Ogg Vorbis: Bytes To Beeps

Abstract

This document details how to decode the audio codec ogg vorbis using the `libvorbis` library. This information is not included in the ogg vorbis distribution. The approach taken is to describe the high level environment and then to present and walk through a sample decoder.

The sample decoder is divided into two sections - a simple decoder client and a reusable core. The reusable core is a source code library and is referred in this document as `libvorbissimple` . It is provided as a source code library since it is expected that as well as reuse, customisation will be needed for some applications.

As far as can be determined, this is the most detailed document on how to build a vorbis decoder using `libvorbis` directly.

Michael Sparks, michaels@rd.bbc.co.uk, July 2004

1 Introduction

Ogg Vorbis is the common name of an open source/open standard audio codec. Ogg is a wrapper format (akin to MXF, etc.). Vorbis is the audio codec.

Vorbis is a fairly good quality codec for relatively low bit rate audio with quality sufficient for Internet streaming. In addition to this, ogg vorbis does not have any associated licensing costs. This also makes it attractive for streaming.

There are 2 main libraries for decoding vorbis data - `libvorbis` and `libvorbisfile`. The former offers greater flexibility, but is not documented. The latter offers significantly less flexibility - and causes limitations if you need to keep control of the flow of control during decode.

1.1 About this document

The scope of this document is to detail how one can write a simple decoder using the primary implementation `libvorbis`. There is a need for this since there is no documentation currently exists on this topic. The decoder presented is designed to allow code reuse and is separated into a decode stub and a simple source code library. For sake of discussion this library has been termed `libvorbissimple`.

This document is the result of the need to use a simple decoder which allows the following decode idiom:

- While decoding and no errors:
    - Ask for audio - to which we get a response
        - If we're asked for data, respond with bytes for decode (received, for example, from a network socket)
        - If we get audio we can do something with it. (e.g. go "beep")

The source code library presented is designed to be used in this manner.

The standard method of decoding ogg vorbis (to use `libvorbisfile`) does not make this approach simple or clear. This is due to an assumption by `libvorbisfile` that file reading is coupled with playback.

Whilst `libvorbisfile` has a well documented API, and is simple to use, unlike `libvorbis`. The sample decoder supplied with `libvorbis` uses many undocumented calls into `libvorbis`. It does not make clear the steps required for using `libvorbis` directly. The structure used by the supplied decoder takes some unusual design decisions which leads to an essentially impenetrable design.

As a result, for all intents and purposes the supplied sample decoder leaves ogg vorbis essentially undocumented. This *appears* to have been a concious decision with the aim of encouraging people to use `libvorbisfile` . Unfortunately `libvorbisfile` is not suitable in all cases.

1.2 Document Organisation

The remainder of this document is structured as follows:

- Section 2 provides a basic overview of the Ogg Vorbis decode chain, listing the issues, and responsibilities involved in creating an ogg vorbis decoder.
- Section 3 provides a conceptual overview of the steps a user must go through in order to use the presented `libvorbissimple` source code library.
- Section 4 presents a sample decoder using `libvorbissimple` . The approach taken is to discuss the code that is being presented, what each section does and why. Since this approach does not suit all readers, the full sample decoder using `libvorbissimple` is presented at the end of section 4.
- Section 5 discusses the structure of `libvorbissimple` . The approach taken here is to follow the usage of the source code library. The library itself uses a layered design.
- Section 6 wraps up the document
- 2 Appendices are included – one is a graphical cross reference of layers/functions, the other is a summary of `libvorbis`.

1.3 What is a Source Code Library?

Source code libraries are rarely used these days, so this is a very brief description reminding us what they are and why `libvorbissimple` was chosen to be one.

Modern static libraries or dynamic libraries are designed to be used via static binding by the C linker or by dynamic. The reason is this allows unmodified reuse of code. Source code libraries are a very old idea that is rarely used these days since it generally results in code duplication.

The idea is simple – it is code that you include into your source code directly – either by manual code copying or by use of an include directive using the preprocessor. It's not an approach you would normally take for a piece of production code, but for a tutorial/documentation purposes the separation of responsibility is useful.

The reason it was chosen here for `libvorbissimple` is because this document discusses how to write a vorbis decoder using `libvorbis`. Changing the `libvorbissimple` to a separate library would detract from this aim.

Clearly there is merit to changing from a source code library to a modern library style in terms of *actual use and reuse* of the code in `libvorbissimple`. In the context of simply describing how to go from data in a file to audio, it was viewed that the simpler description in terms of "this part is reusable" versus "this is general high level things" was a more useful approach.

The actual changes to make this work as a standalone static or dynamic library are *relatively* trivial (split the .c file into .c and .h files, change build and link process), but would complicate the description presented here.

1.4 Caveat

Two points worth noting:

- Details presented on what the `libvorbis` library calls do is based on deciphering undocumented `libvorbis` code (both internally and externally).
- The details on steps needed to decode ogg vorbis files have been deciphered by picking apart the sample code and examining it to reveal the intent behind the structure. This process was made harder by the unusual implementation style.

As a result, some of the information outlined may be incomplete or worse, inaccurate.

2 An Overview of the Ogg Vorbis Decode Chain

Ogg is a wrapper format, and vorbis is the codec. A sequence of ogg pages may contain a number of packets containing vorbis encoded audio. In order to decode ogg vorbis the follow high level steps have to occur:

- The system needs to take whole ogg pages out of the data stream - this forms a sequence of ogg pages
- These then get taken and integrated into packets
- The decode then takes the data out of the packets, decodes and spits out buffers containing PCM audio.

