BOS page starts with the byte 0x01, followed by 'vorbis' (a total of 7 bytes of identifier).

So, a physical bitstream begins with the BOS pages of all logical bitstreams containing one initial header packet per page, followed by the subsidiary header packets of all streams, followed by pages containing data packets. The following diagram shows a schematic example of a media mapping using Ogg and grouped logical bitstreams:



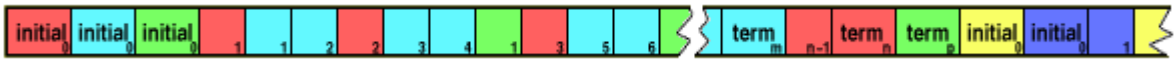
Multiplexing

Ogg knows two types of multiplexing: concurrent multiplexing ('grouping') and sequential multiplexing ('chaining'). Ogg Vorbis uses chaining. Chaining is defined to provide a simple

mechanism to concatenate physical Ogg bitstreams, as is often needed for streaming applications. In chaining, complete logical bitstreams are concatenated. The bitstreams do not overlap - the BOS page of the next immediately follows the EOS page of a given logical bitstream. Each chained logical bitstream must have a unique serial number within the scope of the physical bitstream. Ogg Vorbis also requires that all the EOS pages for all grouped bitstreams need to appear in direct sequence.

Logical bitstreams may also be multiplexed in parallel (grouped). An example of grouping would be to allow streaming of separate audio and video streams, using different codecs and different logical bitstreams in the same physical bitstream. Whole pages from multiple logical bitstreams are mixed together. The initial pages of each logical bitstream must appear first; the media mapping specifies the order of the initial pages. Unlike initial pages, terminal pages for the logical bitstreams need not all occur contiguously. Terminal pages may be 'nil' pages, containing no content but simply a page header with position information and the 'last page of bitstream' flag set in the page header. Each grouped bitstream must have a unique serial number within the scope of the physical bitstream.

Groups of concurrently multiplexed bitstreams may be chained consecutively. Such a physical bitstream obeys all the rules of both grouped and chained multiplexed streams. Consider the following example of a grouped and chained bitstream:



We see pages from five total logical bitstreams multiplexed into a physical bitstream.

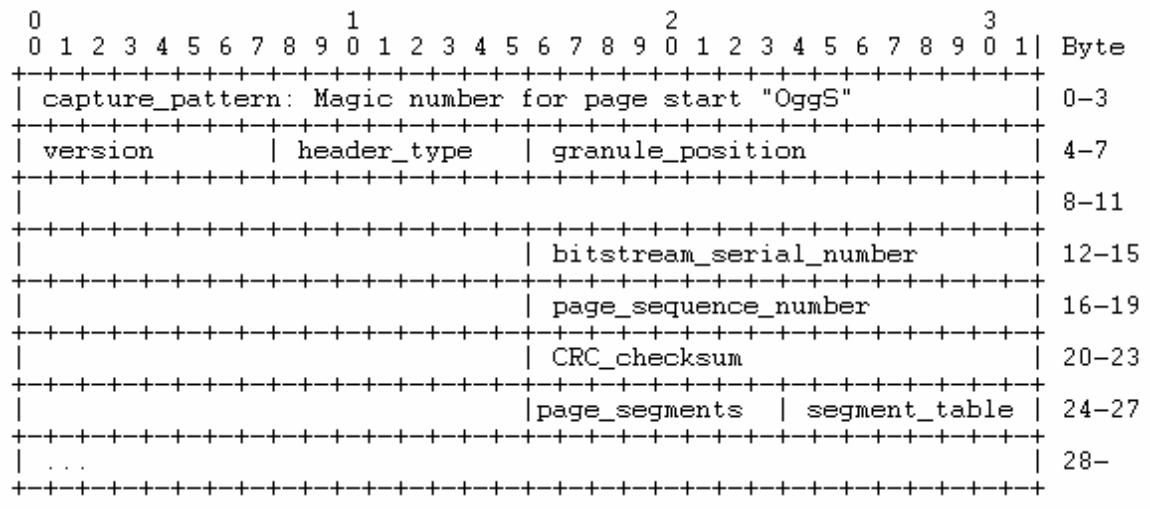
- Grouped bitstreams begin together; all of the initial pages must appear before any data pages. When concurrently multiplexed groups are chained, the new group does not begin until all the bitstreams in the previous group have terminated.
- The pages of concurrently multiplexed bitstreams need not conform to a regular order; the only requirement is that page n of a logical bitstream follow page n-1 in the physical bitstream. There are no restrictions on intervening pages belonging to other logical bitstreams.

Ogg demultiplexing reconstructs the original logical bitstreams from the physical bitstream by taking the pages in order from the physical bitstream and redirecting them into the appropriate logical decoding entity.

The Ogg page format

A physical Ogg bitstream consists of a sequence of concatenated pages. Pages are of variable size, usually 4-8 kB, and maximum 65307 bytes. A page header contains all the information needed to demultiplex the logical bitstreams out of the physical bitstream and to perform basic error recovery and landmarks for seeking. Each page is a self-contained entity such that the page-decode mechanism can recognize, verify, and handle single pages at a time without requiring the overall bitstream.

The Ogg page header has the following format:



The LSb (least significant bit) comes first in the Bytes. Fields with more than one byte length are encoded LSB (least significant byte) first.

The fields in the page header have the following meaning:

1. 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 found a page boundary. It is a 4-byte field that signifies the beginning of a page. It contains the magic pattern:

Byte	Value
0	0x4f 'O'
1	0x67 'g'
2	0x67 'g'
3	0x53 'S'

It helps a decoder to find the page boundaries and regain synchronization after parsing a corrupted stream. Once the capture pattern is found, the decoder verifies page sync and integrity by computing and comparing the checksum.

2. stream_structure_version

It is a single byte signifying the version number of the Ogg file format used in this stream (specified here as version 0). The capture pattern is followed by the stream structure revision:

Byte	Value
4	0x00

3. header_type_flag

The bits in this 1 byte field identify the specific type of this page (the page's context in the bitstream).

Value	Status	
0x01	set	continued packet
	unset	fresh packet
0x02	set	first page of logical bitstream (BOS)
	unset	not first page of logical bitstream
0x04	set	last page of logical bitstream (EOS)
	unset	not last page of logical bitstream

4. absolute_granule_position

This is an 8-byte field containing positional information. This is packed in the same way as the rest of Ogg data is packed. The 'position' data specifies a 'sample' number. 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). For audio stream, it may contain the total number of PCM samples encoded after including all frames finished on this page. This is a hint for the decoder and gives it some timing and position information. Its meaning is dependent on the codec for that logical bitstream and specified in a specific media mapping. 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

5. bitstream_serial_number

This is a 4-byte field containing the unique serial number by which the logical bitstream is identified. 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 of a physical stream 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

6. page_sequence_number

This is a 4-byte field containing the sequence number of the page so the decoder can identify page loss. This sequence number is increasing on each logical bitstream separately. Thus, it is a page counter, which 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

7. CRC_checksum

A 4-byte field containing a 32-bit CRC checksum of the page (including header with zero CRC field and page content).The generator polynomial is 0x04c11db7.

Byte	Value
22	0xXX LSB
23	0xXX
24	0xXX
25	0xXX MSB

8. number_page_segments

It is a single byte giving the number of segment entries encoded 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.

Byte	Value
26	0x00-0xff (0-255)

9. segment_table

It consists of number_page_segments bytes containing the lacing values of all segments in this page. Each byte contains one lacing value. 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.
The total header size in bytes is given by:

header_size = number_page_segments + 27 [Byte]

The total page size in Bytes is given by:

page_size = header_size + sum (lacing_values: 1...number_page_segments) [Byte]

Ogg Decoding

This decoding is based around the ogg synchronization layer. We read data into the synchronization layer, submit the data to the stream, and output raw packets to the decoder. 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. Decoding through the Ogg layer follows a specific logical sequence. The steps to be followed are as follows:

- Expose a buffer from the synchronization layer in order to read data..
- Read data into the buffer, using fread() or a similar function.
- Call a function to tell the synchronization layer how many bytes you wrote into the buffer.
- Write out the data by outputting a page from the synchronization layer.
- Submit the completed page to the streaming layer (using appropriate function).
- Output a packet of data to the codec-specific decoding engine.

Ogg also must handle headers, incomplete or dropped pages, and other errors in input.

- The buffer exposure is performed by ogg_sync_buffer().
- The data reading is done using fread() or a similar function.
- ogg_sync_wrote() tells the synchronization layer how many bytes you wrote into the buffer.
- ogg_sync_pageout() writes out the data by outputting a page from the synchronization layer.
- ogg_stream_pagein() submits the completed page to the streaming layer.
- ogg_stream_packetout() outputs a packet of data to the codec-specific decoding engine.

Flowchart for the decoding process

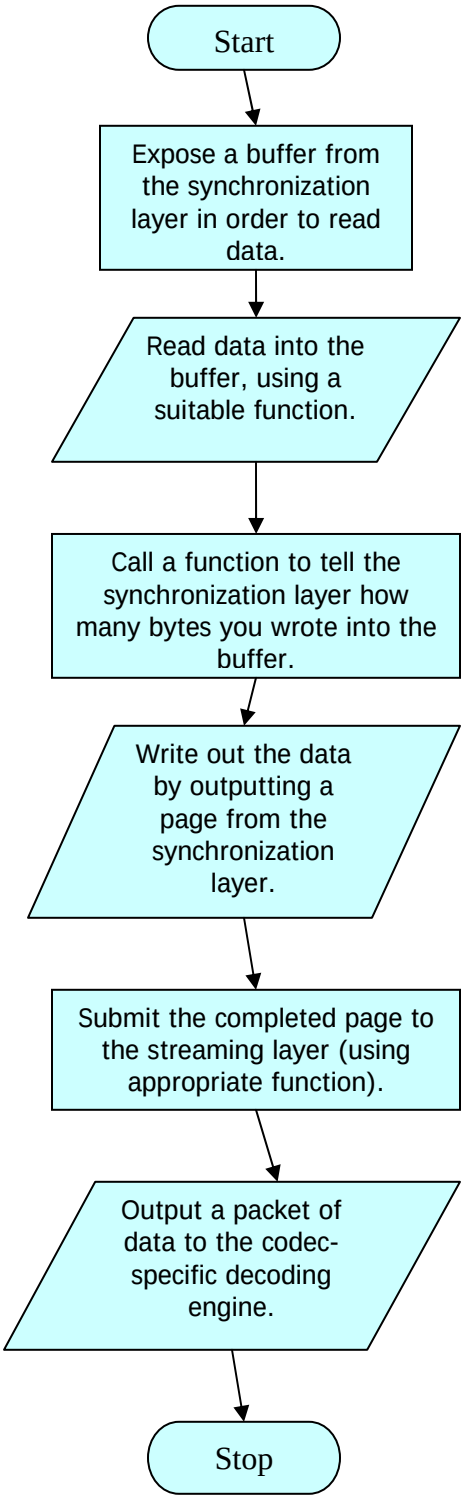


Figure: Flowchart of decoding steps

Vorbis I Specification

Introduction

General Overview

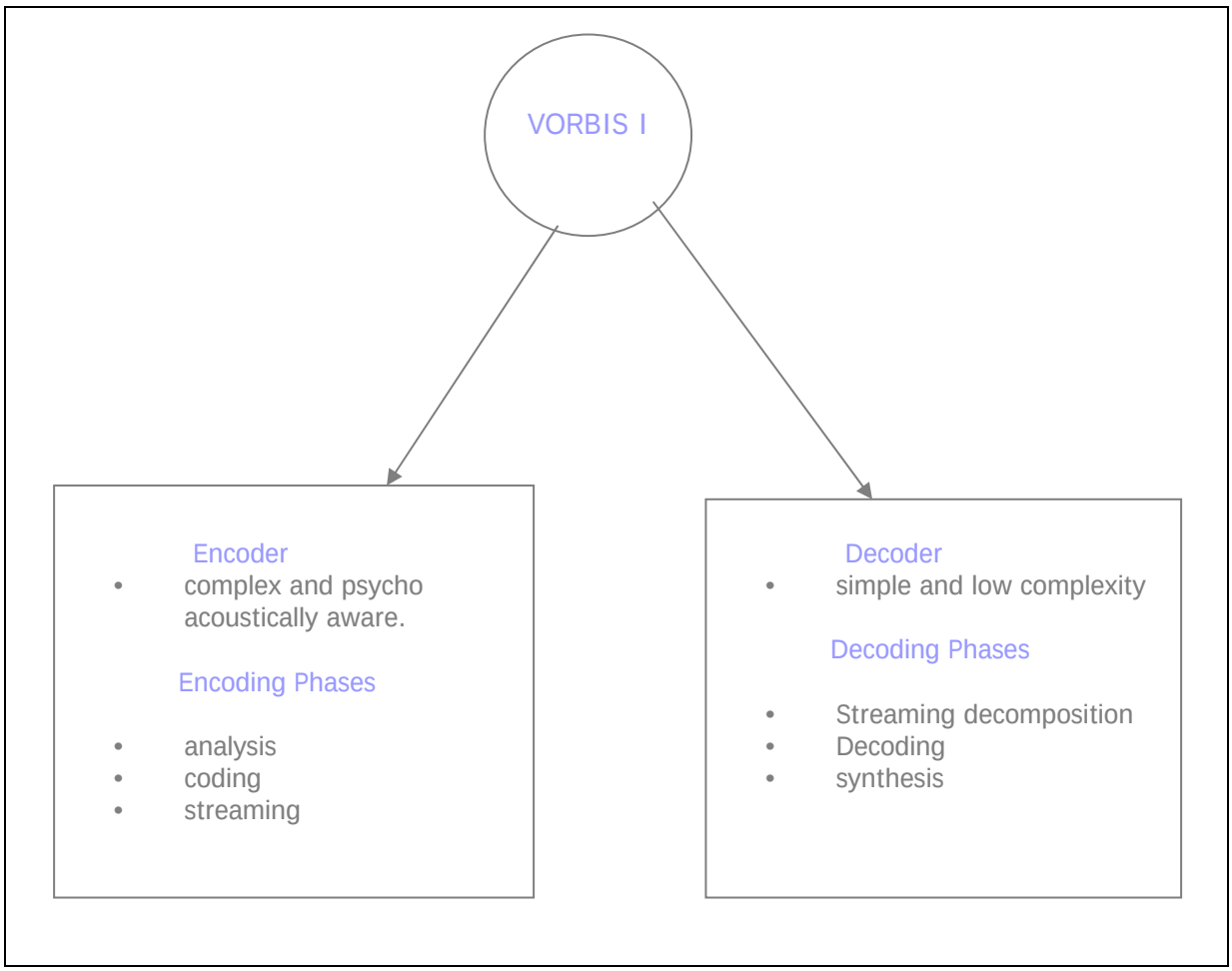
Vorbis

- Is a general-purpose perceptual audio CODEC.
- Allows maximum encoder flexibility.
- Can be scaled over exceptionally wide range of bit rates both lower and high bit rates.
- Is of the same league as MPEG-2 and MPC.