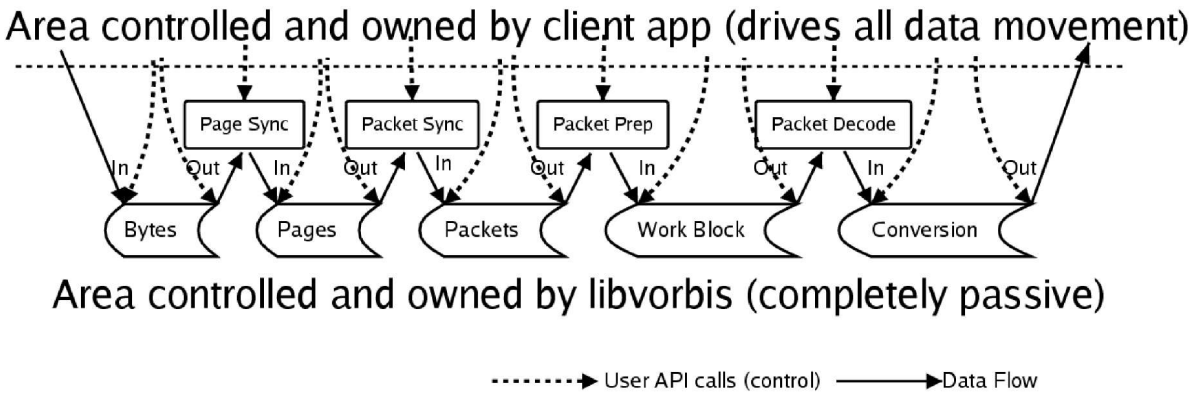
As a result, writing an ogg vorbis decoder requires some understanding of the ogg decapsulation chain. This is in addition to priming the actual decode chain. The reason for this when using `libvorbis` directly *you* are responsible for making this full decode chain operate.

You do not just put wrapped encoded bytes in one end and have bytes to come out the other side. You put all of the components of the chain together, put bytes in one end and repeatedly prod all of the intermediate points in the chain to get bytes come out the other end.

In order to decode vorbis a number of processing steps occur:

- The user application reads some raw data from a data source - such as an ogg vorbis file.
- This application passes these raw bytes to the ogg library via the means of a shared buffer
- The ogg library takes bytes out of the shared buffer in the form of ogg pages, and places these in a shared buffer called a stream
- The vorbis library then takes packets out of the shared stream buffer and places this in a working buffer
- The vorbis library then takes the packets out of the working buffer decodes then and places floating point PCM into a conversion buffer. (Ogg vorbis describes this process as synthesis)
- It is then up to the user to convert this to integer PCM (and to effectively free up the buffer)

Essentially the decode chain you are responsible for constructing and managing looks like this:



However, every part of this decode chain needs to be set up and managed by any client of the `libvorbis` library. Each dashed line represents an API call that the user must perform in order to:

- Move data out of buffers
- Move data into buffers
- Tell `libvorbis` to perform some work

When different parts of this pipeline stall due to lack of data:

- The system signals that more data is required
- The user must then provide more data
- The user must then repeatedly prod the various parts of this system, to feed the data through. They do this until they get back to where they were in the hope of getting more data out.

Clearly, this API is intended to be effective when running in a parallel environment. Each processing stage can run in parallel with every other safely. This is due to the processing stages being decoupled by buffers connected by unidirectional pipes.

For the common case, single threaded execution it is much more complex than most users require (especially if undocumented). Furthermore this diagram only indicates the structure required for the main decode loop, rather than the set-up and tear down.

The approach taken by the sample ogg vorbis decoder supplied with `libvorbis` is to wrap each failure point inside a loop. This allows the system to exit the loop upon failure and repeat an earlier step. This means that the bulk of activity that happens inside a state should happen immediately after the nested while loop. Since the sample decoder uses nested while loops this means that decoding headers and decoding audio results in code duplication. Using nested loops hides the inherent nature of the system - that is of independent state machines, each with simple processing steps.

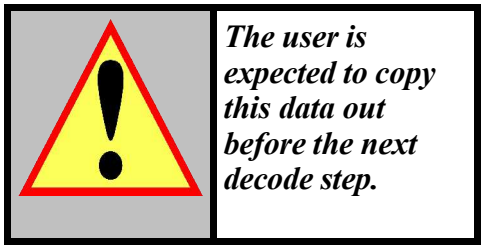
This structure forms a collection of state machines which are prodded from time to time. Note that the user essentially forms a scheduler. This is made more complex by not having been explicitly told which activities they should schedule.

The sample decoder associated with this document simplifies the common case by using a layered model for decode. This decoder presented in section 4 focusses on a "bytes in, beeps out" application. The sample decoder throws C-style exceptions back to the top level as needed, in a similar way to network handling code. This layered model takes charge of this scheduling requirement - moving complexity out of the client side.

2.1 The Ogg Vorbis Decode Lifecycle

When decoding ogg vorbis the following conceptual steps occur:

1. The buffers are initialised. We assume that the page and packet buffers contain sufficient data.
2. The first packet is read and decoded. If this is an Ogg Vorbis header packet, processing continues.
3. The next 2 packets are then read and decoded. These are the codebook and comment headers.
4. After this vorbis packets are repeatedly read and decoded. Vorbis calls this process synthesis. New ogg pages pulled in as required, and more bytes read and pulled in as required
5. These packets are then sent to the decoder which returns the decoded audio in a shared conversion buffer owned by the library.



This continues until the end of the vorbis stream. The common case for this, is reaching the end of the file.

Complexity in this system occurs due to the inherent concurrency in the sub stages.

Observations:

- Since we need these 5 steps to run in parallel with the users code, this forms a simple state machine. We need to periodically relinquish control back to the user and then restart back to where we had reached.
- There is state required throughout the entire decode process that accumulates with time. We could use global variables to keep track of this, but then this would prevent reuse of the code.
- Since many ogg vorbis files only contain one vorbis stream, the decoder presented is optimised to this case.
- A specific goal is reuse. The code presented is suitable for use as a simple to use library, as well as a standalone decoder.
- Any decoder using `libvorbis` directly must make similar choices, and `libvorbissimple` just catches the common cases for reuse.

3 Decoding using `libvorbissimple`

The approach taken is to use a layered model. The internals of this will be dealt with later, but first we'll deal with the API first. `libvorbissimple` exposes a simple interface for users who just wish to decode ogg vorbis data, rather than manipulate state machines in order to decode ogg vorbis.

From a user perspective the design specifically allows this style of coding:

- Allocate a **decode context**
- Loop until done
    - Ask to get audio from the decode context
    - If that succeeds, use the audio
    - Otherwise, if the return status is `NEEDDATA`, we either:
        - Populate a source buffer, send it to the decode context, and continue looping
        - If we have no more data for a source buffer, finish looping
    - If the return status was anything else, we're done.

A sample code fragment for using `libvorbissimple` looks like this:

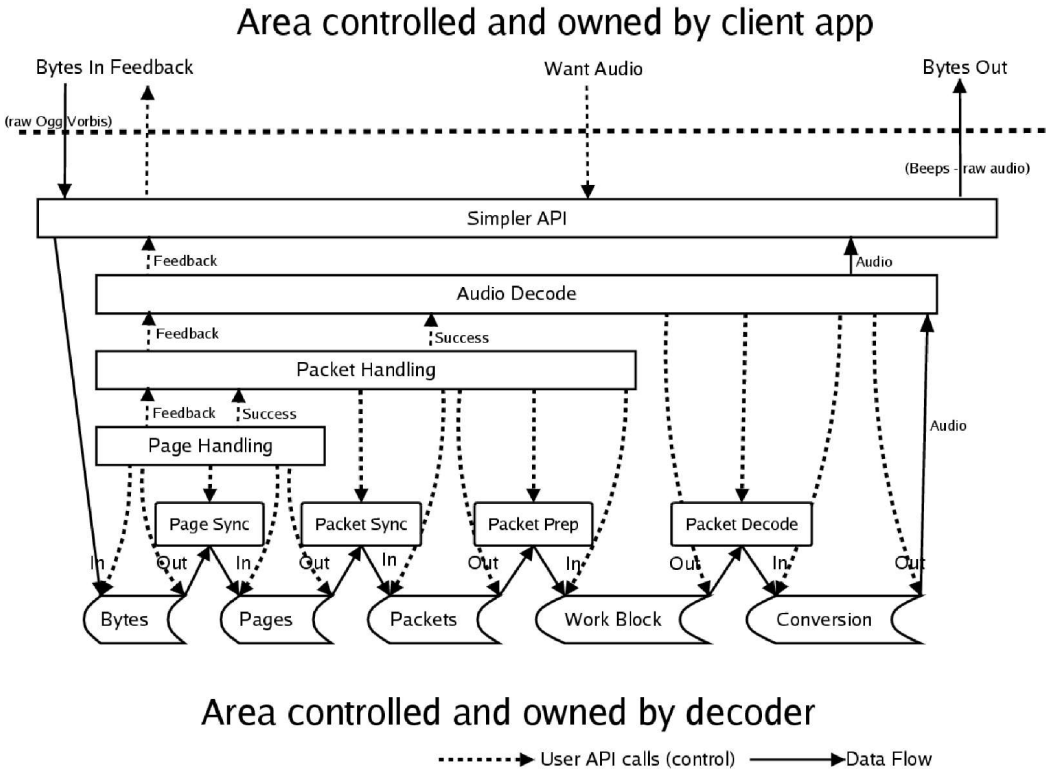
```
oggVorbisContext = newOggVorbisContext();
while(decoding) {
   decodeBuffer = getAudio(oggVorbisContext);
   if (decodeBuffer->status==HAVEDATA) {
   /* Do something with the contents of the decodeBuffer, and
   copy out if required on next loop
    ...
   } else if (decodeBuffer->status==NEEDDATA) {
   /* Acquire some data and fill the sourceBuffer */
      sendBytesForDecode(oggVorbisContext, sourceBuffer);
   } else {  /* Internal decoding error, and/or decode complete */}
}
```

This approach requires the user to:

- Create an `oggVorbisContext` variable (type `ogg_vorbis_context`)
- Create and fill a `sourceBuffer` variable (type `source_buffer`)
- Accept and use values returned to the `decodeBuffer` variable

Internally the top layer implements a state machine for the life cycle described in section 2.1. Each further layer is simple and targeted, handling specific parts of the decode chain. The lower level layers handle things like bytes and pages. The upper layers handle vorbis decoding. If lower layers have any errors these are passed back up the chain.