Classification

- Vorbis I - a forward adaptive monolithic transform codec based on Modified Discrete Cosine Transform.
- Vorbis II – a codec structured to allow addition hybrid wavelet filter banks to offer better transient response and reproduction using a transform better suited to localized time events.

Vorbis - Encoder and Decoder



Vorbis Encoder

A Vorbis encoder takes in overlapping (but contiguous) short-time segments of audio data. The encoder analyzes the content of the audio to determine an optimal compact representation; this phase of encoding is known as analysis. For each short-time block of sound, the encoder then packs an efficient representation of the signal, as determined by analysis, into a raw packet much smaller than the size required by the original signal; this phase is coding. Lastly, in a streaming environment, the raw packets are then structured into a continuous stream of octets; this last phase is streaming. Note that the stream of octets is referred to both as a 'byte-' and 'bit-'stream; the latter usage is acceptable as the stream of octets is a physical representation of a true logical bit-by-bit stream.

Vorbis Decoder

A Vorbis decoder performs a mirror image process of extracting the original sequence of raw packets from an Ogg stream (stream decomposition), reconstructing the signal representation from the raw data in the packet (decoding) and them reconstituting an audio signal from the decoded representation (synthesis).

Analysis and Synthesis

- Analysis begins by separating an input audio stream into individual, overlapping short-time segments of audio data. These segments are then transformed into an alternate representation, seeking to represent the original signal in a more efficient form that codes into a smaller number of bytes. The analysis and transformation stage is the most complex element of producing a Vorbis bitstream.
- The corresponding synthesis step in the decoder is simpler; there is no analysis to perform, merely a mechanical, deterministic reconstruction of the original audio data from the transform-domain representation.

Coding and Decoding

- Coding and decoding converts the transform-domain representation of the original audio produced by analysis to and from a bitwise packed raw data packet. Coding and decoding consist of two logically orthogonal concepts, back-end coding and bitpacking.
- Back-end coding uses a probability model to represent the raw numbers of the audio representation in as few physical bits as possible; familiar examples of back-end coding include Huffman coding and Vector Quantization.
- Bitpacking arranges the variable sized words of the back-end coding into a vector of octets without wasting space. The octets produced by coding a single short-time audio segment constitute one raw Vorbis packet.

Vorbis streaming and stream decomposition

Vorbis packets contain the raw, bitwise-compressed representation of a snippet of audio. These packets contain no structure and cannot be strung together directly into a stream; for streamed transmission and storage, Vorbis packets are encoded into an Ogg bitstream.

Decode Overview

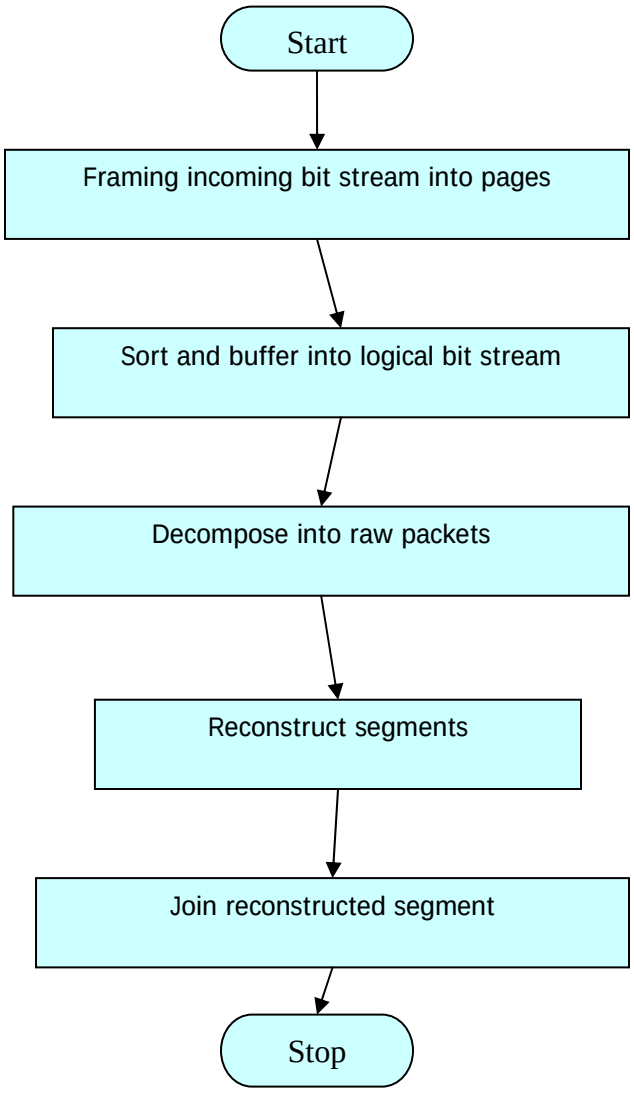
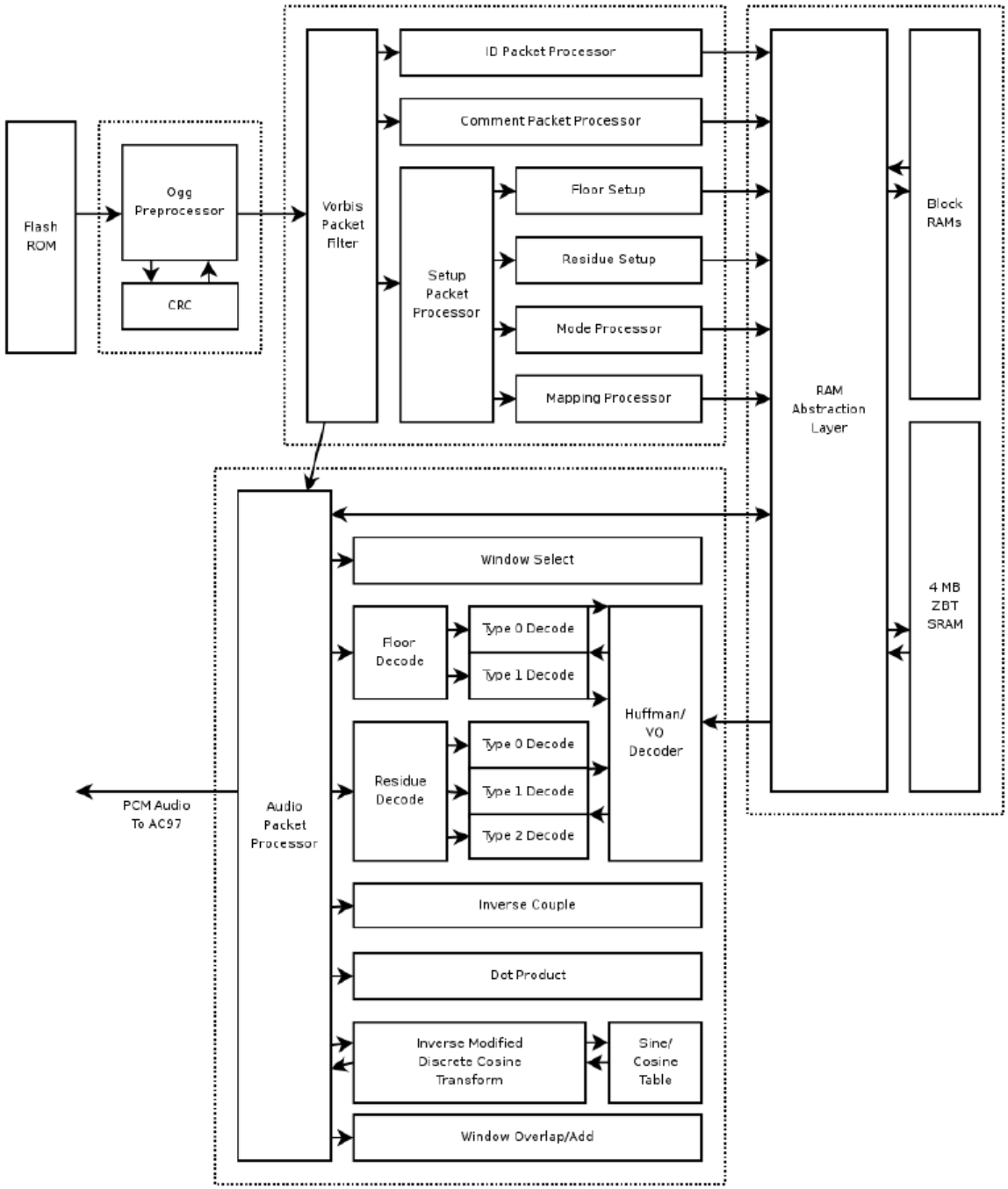


Figure: Flowchart of decoding steps

- ogg_sync_state buffer is used for framing, and ogg_stream_state – a logical bit stream buffer is used in sorting.
- Ogg bit stream manipulation structs – ogg_packet (one raw packet of data for decode) and ogg_page (one ogg bit stream page, vorbis packets are inside).
- There are many ogg bit stream manipulation functions that help to initializing buffers, to find version no., to find start and end of stream etc.

The following gives the Block Diagram of the Ogg Vorbis Decoder:



High-level Decode Process

Decode Setup

Before decoding can begin, a decoder must initialize using the bit stream headers matching the stream to be decoded. Vorbis uses three header packets; all are required, in-order, by this specification. Once set up, decode may begin at any audio packet belonging to the Vorbis

stream. In Vorbis I, all packets after the three initial headers are audio packets. The header packets are, in the order, as follows:

Identification Header → It is used to identify bit stream, Vorbis version, Sample rate and number of channels.

Comment Header → It includes user text comments ("tags") and a vendor string for the application/library that produced the bit stream.

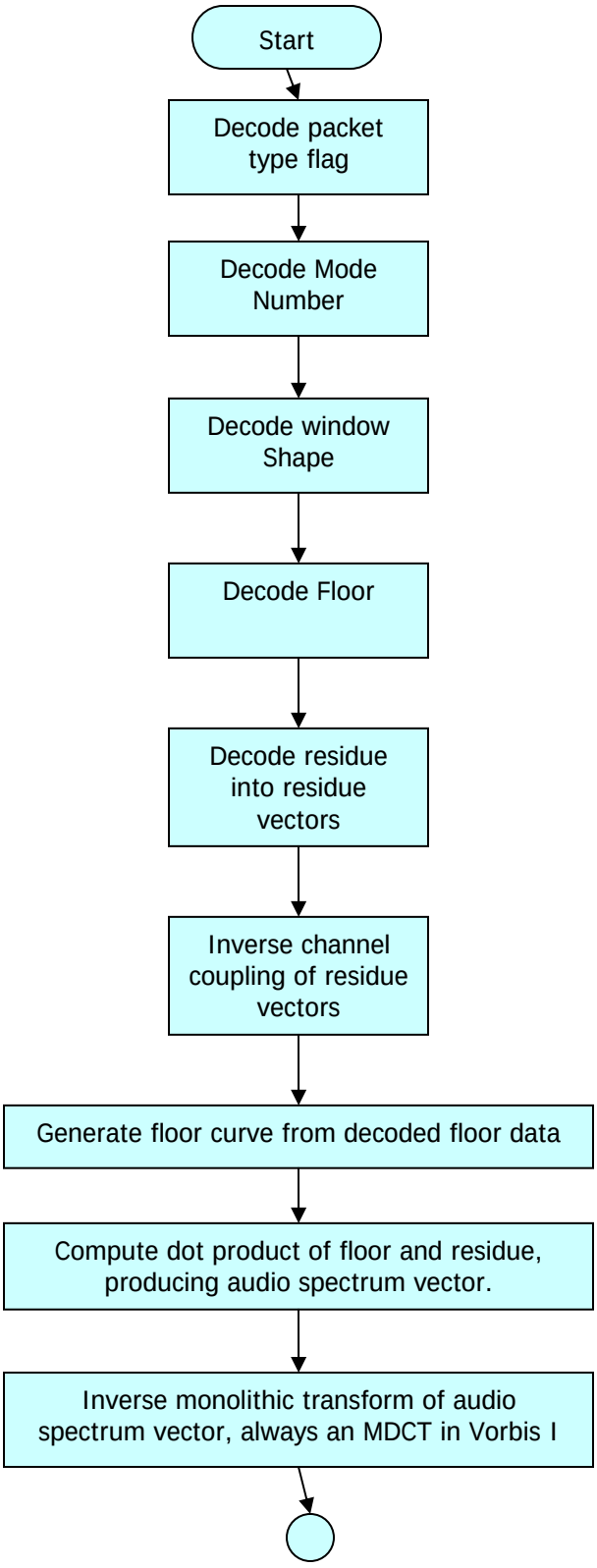
Setup Header → The setup header includes extensive CODEC setup information as well as the complete VQ and Huffman codebooks needed for decode.

Decode Procedure

Decoding and Synthesis Procedure for all audio packets is essentially same and shown in the flowchart.

We can take advantage of the symmetry of MDCT and store right hand transform data of a partial 50% inter frame buffer space savings and then we can complete the transform later before overlap/add with the next frame.

The flowchart for the same is given below:


Start
Decode packet type flag
Decode Mode Number
Decode window Shape
Decode Floor
Decode residue into residue vectors
Inverse channel coupling of residue vectors
Generate floor curve from decoded floor data
Compute dot product of floor and residue, producing audio spectrum vector.
Inverse monolithic transform of audio spectrum vector, always an MDCT in Vorbis I

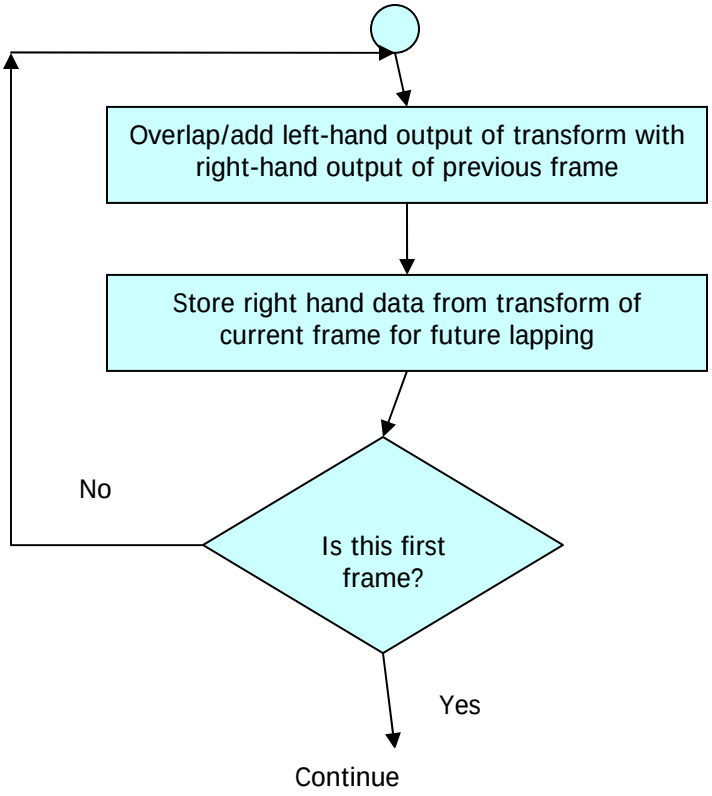
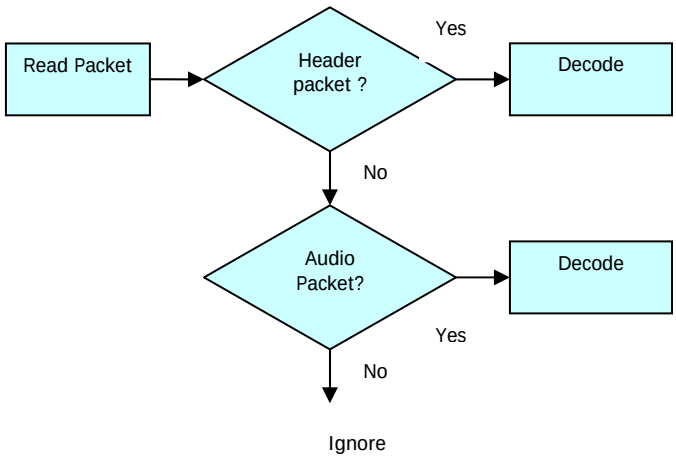


Figure: Flow chart of the Decode Procedure

Explanation of the Decode Process

Packet Decode Process

Vorbis I has 4 packets – first three types are header packets and are as described above and 4th type is audio packet. All other types are marked as reserved and must be ignored.



Mode Decode

Vorbis allows multiple, numbered packet ‘modes’.

Window shape Decode (long Windows only)

Vorbis uses an overlapping transform, namely the MDCT, to blend one frame into the next, avoiding most inter-frame block boundary artifacts. The MDCT output of one frame is windowed according to MDCT requirements overlapped 50% with the output of the previous frame and added. The window shape assures seamless reconstruction.

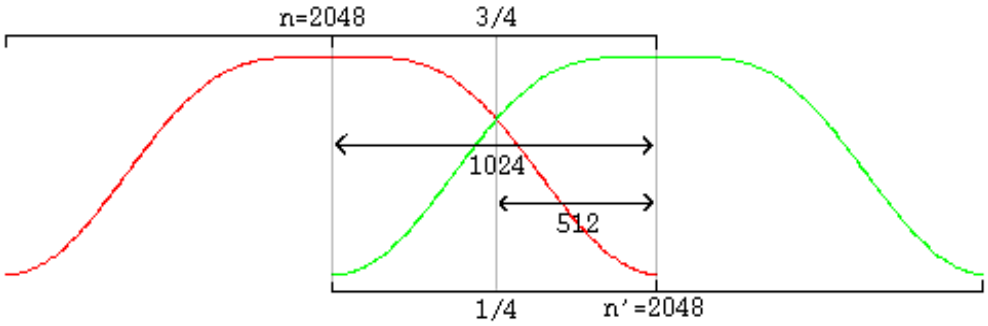


Figure : Equal sized windows overlap

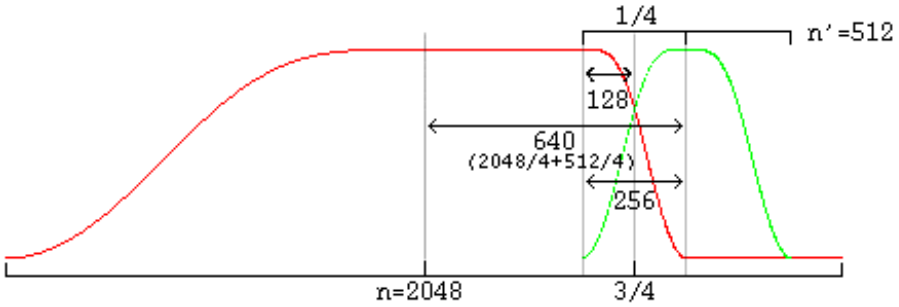


Figure: Overlap of unequal sized windows

In unequal-sized overlap, window shape must be modified for seamless lapping. Vorbis also codes two flag bits to specify pre and post – window shape.

Floor Decode

Each floor is encoded/decoded in channel order, however each floor belongs to a 'sub map' that specifies which floor configuration to use. All floors are decoded before residue decode begins.

Residue Decode

Although the number of residue vectors equals the number of channels, channel coupling may mean that the raw residue vectors extracted during decode do not map directly to specific channels. When channel coupling is in use, some vectors will correspond to coupled magnitude or angle. The coupling relationships are described in the codec setup and may differ from frame to frame, due to different mode numbers. Vorbis codes residue vectors in groups by sub map; the coding is done in sub map order from sub map 0 through n-1. This differs from floors which are coded using a configuration provided by sub map number, but are coded individually in channel order.

Inverse Channel Coupling

Vorbis coupling applies to pairs of residue vectors at a time; decoupling is done in-place a pair at a time in the order and using the vectors specified in the current mapping configuration. The decoupling operation is the same for all pairs, converting square polar representation (where one vector is magnitude and the second angle) back to Cartesian representation. After decoupling, in order, each pair of vectors on the coupling list, the resulting residue vectors represent the fine spectral detail of each output channel.

The inverse MDCT is known as the IMDCT. Because there are different numbers of inputs and outputs, at first glance it might seem that the MDCT should not be invertible. However, perfect inversion is achieved by *adding* the overlapped IMDCTs of subsequent overlapping blocks, causing the errors to *cancel* and the original data to be retrieved; this technique is known as time-domain aliasing cancellation (TDAC).

The IMDCT transforms N real numbers $X_0,\ldots, X_{N-1}$ into $2N$ real numbers $y_0,\ldots, y_{2N-1}$ according to the formula:

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2} + \frac{N}{2}\right)\left(k + \frac{1}{2}\right)\right]$$

(Like for the DCT-IV, an orthogonal transform, the inverse has the same form as the forward transform.)

In the case of a windowed MDCT with the usual window normalization, the normalization coefficient in front of the IMDCT should be multiplied by 2 (i.e., becoming $2/N$).

Generate Floor Curve

The decoder generates floor curve at appropriate times. It is generated when floor data is decoded from the raw packet, or after inverse coupling. Both floor 0 and floor 1 generate a linear-range, linear-domain output vector to be multiplied (dot product) by the linear-range, linear-domain spectral residue.

Compute floor / residue dot product

The decoder multiplies the floor curve and residue vectors element by element, producing the finished audio spectrum of each channel.

Inverse Monolithic transform (MDCT)

The audio spectrum is converted back into time domain PCM audio via an inverse Modified Discrete Cosine Transform (MDCT). Note that the PCM produced directly from the MDCT is not yet finished audio; it must be lapped with surrounding frames using an appropriate window (such as the Vorbis window) before the MDCT can be considered orthogonal.

Overlap / add data

Windowed MDCT output is overlapped and added with the right hand data of the previous window such that the 3/4-point of the previous window is aligned with the 1/4 point of the current window (as illustrated in the window overlap diagram). At this point, the audio data between the center of the previous frame and the center of the current frame is now finished and ready to be returned.

Cache right hand data

The decoder must cache the right hand portion of the current frame to be lapped with the left hand portion of the next frame.

Return finished audio data

The overlapped portion produced from overlapping the previous and current frame data is finished data to be returned by the decoder. This data spans from the center of the previous window to the center of the current window. In the case of same-sized windows, the amount of data to return is one-half block consisting of and only of the overlapped portions. When overlapping a short and long window, much of the returned range is not actually overlap. This does not damage transform orthogonality.

The amount of data to be returned is:

window_blocksize (previous_window) /4 + window_blocksize (current_window) / 4 from the center of the previous window to the center of the current window.

Data is not returned from the first frame; it must be used to 'prime' the decode engine. The encoder accounts for this priming when calculating PCM offsets; after the first frame, the proper PCM output offset is '0' (as no data has been returned yet).

Bitpacking Convention

Overview

- Vorbis codec uses relatively unstructured raw packets containing arbitrary binary integer fields.
- Packets are bitstreams. Bits coded one-by-one by encoder and read one-by-one monotonically by decoder.
- Octets, bytes, words are commonly used.

Octets, bytes and words

- Byte ordering used most commonly are little endian (0-1-2-3) and big endian (3-2-1-0), while mixed endian (0-2-1-3) is used less.
- Vorbis bitpacking convention specifies storage and bitstream manipulation at the byte level.

Probability Model and Code Books

1. Vorbis has no statically configured probability model, instead packing all entropy-decoding configuration, VQ and Huffman, into bitstream itself in the third header the codec setup header.
2. Packed configuration has multiple ‘codebooks’ and lookup tables.

Vorbis I supports three lookup types:

- No lookup.
- Implicitly populated value mapping(lattice VQ).
- Explicitly populated value mapping(foam VQ).

Codebook decode takes place before lookup.

Lookup type ‘0’ → no look up.
Lookup type ‘1’ and Lookup type ‘2’ are similar, both read the same vector list, but type ‘2’ reads each scalar explicitly.