The following diagram details the architecture of the main decode loop. (Step 4 in the previous section)



It should be clear that the standard ogg vorbis decode chain still exists in this system. However the dotted lines in this diagram indicate lines of visibility. The viewpoint of the user is significantly simplified.

Each layer is relatively simple. It communicates over a simple interface with the next layer down or directly handles a small part of the decode chain.

This architecture does make the decoder larger than the sample vorbis decoder, and arguably a more complex architecture, however:

- It reduces the amount of information a user needs to know in order write a decoder (cf the sample decoder above).
- It reduces the amount of information a user needs to know in order to change the processing at any step
- It lends itself to reuse. Unlike the sample `libvorbis` decoder the decoder presented uses the same code for page and packet handling in the set-up steps as it does in the main decode loop.
- Essentially the presented decoder implements the basic scheduler required to ensure that each state machine is activated as required. The user is noted relied on to deal with every step right along the way.

Suppose a developer chose to develop their own decoder without using `libvorbissimple`. Suppose they too chose to use `libvorbis` directly and are interested in reusing code. That user would need to implement a similar model. Hence this reduces the amount of code a developer needs to write in their applications.

4 A Walk Through the `libvorbissimple` Decoder

This section literally walks through the code discussing the structure and what each section is for.

See section 5 for the flip side of this discussion.

A number of `#define`s (preprocessor macros defining constants) are made to improve clarity of the system. Rather than list them all here, We note the following:

- Anything positive is used as a truth value (rather than non zero)
- Anything 0 or less is either an error or a state tracked by the system. If a function encounters a negative value as a return value or a status code and it can't handle it, it immediately returns with that value to it's caller. (Essentially this is a similar style of exception handling as the standard socket libraries)
- One `#define - BUFSIZE` is used throughout the code for various buffer sizes. You are expected to use this when creating buffers to pass over to `libvorbissimple.`

A limitation of the system as a whole is that the minimum usable buffer size is 58 bytes. (This is a limitation of the sample vorbis decoder as well)

4.1 Necessary Includes

`libvorbissimple` is designed for use as a source code library. We pull it into our program thus: (note we also pull in stdio.h for all the usual reasons)

```
#include "libvorbissimple.c"
#include <stdio.h>
```

Note: The include above has a **.c** extension rather than the usual **.h** due to being a source code library

4.2 Decoder Structure

`libvorbissimple` is designed to allow client applications to have a simple structure. The inside of `libvorbissimple` also has a relatively simple call structure. We will come to that in section 5. The `libvorbissimple` sample decoder contains just one `main` function, which makes a minimal set of calls into `libvorbissimple` .

The following diagram indicates the call structure:

```
int main(int argc, char *argv[])
    Initialise
        ogg_vorbis_content * newOggVorbisContent(void);
        source_buffer * newSourceBuffer(FILE * fh, int buffersize);
    Main Loop
        decode_buffer * getAudio(ogg_vorbis_context * oggVorbisContext);
        Then depending on returned status, exit, or:
            void readData(source_buffer * sourceBuffer);
            void sendBytesForDecode(ogg_vorbis_context * ovc, source_buffer * sourceBuffer);
```

It should clear from the call structure that we pass an `ogg_vorbis_context` around the decoder so that each part of the system can maintain state between invocations.

This lends the decoder to being wrapped for use by other languages with relative ease. The user is not required to know the internal structure of `ogg_vorbis_context`.

4.3 Using libvorbissimple

The bulk of this `libvorbissimple` client is one fairly simple to use function as a result of the layering approach. First we have a small initialisation phase, followed by a simple decode loop and then a shut down.

4.3.1 Declarations

The following is the minimal set of declarations needed at the start of the code:

```
int main(int argc, char *argv[]) {
   bool decoding;
   ogg_vorbis_context * oggVorbisContext;
   source_buffer * sourceBuffer;
   decode_buffer * decodeBuffer;
```

The purpose of these variables is as follows:

`decoding`

This is used to track where we are with decoding the stream

`oggVorbisContext`

This data structure wraps up all the context required for decode. For example, this tracks all the state machines referred to in section 2. This is intended as a black box. Since the library is intended as a source code library, it's useful to be aware that this is where the state is stored.

Using this structure to wrap up the global context means that we can have multiple decode streams operating concurrently. In order to do this you would create an `ogg_vorbis_context` per decode to maintain local state. The decoder does presented in this section does not do this, and just decodes one ogg vorbis source.

`sourceBuffer`

You are required wrap up the data you send to `libvorbissimple` in a `source_buffer`. Your client code owns this buffer. The `source_buffer` has fields for file handles, buffer space and size. `libvorbissimple` provides functionality for creating and filling source buffers. However you is not required to use these, and may create and fill your own buffers.

`decodeBuffer`

The decoder creates these as return values. Each `decodeBuffer` contains 3 values : a return status, a pointer to the buffer and a buffer size. *The contents of the buffer is guaranteed **not** to stay valid between requests for audio.*

Since the user deals directly with all 3 of these data structures, the user of `libvorbissimple` is responsible for deallocating the memory associated with them. This approach is designed to simplify usage. (*"Do I allocate this? or that? Or?"* - **"yes, you are responsible for deallocating all of them"**)

The internals of these are dealt with in the `libvorbissimple` section.