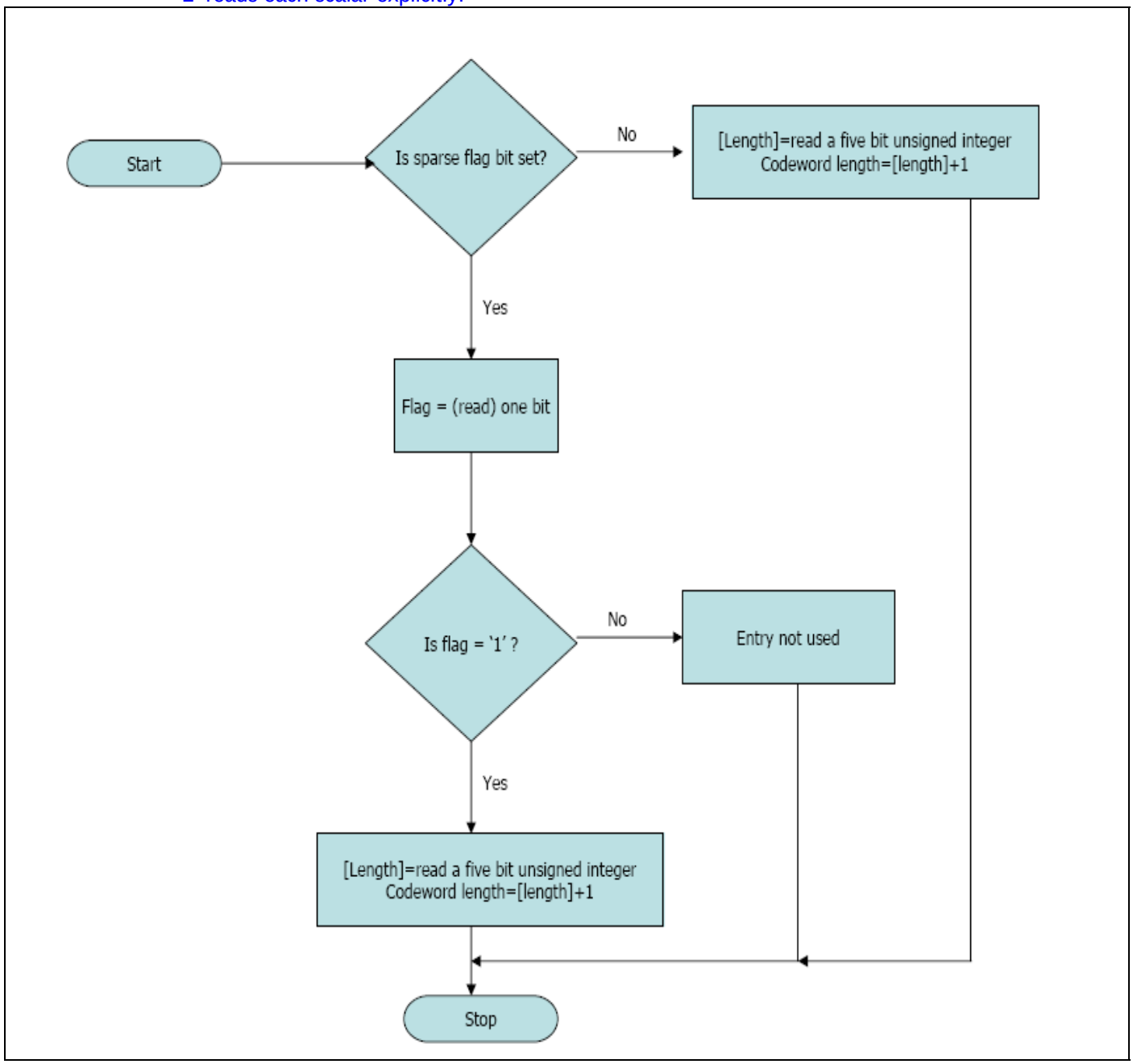


Figure: Flowchart of decoding of each codebook_entry

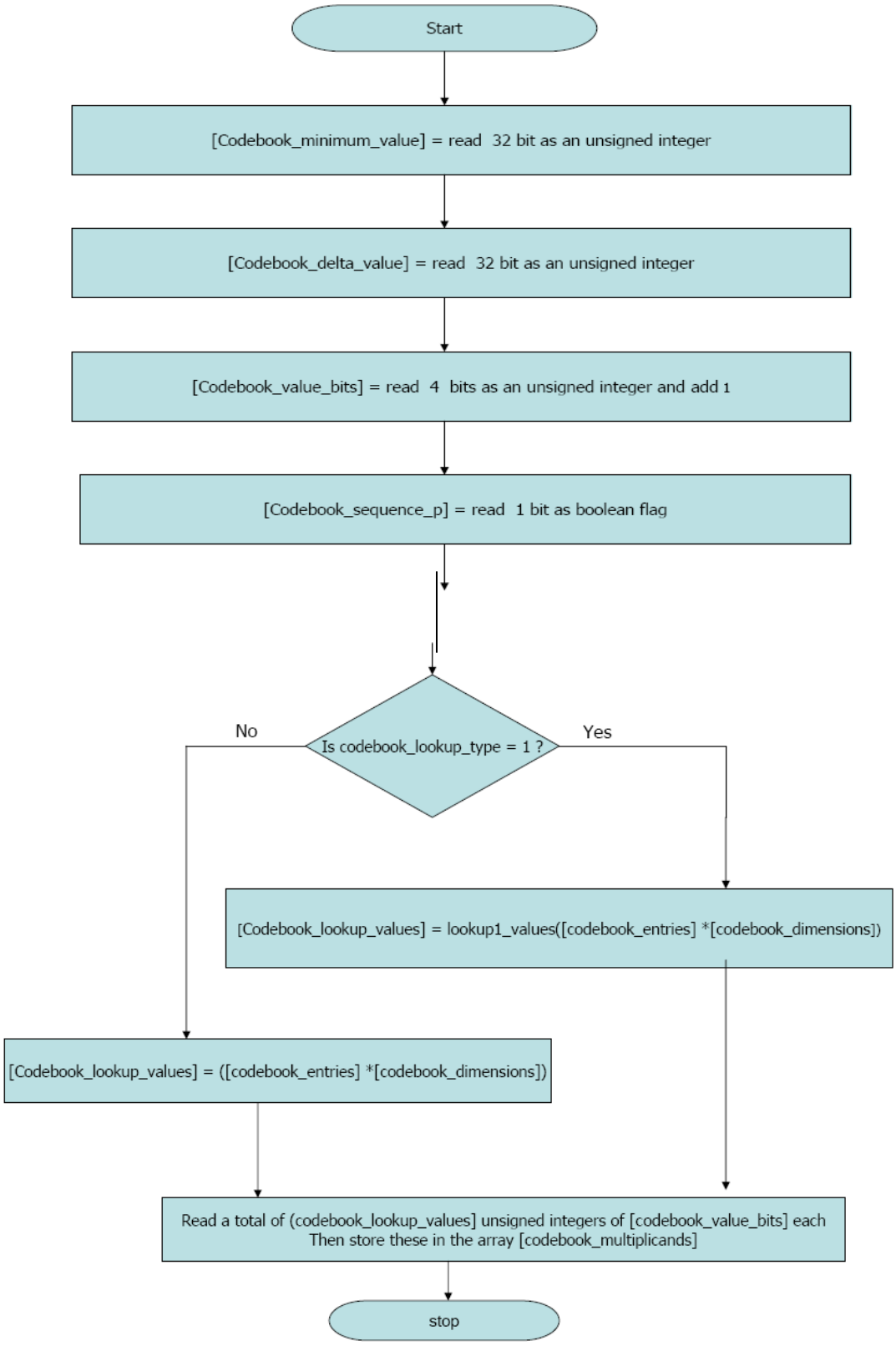


Figure: flowchart of the lookup table type.

Huffman Decision tree representation

- The [codebook_codeword_lengths] array and [codebook_entries] value uniquely define the Huffman decision tree used for entropy decoding.

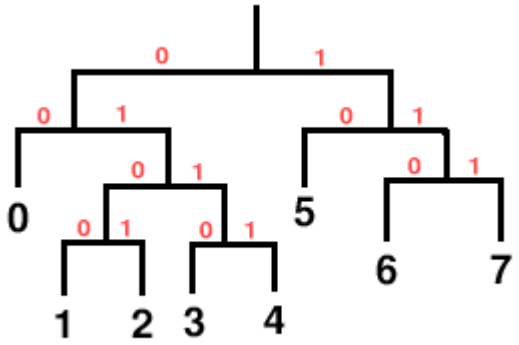


Figure: Huffman Decision tree representation.

- Both underspecified and over specified trees are an error condition rendering the stream undecodable.

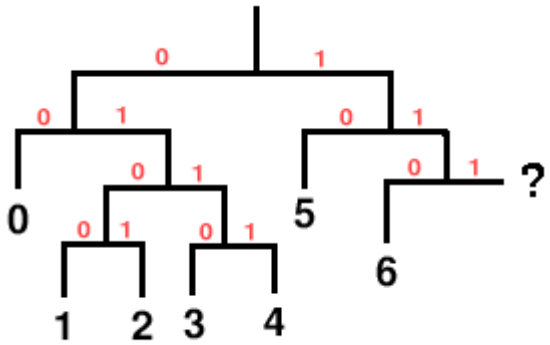


Figure: Underspecified Huffman Decision tree representation.

VQ lookup table vector representation

- Lookup table I specifies a lattice VQ table built algorithmically from a list of scalar values.
- Lookup table II specifies a VQ lookup table in which each scalar in each vector is explicitly set by [codebook_multiplicands] array in a one-to-one mapping.

Decoder uses Codebook abstraction, a specific codeword reads a codeword from a bitstream, decodes it into an entry no. which is used as return value in scalar context and as an offset in VQ context.

Codec Setup and Packet Decode

Header Decode and Decode Setup

- Three header packets: Identification, comment and setup headers.

- An end-of-packet error condition should not occur during decoding of 1st and 3rd packets.
- Decode continues according to packet type; the identification header is type 1, the comment header type 3 and the setup header type 5 (these types are all odd as a packet with a leading single bit of '0' is an audio packet). The packets must occur in the order of identification, comment, and setup.

Identification Header

The identification header is coded as follows:

1. [vorbis_version] = read 32 bits as unsigned integer
2. [audio_channels] = read 8 bit integer as unsigned
3. [audio_sample_rate] = read 32 bits as unsigned integer
4. [bitrate_maximum] = read 32 bits as signed integer
5. [bitrate_nominal] = read 32 bits as signed integer
6. [bitrate_minimum] = read 32 bits as signed integer
7. [blocksize_0] = 2 exponents (read 4 bits as unsigned integer)
8. [blocksize_1] = 2 exponents (read 4 bits as unsigned integer)
9. [framing_flag] = read one bit

Comment Header

The Comment Header is logically a list of eight-bit clean vectors and a single vector for vendor name. The comment header is decoded as follows:

1. [vendor_length] = read an unsigned integer of 32 bits
2. [vendor_string] = read a UTF-8 vector as [vendor_length] octets
3. [user_comment_list_length] = read an unsigned integer of 32 bits
4. Iterate [user_comment_list_length] times {
5. [length] = read an unsigned integer of 32 bits
6. This iteration's user comment = read a UTF-8 vector as [length] octets}
7. [framing_bit] = read a single bit as Boolean
8. If ([framing_bit] unset or end-of-packet) then ERROR
9. Done.

- Comment field line consists of a field name and a corresponding value.
- Field names are as follows: Title, Version, Album, and Artist, Track number, Performer, Copyright, License, Date, Genre, etc.

The comment header is encoded as follows (as per Ogg's standard bitstream mapping which renders least-significant-bit of the word to be coded into the least significant available bit of the current bitstream octet first):

1. Vendor string length (32 bit unsigned quantity specifying number of octets)
2. Vendor string ([vendor string length] octets coded from beginning of string to end of string, not null terminated)
3. Number of comment fields (32 bit unsigned quantity specifying number of fields)
4. Comment field 0 length (if [Number of comment fields]>0; 32 bit unsigned quantity specifying number of octets)
5. Comment field 0 ([Comment field 0 length] octets coded from beginning of string to end of string, not null terminated)

6. Comment field 1 length (if [Number of comment fields]>1…)…

Setup Header

The setup header has the bulk of codec setup information needed for decoding purpose.

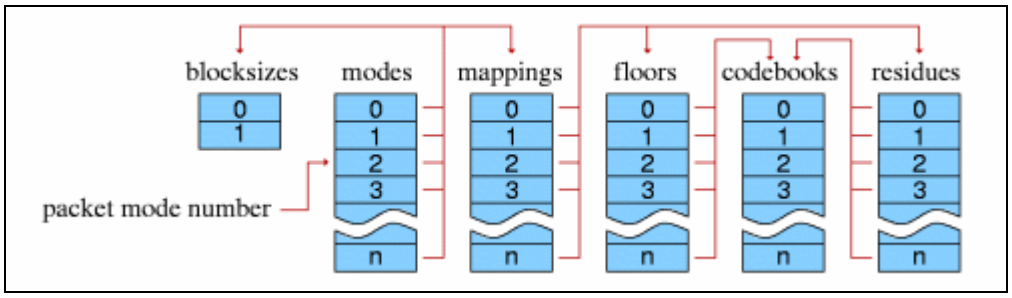


Figure: Setup Header

The setup header contains in order the following:

- Codebook configurations.
- Time-domain transform configurations.
- Floor configurations.
- Residue configurations.
- Channel mapping configurations.
- Mode configurations.

The setup header finishes with a framing bit of '1'.

Audio Packet decode and synthesis

All the packets following the header packets are the audio packets. First step is to read and then verify packet type, and then decode. A non-audio packet is ignored.

The steps involved in audio decoding are as follows:

1. Packet type, mode and window decode.
2. Floor curve decode.
3. Non-zero vector propagate.
4. Residue decoding.
5. Inverse coupling.
6. Dot product of floor and residue vectors.
7. Do inverse MDCT (modified Discrete Cosine Transform).
8. Do overlap and add- windowed MDCT output is overlapped and added with right hand data of previous window.

Specify Output Channel order- Vorbis I specifies only a channel mapping type 0 (now).

Ogg Vorbis Decoder Programming Documentation using VorbisFile

Introduction

Programming of Ogg Vorbis Decoder can be done in the following ways.

- Programming with libvorbis.
- Programming with vorbisfile.

Programming with Vorbisfile

- The Vorbisfile library provides a convenient high-level API for decoding and basic manipulation of all Vorbis I audio streams. Libvorbisfile is implemented as a layer on top of Xiph.org's reference libogg and libvorbis libraries.
- Vorbisfile can be used along with any ANSI compliant stdio implementation for file/stream access, or use custom stream I/O routines provided by the embedded environment.
- The makeup of the Vorbisfile libvorbisfile library API is relatively simple. It revolves around a single file resource. This file resource is passed to libvorbisfile, where it is opened, manipulated, and closed, in the form of an OggVorbis_File struct.

The Vorbisfile API consists of the following functional categories:

- Base data structures
- Setup/Teardown
- Decoding
- Seeking
- File Information

Base data structures

There are several data structures used to hold file and bitstream information during libvorbisfile decoding.
These structures are declared in ‘vorbis/vorbisfile.h’ and ‘vorbis/codec.h’.

data type	purpose
OggVorbis_File	This structure represents the basic file information. It contains a pointer to the physical file or bitstream and information about that bitstream.
vorbis_comment	This structure contains the file comments. It contains a pointer to unlimited user comments, information about the number of comments, and a vendor description.
vorbis_info	This structure contains encoder-related information about the bitstream. It includes encoder info, channel info, and bit rate limits.

Table: Data type structures

Setup/ Tear down

- A bitstream must be properly initialized and cleared while decoding.
- Use fopen() to open Vorbis file, pass file* (pointer) to ov_open call.
- Ov_clear() is used to close the file and deallocate decoding resources. Fclose() not required.

- All libvorbisfile initialization and deallocation routines are declared in "vorbis/vorbisfile.h".
- Functions in this are ov_open(), ov_open_callbacks(), ov_test(), ov_test_open(), ov_test_callbacks(), ov_clear().

Decoding

- All libvorbisfile decoding routines are declared in "vorbis/vorbisfile.h".
- After initializing, decoding audio is done by calling ov_read() , which works to read().

Ov_read () does the following function:

- Multiple stream links - A Vorbis stream may consist of multiple sections (called links) that encode differing numbers of channels or sample rates.
- Returned data amount – ov_read guarantees that the passed back data does not overflow the passed in buffer size. Large buffers may be filled by iteratively looping over calls to ov_read (incrementing the buffer pointer) until the original buffer is filled.
- File cursor position–Vorbis may start at any sample number, and the offset need no be 0.

function	purpose
ov_read	This function makes up the main chunk of a decode loop. It takes an OggVorbis_File structure, which must have been initialized by a previous call to ov_open().
ov_read_float	This function decodes to floats instead of integer samples.

Seeking

- Seeking functions allows specifying a specific point in the stream to begin or continue decoding.
- All libvorbisfile seeking routines are declared in "vorbis/vorbisfile.h".
- Seeking is available only within a seekable file or stream. Seeking functions will return OV_ENOSEEK on nonseekable files and streams.
- The functions declared are as follows: ov_raw_seek(), ov_pcm_seek(), ov_time_seek(), ov_time_seek_page(), ov_raw_seek_page, ov_pcm_seek_page(), etc,.

File Information

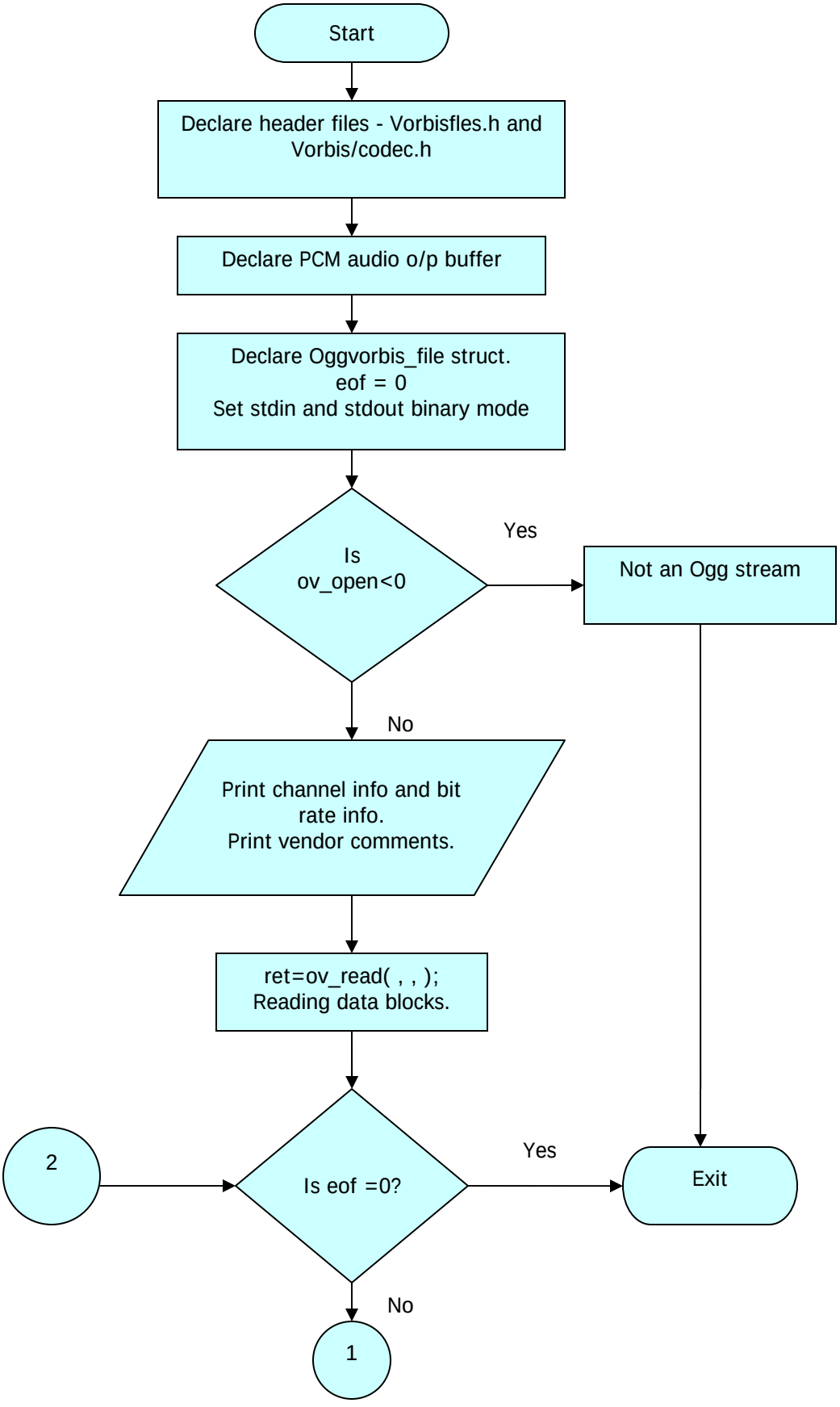
- Libvorbisfile contains many functions to get information about bitstream attributes and decoding status.
- All libvorbisfile file information routines are declared in "vorbis/vorbisfile.h".
- The functions under this are: ov_bitrate(), ov_info, ov_comment(), ov_stream(), ov_serialnumber(), ov_seekable, etc.

OggVorbis_File

- It is declared in "vorbis/vorbisfile.h"
- The OggVorbis_File structure defines an Ogg Vorbis file.
- This structure is used in all libvorbisfile routines.

- It is initialized by ov_open() or ov_open_callbacks(), and closed by ov_clear() and should not be used by applications or functions outside libvorbis API.

```
typedef struct {
  void          *datasource; /* Pointer to a FILE *, etc. */
  int           seekable;
  ogg_int64_t     offset;
  ogg_int64_t     end;
  ogg_sync_state   oy;

  /* If the FILE handle isn't seekable (eg, a pipe), only the current
     stream appears */
  int           links;
  ogg_int64_t     *offsets;
  ogg_int64_t     *dataoffsets;
  long          *serialnos;
  ogg_int64_t     *pcmlengths;
  vorbis_info     *vi;
  vorbis_comment   *vc;

  /* Decoding working state local storage */
  ogg_int64_t     pcm_offset;
  int           ready_state;
  long          current_serialno;
  int           current_link;

  ogg_int64_t     bittrack;
  ogg_int64_t     samptrack;

  ogg_stream_state os; /* take physical pages, weld into a logical
                          stream of packets */
  vorbis_dsp_state vd; /* central working state for the packet->PCM decoder */
  vorbis_block     vb; /* local working space for packet->PCM decode */

  ov_callbacks callbacks;

} OggVorbis_File;
```


Start
Declare header files - Vorbisfles.h and Vorbis/codec.h
Declare PCM audio o/p buffer
Declare Oggvorbis_file struct.
eof = 0
Set stdin and stdout binary mode
Is ov_open<0
Yes
Not an Ogg stream
No
Print channel info and bit rate info.
Print vendor comments.
ret=ov_read(, ,);
Reading data blocks.
2
Is eof =0?
Yes
Exit
No
1

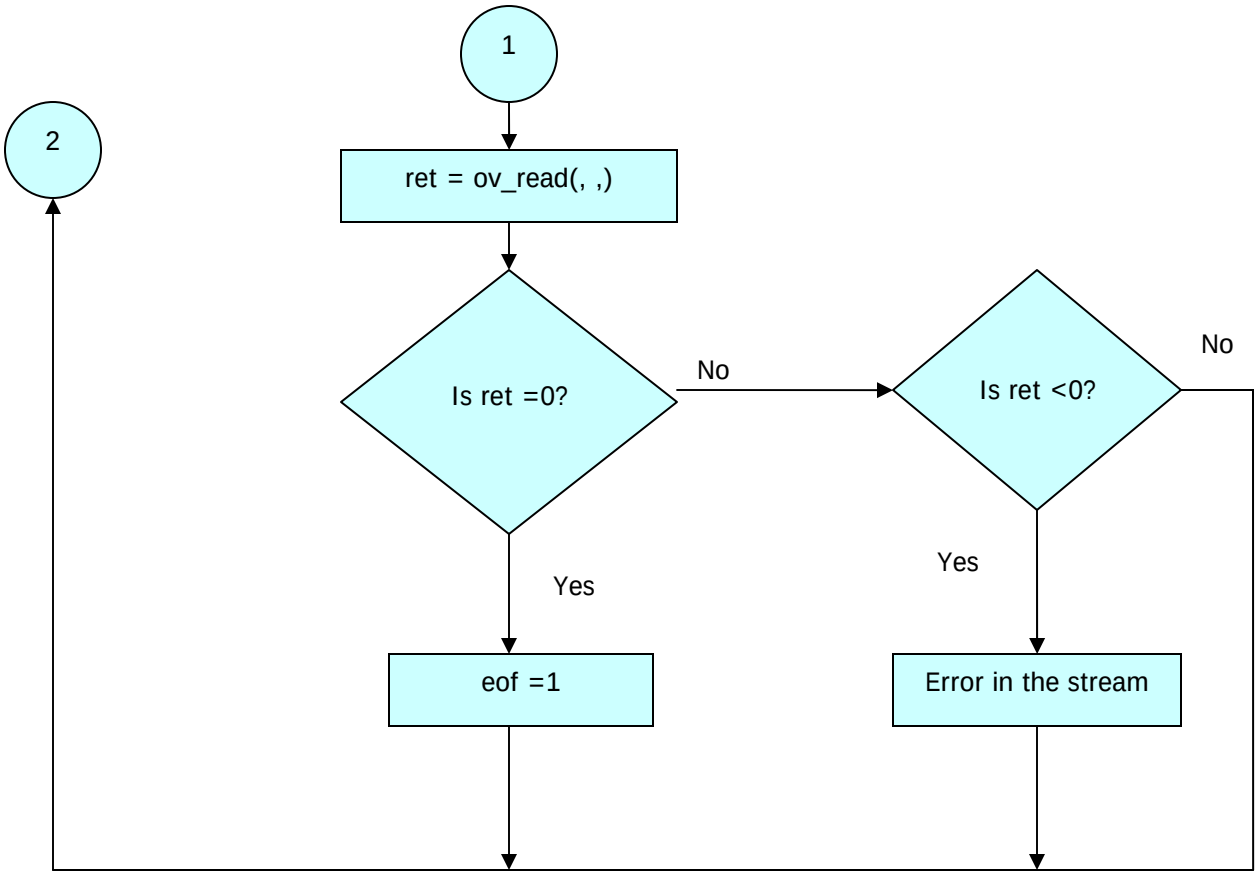
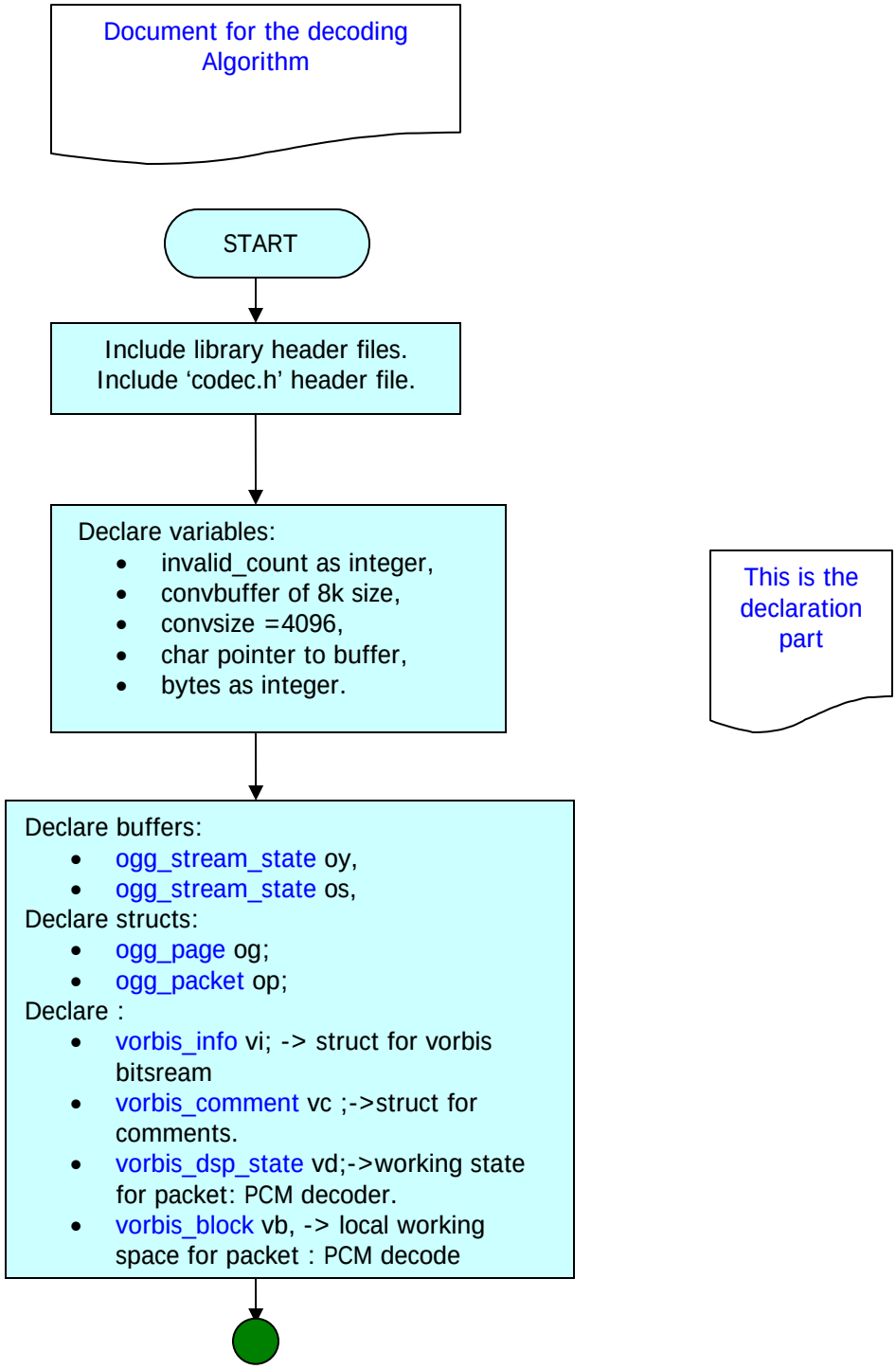
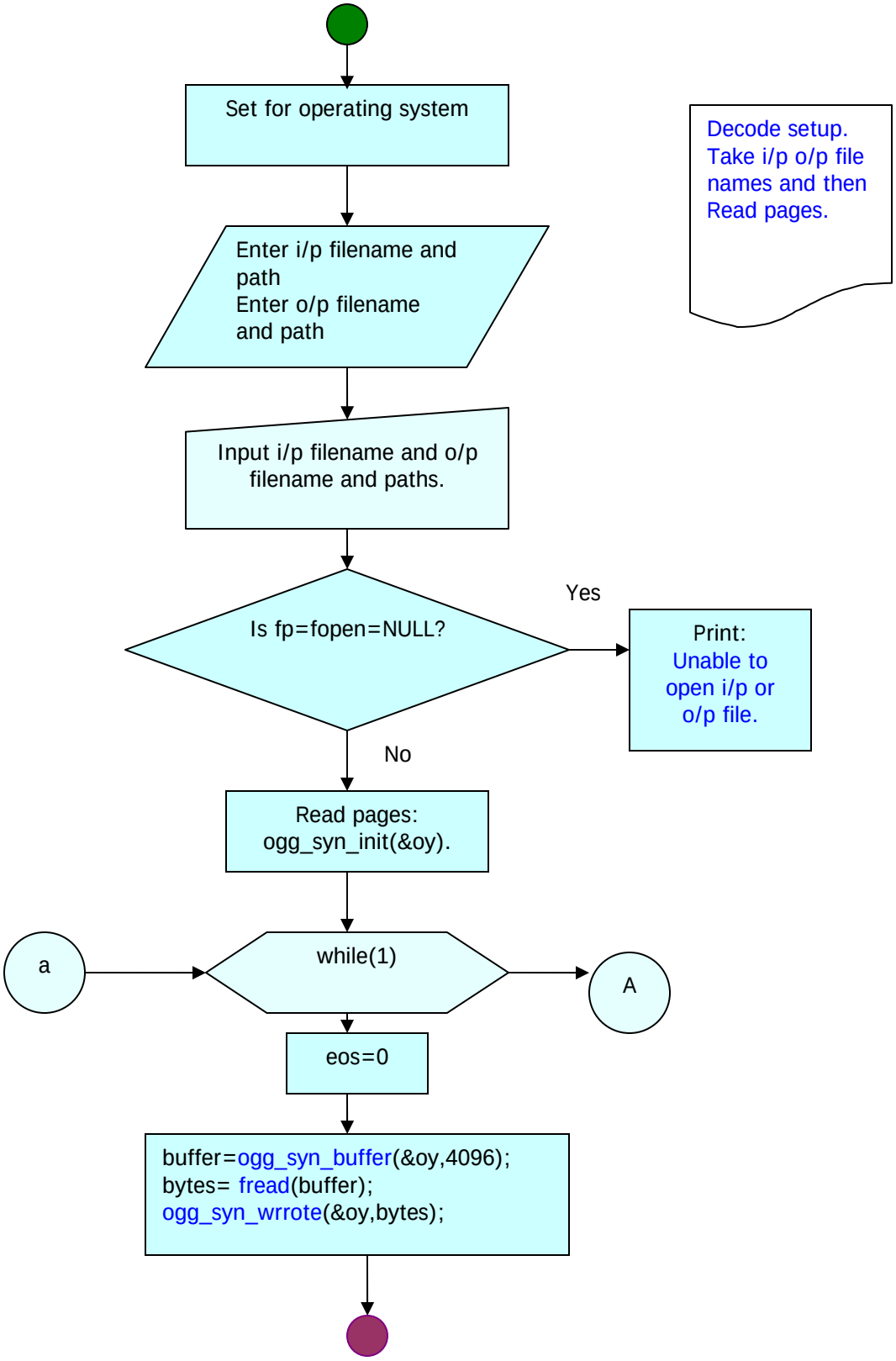