4.3.2 Initialisation

Next we initialise our `oggVorbisContext` and our `sourceBuffer`:

```
decoding = TRUE;
oggVorbisContext = newOggVorbisContext();
sourceBuffer = newSourceBuffer(stdin,BUFSIZE);
```

If you do not use the `BUFSIZE` constant then dropping below the buffer size below 58 bytes ***will*** cause problems.

The file handle supplied to newSourceBuffer can be set to `NULL` rather than a valid file handle.

A file handle may be provided due to `libvorbissimple` providing a trivial function - `readData` - for reading from a file to fill source buffers. This sample decoder uses this facility to read ogg vorbis data from stdin. The buffer size is however required. The suggested value to use is supplied in the form of a constant `BUFSIZE` .

4.3.3 Main Loop

The main loop is then relatively simple:

```
while(decoding) {
   decodeBuffer = getAudio(oggVorbisContext);
   switch (decodeBuffer->status) {
      case NEEDDATA:
         readData(sourceBuffer);
         if (sourceBuffer->bytes > 0)
            sendBytesForDecode(oggVorbisContext, sourceBuffer);
         else
            decoding = FALSE;
         break;
      case HAVEDATA:
   /* We have to use the contents of the decodeBuffer before asking for more
   bytes, or else we lose the contents, this could mean just buffering */
         fwrite(decodeBuffer->buffer,sizeof(char),decodeBuffer->len,stdout);
         break;
      case NORMAL:
   /* Just so that we can signify normal status */
         break;
```

```
            default:  /* Unknown status, exit */
               decoding = FALSE;
               break;
         }
         free(decodeBuffer);
      }
```

Structures and Constants

Essentially the `switch` statement here is being used in the same way people tend to use exception handling. i.e. perform the task you want to do, and clean up errors afterwards. The specific exceptions here are modelled using constants. This is very similar to the way networking code uses the special `errno` global variable, without relying on the use of a global.

The result of `getAudio` is a `decode_buffer`. This contains the following elements:

```
typedef struct {
   char * buffer;
   int len;
   int status;
} decode_buffer;
```

Taking each attribute in turn:

- The `buffer` attribute is a pointer to the decoded audio - if any was available.
- The `len` attribute indicates the size of the buffer in bytes.
- `status` is an integer value indicating the result of the `getAudio` call

Symbolic constants are made available for the `status` attribute so that the caller can decide what to do with the buffer:

HAVEDATA

This is generally what we're interested in - it signifies to the caller that data exists in the `buffer` and is suitable for us to use. It's worth noting that we ***must*** use this data before the next call to `getAudio` . If we don't wish to use it immediately we ***must*** copy the data somewhere else before the next call to `getAudio`. If we do not, we lose that audio data.

NEEDDATA

This is the next most common return value. This indicates that the system cannot give us more decoded audio until it is provided with more data. Repeated calls to `getAudio` without supplying any more audio will result in further `NEEDDATA` values.

NORMAL

This value is returned if the system can for some reason not give us any data at this instant, but has to return control to us. For example, this can happen during header decode. This return status has very similar meaning to the `EAGAIN` error code from the standard C library. It means that you are expected to retry the call to `getAudio` again without sending more data to the decoder.

Any other status value returned is an internal error, and decoding should cease at that point with no further calls to `getAudio`.

Actions Taken

In this particular example our code does the following in each case:

HAVEDATA

Sends the audio in the `buffer` attribute of `decodeBuffer` to `stdout`. This uses a standard `fwrite` call. Note that the buffer size is defined by the number of chars it can contain (hence the `sizeof(char)` parameter, in the code at the beginning of section 4.3.3).

NEEDDATA

This means we need to send the system more data. As a result, it reads data into the `sourceBuffer` using the `readData` call. Since `sourceBuffer` was previously set up with `stdin` as the data source, it defaults to reading from that source. If we could not read any data, we've reached the end of file (or similar problem), and change our decoding status to `FALSE` in order to exit the loop. Otherwise we send the data to the decode subsystem in the hope that this is sufficient data for decode.

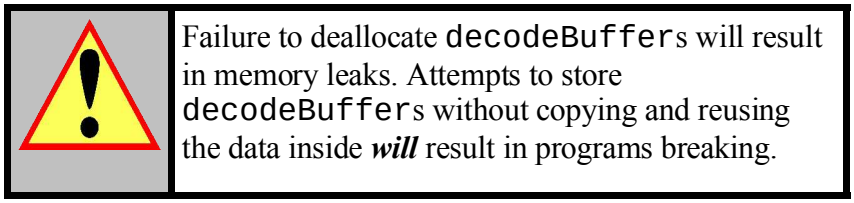
NORMAL

As noted above this is just a "try again" code from the decode subsystem. As a result we just retry on the next loop to get data.

Any other value

This indicates unexpected behaviour from the decoder. A client is expected to exit decoding at this point and free up all structures

After exiting the `switch` control structure, we have to deallocate the `decodeBuffer` we were sent.



4.3.4 Shut down

Finally, after the loop exits we have to do some clean up:

```
    free(sourceBuffer);
    free(oggVorbisContext);
    return 0;
}
```

We don't free up the `decodeBuffer` here because that's the last thing that happens inside the loop and is guaranteed to have occurred earlier.