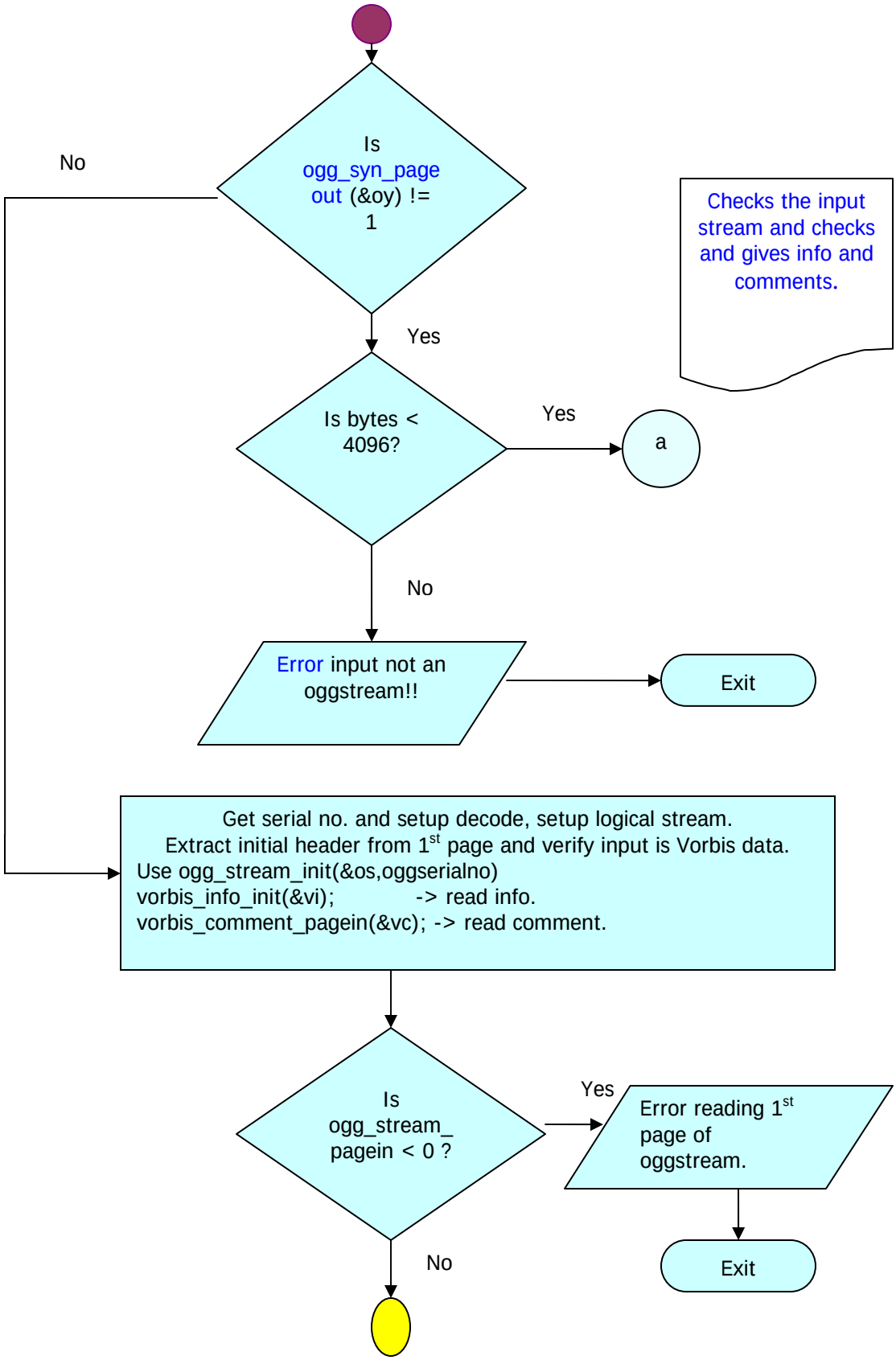
Figure: Flowchart of the vorbisfile decoding

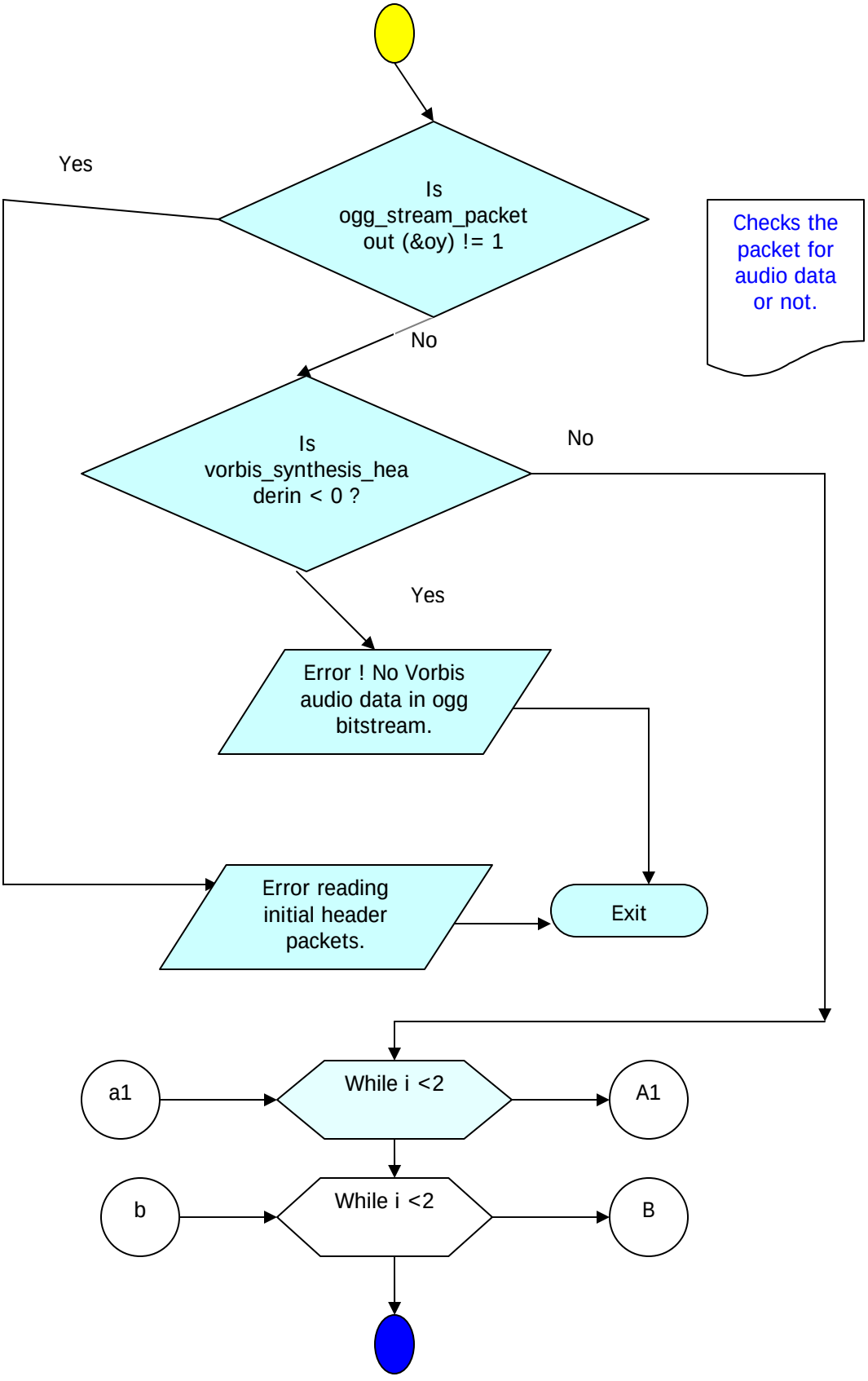
Decoder Flowchart

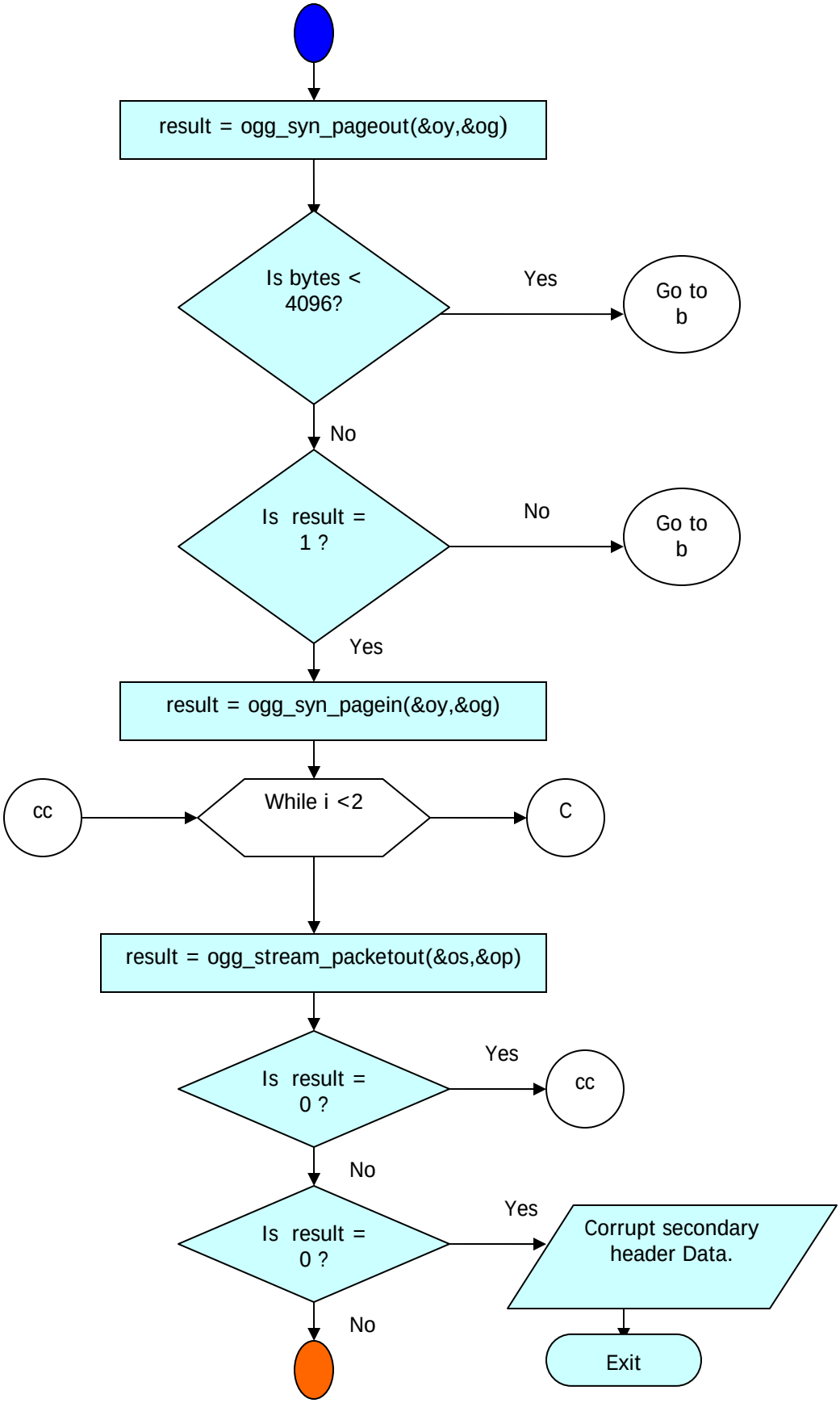
This document gives the flowchart of decoding algorithm derived from the decoder_example.c file.

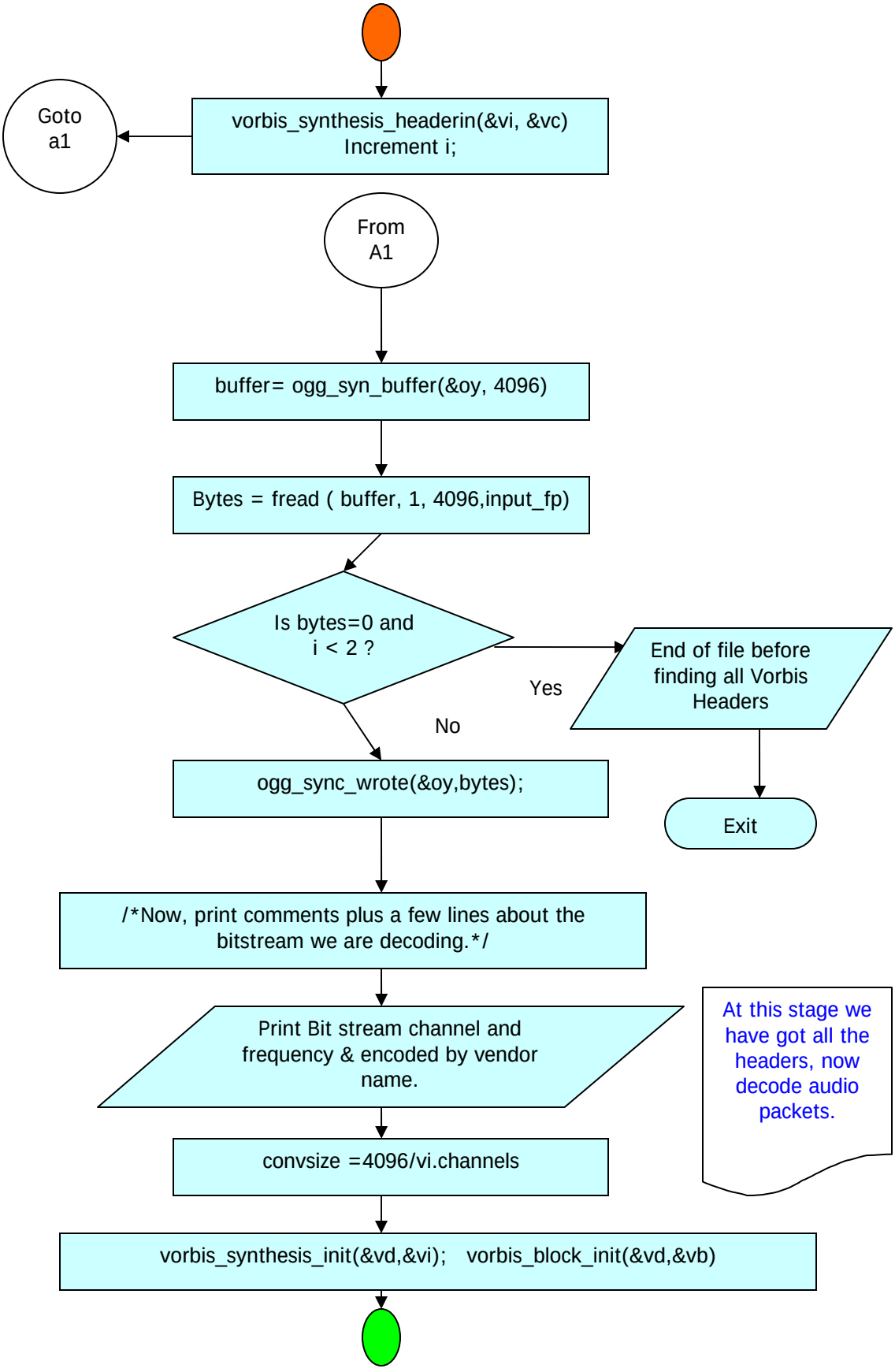

Document for the decoding Algorithm
START
Include library header files.
Include 'codec.h' header file.
Declare variables:
invalid_count as integer,
convbuffer of 8k size,
convsize =4096,
char pointer to buffer,
bytes as integer.
This is the declaration part
Declare buffers:
ogg_stream_state oy,
ogg_stream_state os,
Declare structs:
ogg_page og;
ogg_packet op;
Declare :
vorbis_info vi; -> struct for vorbis bitsream
vorbis_comment vc ;->struct for comments.
vorbis_dsp_state vd;->working state for packet: PCM decoder.
vorbis_block vb, -> local working space for packet : PCM decode

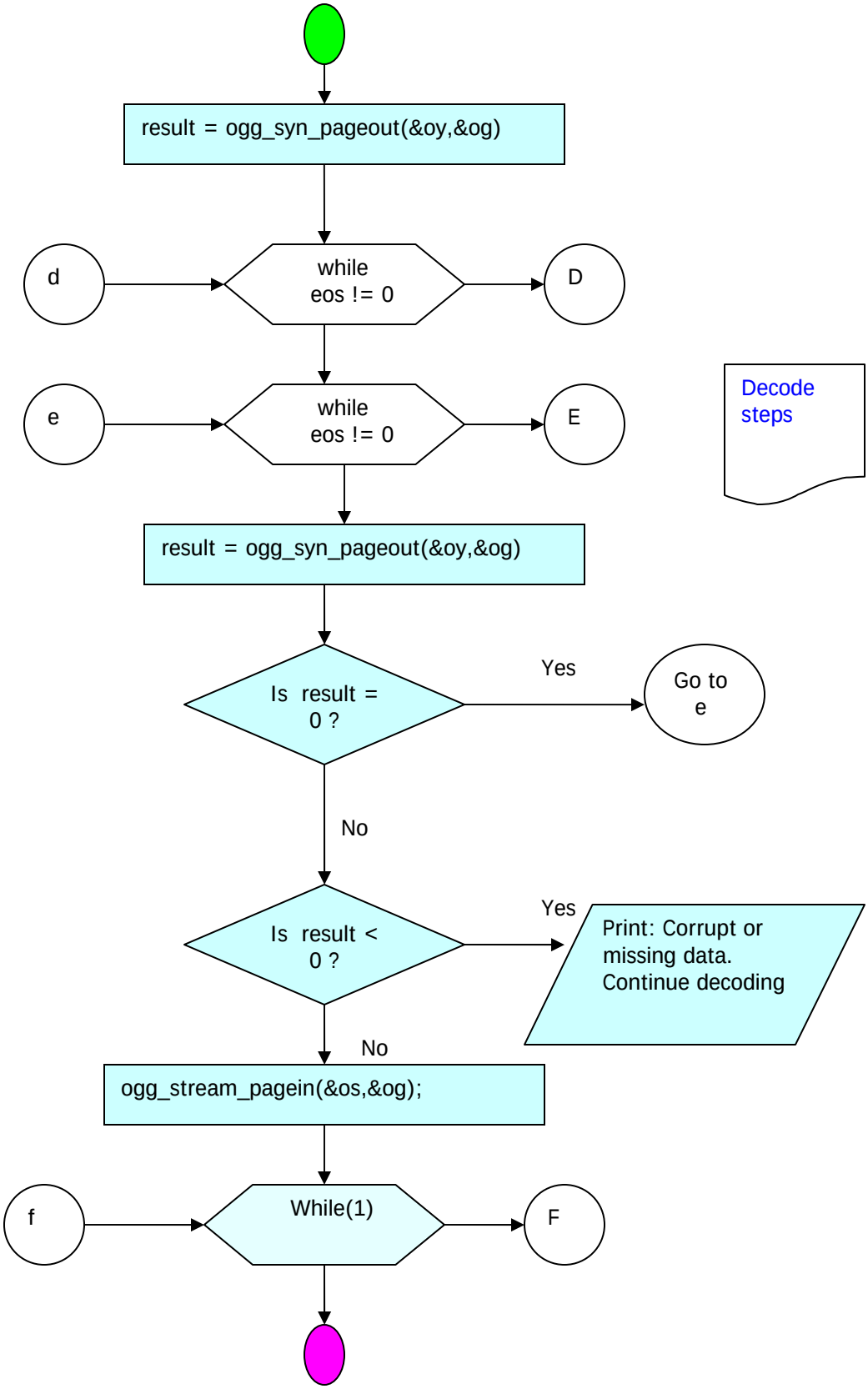

Set for operating system
Decode setup.
Take i/p o/p file
names and then
Read pages.
Enter i/p filename and
path
Enter o/p filename
and path
Input i/p filename and o/p
filename and paths.
Is fp=fopen=NULL?
Yes
Print:
Unable to
open i/p or
o/p file.
No
Read pages:
ogg_syn_init(&oy).
a
while(1)
A
eos=0
buffer=ogg_syn_buffer(&oy,4096);
bytes= fread(buffer);
ogg_syn_wrrote(&oy,bytes);