4.3.5 Complete Listing

For convenience, and to put all the points so far in context, the complete listing for the sample `libvorbissimple` client is presented below:

```
#include "libvorbissimple.c"
#include <stdio.h>

int main(int argc, char *argv[]) {
   bool decoding = TRUE;
   ogg_vorbis_context * oggVorbisContext;
   source_buffer * sourceBuffer;
   decode_buffer * decodeBuffer;

   oggVorbisContext = newOggVorbisContext();
   sourceBuffer = newSourceBuffer(stdin,BUFSIZE);
   while(decoding) {
      decodeBuffer = getAudio(oggVorbisContext);
      switch (decodeBuffer->status) {
         case NEEDDATA:
            readData(sourceBuffer);
            if (sourceBuffer->bytes > 0)
               sendBytesForDecode(oggVorbisContext, sourceBuffer);
            else
               decoding = FALSE;
            break;
         case HAVEDATA:
            /* We have to use the contents of the decodeBuffer before asking
              for more bytes, or else we lose the contents, this could mean just
              buffering */
            fwrite(decodeBuffer->buffer,sizeof(char),decodeBuffer->len,stdout);
            break;
         case NORMAL:
            /* Just so that we can signify normal status */
            break;
         default: /* Unknown status, exit */
            decoding = FALSE;
            break;
      }
      free(decodeBuffer);
   }
   free(sourceBuffer);
   free(oggVorbisContext);
   return 0;
}
```

5 `libvorbissimple` internals

The basic structure of `libvorbissimple` is to present the user of the library a simple interface whilst having a layered structure internally. The top layer is implemented by creating a simple state machine, whose state is tracked by an `ogg_vorbis_context`.

This section will largely reference the code in `libvorbissimple` rather include it in-line.

Basic design principles in `libvorbissimple` :

- The top level of the system implements a simple state-machine. This is unavoidable, but the user is only aware of this in the simplest of manners.
- The coding style is *loosely* OO. Each structure has associated constructors rather than being allocated and initialised in-line. If a structure is passed back to the user of the library, the user is responsible for deallocating structures. If there are structures inside those structures, the library is responsible for deallocating those.
- Since the return values are normally used for returning errors, the `ogg_vorbis_context` is normally passed by reference. This is akin to object methods modifying the object they're bound to.
- Each layer of the system assumes that the layer below can provide the data it needs just by asking for it. This means that the layer will throw errors back up the chain.
- Errors are constants less than zero. This means that we can trivially check for error, and simultaneously encode **what** the error is. Generally error codes are not reused. This approach means that should an error occur during decode, we know more or less precisely where the problem occurred. The rationale here is to try to simplify the job of someone wishing to maintain or extend this code.

This style of coding was chosen to simplify creation of bindings for other languages.

5.1 Library Dependencies

`libvorbissimple` depends on the following external libraries/header:

- Standard libraries: stdio.h, stdlib.h, string.h, math.h
- Ogg headers for `libogg` : ogg/ogg.h
- Vorbis headers for `libvorbis` : vorbis/codec.h

5.2 Constants

General utility constants for use outside the library:

- `TRUE`, `FALSE`, `BUFSIZE` - the first 2 have obvious meaning, `BUFSIZE` however defines a constant that defines the buffer size a user should use when sending buffer data to the system (If the user is using the `readData` support function, this isn't an issue).

Constants used for top level external state:

- NORMAL, NEEDDATA, HAVEDATA - these have all been defined in section 4.3.3.

The following constants will be defined in later sections, so that they're described in context.

Constants used for top level internal state:

- STARTSTATE, CHECKSTREAMOGGVORBIS, READCOMMENTANDCODEBOOKHEADERS, MAINDATADECODELOOP, FAILSTATE

Other constants:

- PAGEREAD, PACKETREAD, COMMENTANDCODEBOOKHEADERSREAD - All success
- READFIRSTOGGPAGE, NOFIRSTPAGE - Return values during set up (latter is an error)
- CORRUPTPAGE, CORRUPTPACKET, NOTOGGVORBIS - errors during decode

5.3 Supporting Data Structures

As noted earlier, our users of libvorbissimple create source_buffer objects with raw ogg vorbis data in. They send these to the decoder. In return libvorbissimple sends back decode_buffer objects.

From a users of libvorbissimple's perspective, ogg_vorbis_context acts as a logical handle. We do not need/want to know the internal structure. From the perspective of the library, ogg_vorbis_context is used to track all the state required for decoding one ogg vorbis data stream . This is much the same way one might use a "self" or "this" object in object oriented code.

5.3.1 source_buffer

```
typedef struct {
   FILE * fh;
   char * buffer;
   int bytes;
   int buffersize;
} source_buffer;
```

Ownership

The user is responsible for creating, populating and deallocating source_buffers

External Support Code

There is support functionality in the library to help with creating and populating these buffers:

newSourceBuffer(FILE * fh, int buffersize)

This takes a file handle and a buffersize. The buffer size should take the value BUFSIZE and indicates the size of the buffer in terms of bytes. The result is an empty, but valid source_buffer . If the buffer allocation failed, the result is NULL.

readData(source_buffer * sourceBuffer)

This reads data from the fh (file handle) attribute stored inside the source_buffer and places the resulting data into the buffer attribute of the source_buffer . The number of bytes read is stored inside the bytes attribute of the source_buffer.

`sendBytesForDecode(ogg_vorbis_context * ovc, source_buffer *sourceBuffer)`

This takes the contents of the given `source_buffer` , and passes it to `libogg` . Specifically it performs the following steps:

- Asks `libogg` to allocate a shared data buffer
- Copies the data from the source buffer into this shared buffer
- Tells `libogg` how many bytes were written into the shared buffer

5.3.2 decode_buffer

The purpose of the `decode_buffer` is to allow the following logic:

```
(theBuffer, returnStatus) = getAudio(context)
```

Since C doesn't support multiple return values, we wrap the return values up in a structure. Essentially it can be viewed as a stub structure.

```
typedef struct {
   char * buffer;
   int len;
   int status;
} decode_buffer;
```

Ownership

`decode_buffer`s are created and populated by `libvorbissimple`. `libvorbissimple` is responsible for allocating and de-allocating the buffers inside these structures. The user of `libvorbissimple` is responsible for deallocating the structure itself.

Support Code

`newDecodeBuffer(int status)`

Given decode buffers are essentially a tuple of buffer space and a return status, the system allows for `decode_buffer`s to be created with an initial status. The result is an empty, but valid `decode_buffer`. If the buffer allocation failed, the result is NULL.

NB. This only allocates a `decode_buffer` structure- it does not populate it, or allocate any buffer space - it is merely used to pass back an existing buffer.

`getAudio(ogg_vorbis_context * oggVorbisContext)`

This function returns `decode_buffer`s, but operates on an `ogg_vorbis_context` and so is dealt with in section 5.5.1. See also section 4.3.3.

5.4 The `ogg_vorbis_context` structure

This structure is the key structure in `libvorbissimple`.

```
typedef struct {
   int decodeState;
   ogg_sync_state   oggSync;
   ogg_page         oggBitstreamPage;
   ogg_stream_state oggStream;
   ogg_packet       oggPacket;
   vorbis_info      vorbisInfo;
   vorbis_comment   vorbisComment;
   vorbis_dsp_state vorbisDSP;
   vorbis_block     vorbisWorkingBlock;
   ogg_int16_t convbuffer[BUFSIZE];
   bool streamInitialised;
   bool warnClipping;
   int headerPacketsRead;
   int convsize;
   char * buffer;
   int    bufferlen;
} ogg_vorbis_context;
```

It is used as an abstract data type in an object oriented fashion. Specifically a number of functions expect to receive an `ogg_vorbis_context` as their first parameter and modify attributes of this structure. If there is one attribute to pull attention to, it would be the `vorbisDSP` attribute, since this is used to handle the raw decode aspects.

This may seem initially unusual to some C programmers, however this is the same approach that the socket, file and X Windows systems use. This approach allows the code in `libvorbissimple` to be reused multiple times in a single piece of code. To do so you would create a new `ogg_vorbis_context` and use it in an identical manner to any other `ogg_vorbis_context` you have allocated.

Normally, this would contain a number of substructures. However, in the interests of simplifying this example, a flat data structure has been chosen. When decoding vorbis, you still need to track all of these values. As a result, splitting this structure numerous smaller structures is of dubious benefit, and simply raises the number of buffers and allocations the system must manage for little benefit. This shields the user of the library from implementation details. Having one structure similarly simplifies wrapping for other languages.

The fields in this structure are divided into the following groupings:

- A small collection attributes for tracking state.
- A collection of attributes that keep track of the ogg subsystem. These handle acceptance of data into the decode pipeline, taking pages out, and finally taking packets out.
- A collection of attributes for reading vorbis information. This includes meta data about the bit stream, a comment, as well as a handle for the actual decode process, along with working buffer space for decode.
- Finally there is a collection of attributes directly related to decoded data, which include a pass back buffer, it's size, and related information.

5.4.1 State Attributes

`int decodeState`

This is the key state attribute for the whole system. As we shall see later `getAudio` uses the value this stores to step through the various stages of decoding ogg vorbis. You can think of this as a value similar to a line number we're up to in the program.

`bool streamInitialised`

This is set to false until we've successfully retrieved the first vorbis packet from the stream. We *may* have read a number of ogg pages to reach this stage.

`int headerPacketsRead`

This is used during the initial decode set up. After we have read 3 header packets we will be either decoding vorbis data, or identified that we cannot decode the data supplied. It takes a number of calls to `getAudio` to identify this, so we need to store this value in the context.

5.4.2 Ogg Decode Attributes

If you are manually decoding ogg vorbis, you need 4 variables matching the data types/usage of the following 4 attributes.

`ogg_sync_state oggSync`

The value stored by this attribute is used by the `libogg` library to synchronise its buffers with the client application. Its primary purpose is to act as a buffer upon which a raw bytes to pages transform can be performed. It has a number of functions that can act upon it. These handle interaction with the client of the `libogg` library to ensure buffers stay synchronised. Functions in `libogg` / `libvorbis` that operates on `ogg_sync_state`s are:

Initialisation:

- `ogg_sync_init` to reset data structures when decoding a new data stream.

Receiving data:

- `ogg_sync_buffer` to track the buffers allocated to the client of `libogg`
- `ogg_sync_wrote` to receive information from the client application as to how many bytes the client application placed inside the buffer.

Extracting pages:

- `ogg_sync_pageout` uses the information stored in an `ogg_sync_state` to determine the buffer from which to extract ogg pages

`ogg_page oggBitstreamPage`

This essentially forms a buffer during decode. Pages are taken out of the raw page buffer and placed into the stream buffer. Functions in `libogg` / `libvorbis` that operates on `ogg_page`s are:

- `ogg_page_serialno` takes `ogg_page` structures to generate a serial number during stream initialisation.
- `ogg_sync_pageout` uses `ogg_page` structures as a temporary destination for retrieval of pages.
- `ogg_sync_pageout` takes this temporary destination and inserts it into the a buffer that is used for aggregating pages into packets.