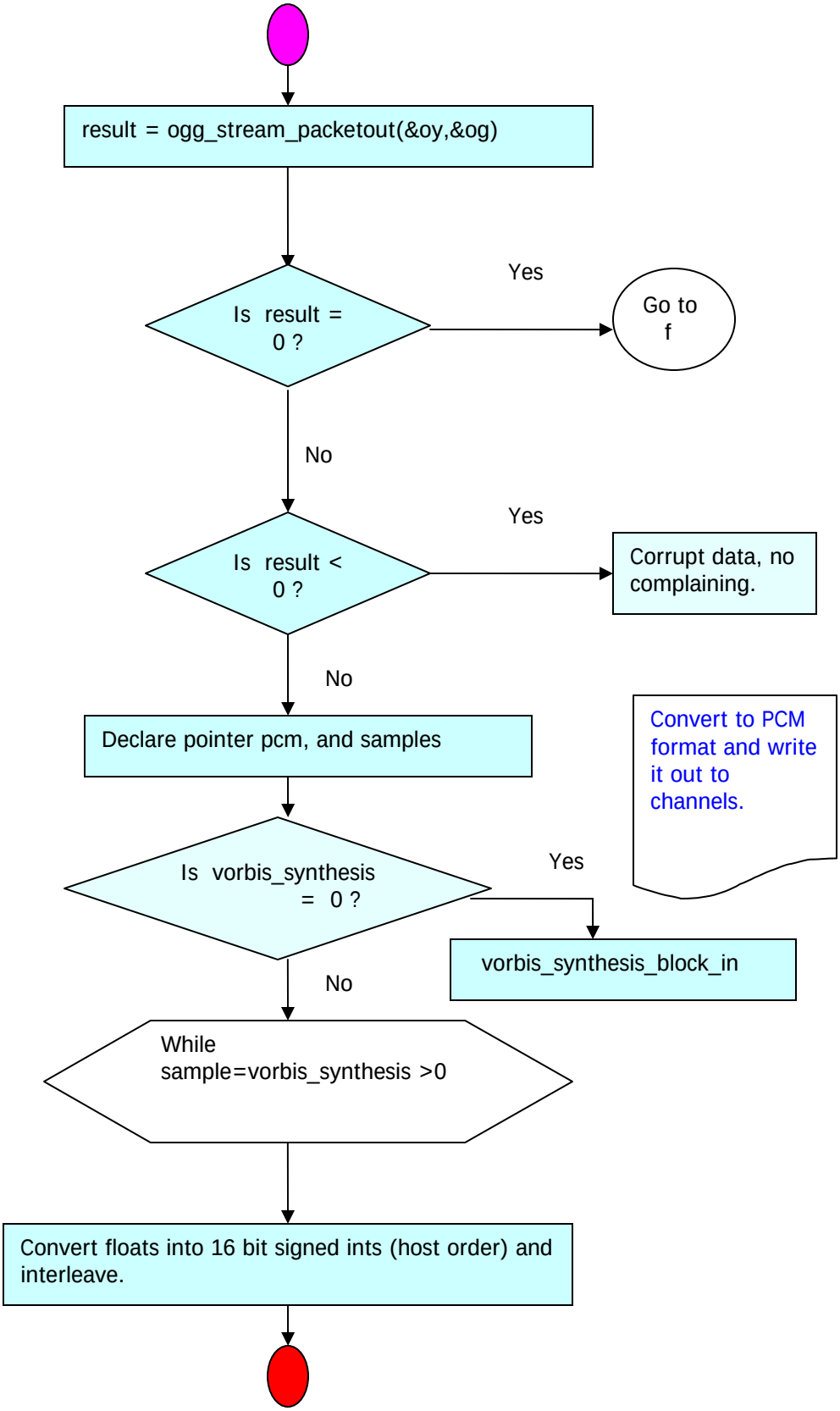

Is ogg_syn_page out (&oy) != 1
No
Yes
Checks the input stream and checks and gives info and comments.
Is bytes < 4096?
Yes
a
No
Error input not an oggstream!!
Exit
Get serial no. and setup decode, setup logical stream.
Extract initial header from 1st page and verify input is Vorbis data.
Use ogg_stream_init(&os,oggserialno)
vorbis_info_init(&vi); -> read info.
vorbis_comment_pagein(&vc); -> read comment.
Is ogg_stream_ pagein < 0 ?
Yes
Error reading 1st page of oggstream.
Exit
No

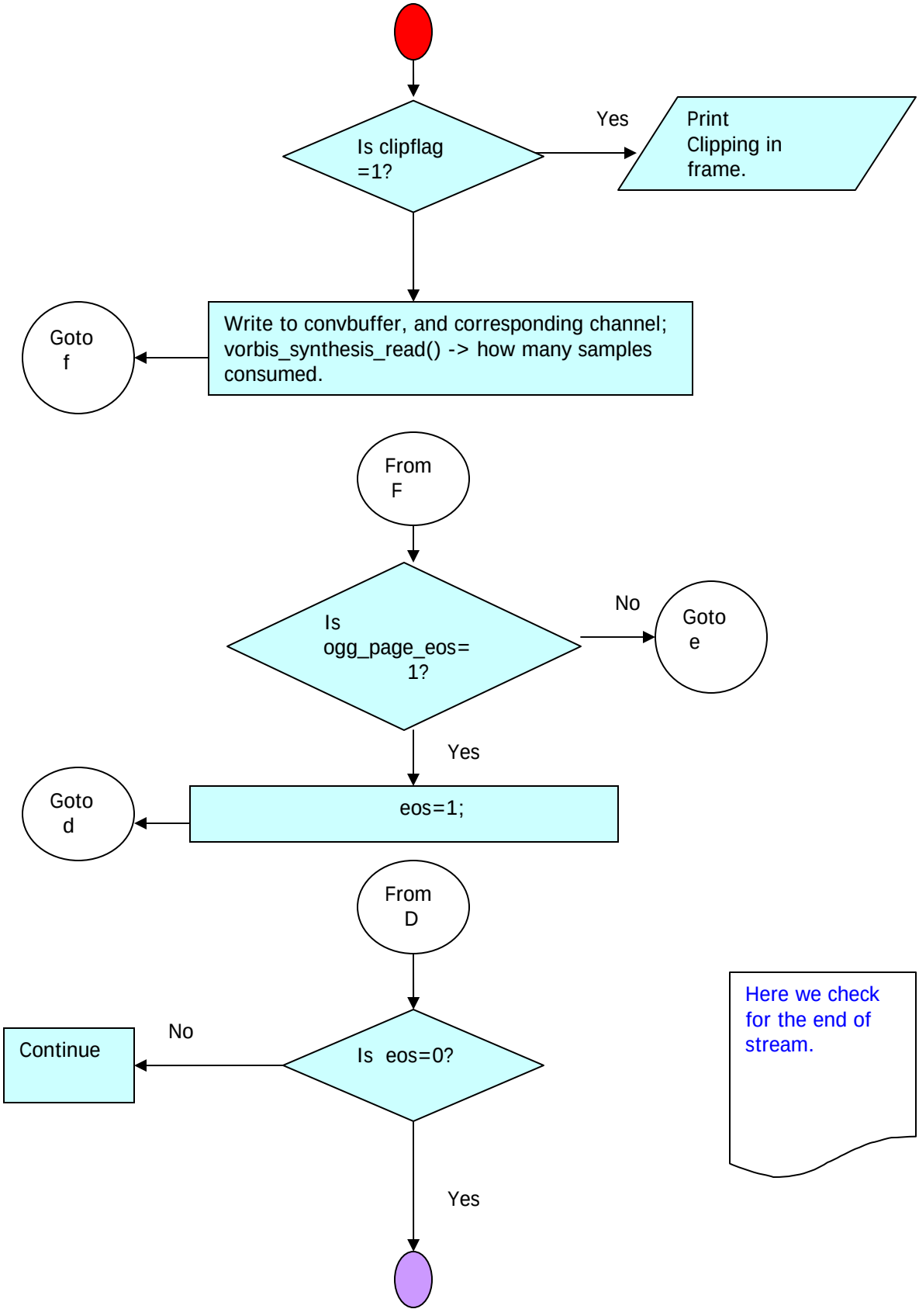

Yes
Is
ogg_stream_packet
out (&oy) != 1
Checks the
packet for
audio data
or not.
No
Is
vorbis_synthesis_hea
derin < 0 ?
No
Yes
Error ! No Vorbis
audio data in ogg
bitstream.
Error reading
initial header
packets.
Exit
a1
While i <2
A1
b
While i <2
B

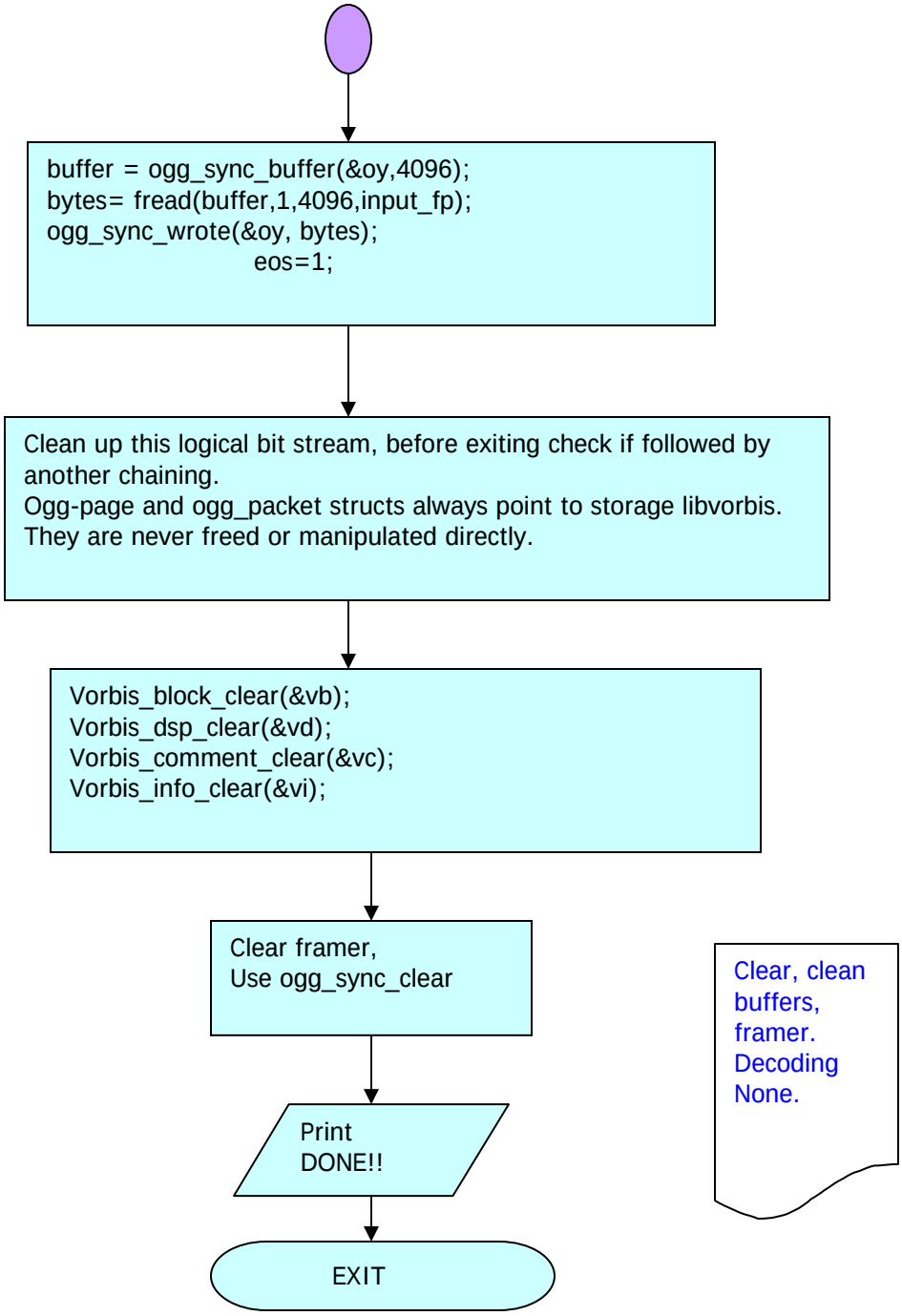

result = ogg_syn_pageout(&oy,&og)
Is bytes < 4096?
Yes
Go to b
No
Is result = 1 ?
No
Go to b
Yes
result = ogg_syn_pagein(&oy,&og)
cc
While i <2
C
result = ogg_stream_packetout(&os,&op)
Is result = 0 ?
Yes
cc
No
Is result = 0 ?
Yes
Corrupt secondary header Data.
No
Exit


Goto
a1
vorbis_synthesis_headerin(&vi, &vc)
Increment i;
From
A1
buffer= ogg_syn_buffer(&oy, 4096)
Bytes = fread (buffer, 1, 4096,input_fp)
Is bytes=0 and
i < 2 ?
Yes
End of file before
finding all Vorbis
Headers
No
Exit
ogg_sync_wrote(&oy,bytes);
/*Now, print comments plus a few lines about the
bitstream we are decoding.*/
Print Bit stream channel and
frequency & encoded by vendor
name.
At this stage we
have got all the
headers, now
decode audio
packets.
convsize =4096/vi.channels
vorbis_synthesis_init(&vd,&vi); vorbis_block_init(&vd,&vb)


result = ogg_syn_pageout(&oy,&og)
d
while
eos != 0
D
e
while
eos != 0
E
Decode
steps
result = ogg_syn_pageout(&oy,&og)
Is result =
0 ?
Yes
Go to
e
No
Is result <
0 ?
Yes
Print: Corrupt or
missing data.
Continue decoding
No
ogg_stream_pagein(&os,&og);
f
While(1)
F


result = ogg_stream_packetout(&oy,&og)
Is result = 0 ?
Yes
Go to f
No
Is result < 0 ?
Yes
Corrupt data, no complaining.
No
Declare pointer pcm, and samples
Convert to PCM format and write it out to channels.
Is vorbis_synthesis = 0 ?
Yes
vorbis_synthesis_block_in
No
While sample=vorbis_synthesis >0
Convert floats into 16 bit signed ints (host order) and interleave.


Is clipflag =1?
Yes
Print Clipping in frame.
Write to convbuffer, and corresponding channel; vorbis_synthesis_read() -> how many samples consumed.
Goto f
From F
Is ogg_page_eos= 1?
No
Goto e
Yes
eos=1;
Goto d
From D
Is eos=0?
No
Continue
Yes
Here we check for the end of stream.



Conclusion

Thus, a comprehensive study of the ogg vorbis audio scheme was carried out and flowcharts were designed to carry out the decoding process. We started with Ogg specification- described the ogg bitstream and its classification, page formats and demultiplexing and decoding algorithm. We then moved over to Vorbis I specification. Here, we gave an overview of the Vorbis I

encoder/decoder. We gave a comprehensive overview of the decoder, describing the high-level decode process with explanation, followed by the bit-packing convention.
We then moved over to codec setup procedure where our focus was on packet decoding. Finally, we described the ogg vorbis decoding using vorbisfile, and supplemented it with appropriate flow charts to support our explanation.