`ogg_stream_state oggStream`

This variable is used for tracking the page to packet buffer. Pages are placed into the buffer tracked by this state variable, and packets get pulled out. Functions in `libogg` / `libvorbis` that operates on `ogg_stream_state`s are:

- Before this buffer can be used, `ogg_stream_init` needs to be called. This is done once per stream, and done at the beginning of the decode chain, using the serial number relating to the current value of the `oggBitstreamPage` attribute. In this context, this will also be the first page in the ogg stream.
- `ogg_stream_pagein` is used to take the `ogg_page` stored in the `oggBitstreamPage` attribute, and to place it in the buffer tracked by the `oggStream` attribute.
- `ogg_stream_packetout` is used to remove a packet from the buffer tracked by `oggStream`. The removed packet is then tracked by the `oggPacket` attribute of an `ogg_vorbis_context`.

`ogg_packet oggPacket`

This attribute is used as a temporary variable to pass back packets for decode. The contents of this get used to pass to the vorbis decode system as a vorbis block. Functions in `libogg` / `libvorbis` that operates on `ogg_packet`s are:

- `vorbis_synthesis_headerin` uses `oggPacket` as a data source to populate the `vorbisInfo` and `vorbisComment` attributes of the `ogg_vorbis_context`.
- `ogg_stream_packetout` (see note above)
- `vorbis_synthesis` uses `oggPacket` to initialise the `vorbisWorkingBlock` attribute. This appears to copy the contents of `oggPacket` into the `vorbisWorkingBlock`. This is conjecture. Furthermore this also appear to be the actual main decode step. Again, this is conjecture.

5.4.3 Vorbis Decode Attributes

`vorbis_info vorbisInfo`

This attribute is a structure that stores all the static vorbis bit stream settings. It is populated by reading the first three ogg packets in the supplied bit stream. Once this has been read, the `convsize` attribute is calculated based upon the number of audio channels listed in the `vorbisInfo` attribute. The channels information is later also used in order to correctly transform the decoded data into something suitable for playback. It also dictates the sizes of various buffers created for decode. Functions in `libogg` / `libvorbis` that operates on the `vorbisInfo` structure are:

- `vorbis_info_init` initialises the `vorbisInfo` structure.
- `vorbis_synthesis_headerin` populates the `vorbisInfo` structure based on the contents of the first 3 packets `oggPacket`s in the data stream.
- `vorbis_synthesis_init` uses the `vorbisInfo` attribute to initialise the `vorbisDSP` attribute of the `ogg_vorbis_context`

`vorbis_comment vorbisComment`

This attribute contains all the user comments. These are not specifically needed for decode, but the library populates this anyway during decode. Functions in `libogg` / `libvorbis` that operates on the `vorbisComment` structure are:

- `vorbis_comment_init` initialises the `vorbisComment` structure.
- `vorbis_synthesis_headerin` populates the `vorbisComment` structure based on the contents of the first 3 packets `oggPacket`s in the data stream.

`vorbis_block vorbisWorkingBlock`

During decode we need to take individual blocks and decode them. This attribute is used as local working space for the packet to PCM decode step. If the system had multiple CPUs decode of multiple blocks could occur in parallel. This would be performed by allocating multiple `vorbis_block` structures duplicating the steps taken with the single `vorbisWorkingBlock` attribute.

Functions in `libogg` / `libvorbis` that operates on the `vorbisWorkingBlock` structure are:

- `vorbis_block_init` uses the `vorbisDSP` attribute to initialise the `vorbisWorkingBlock` attribute for decode. In all likelihood this actually allocates memory/buffer space for decode. ***The lack of docs leaves this point as conjecture.***
- `vorbis_synthesis` uses `oggPacket` to initialise the `vorbisWorkingBlock` attribute. This appears to copy the contents of `oggPacket` into the `vorbisWorkingBlock`. ***This is conjecture.*** Furthermore this also appear to be the actual main decode step. ***Again, this is conjecture.*** It might mean "prepare for synthesis". Either way, it has to happen before `vorbis_synthesis_blockin`.

- `vorbis_synthesis_blockin` The meaning of this is unclear. This appears to mean "take the data in the vorbis working block, and decode into an internal state buffer". Specifically, take the data in the `vorbisWorkingBlock` attribute and pass ownership to the `vorbisDSP` attribute. ***Without documentation for `libvorbis`, this is conjecture.*** This does match the other decode layers - of having buffers for passing data around and having buffers for work. Normally this would mean read a block of data into the vorbis synthesis buffer.

`vorbis_dsp_state vorbisDSP`

This attribute is used by `libvorbis` to handle the key working state for the packet to PCM decode. When the array of array of floats returned by the decode process is being converted to integer PCM, there is the possibility of clipping. The library provides a debugging flag to indicate if clipping occurs. In this situation, the system uses the sequence attribute of the `vorbisDSP` attribute to identify where in the source bit stream the clipping occurred.

Functions in `libogg` / `libvorbis` that operates on the `vorbisDSP` structure are:

- `vorbis_synthesis_init` initialises the `vorbisDSP` attribute using the bit stream information contained in the `vorbisInfo`.
- `vorbis_block_init` uses the `vorbisDSP` attribute to initialise a working block for decode (the `vorbisWorkingBlock` attribute).
- `vorbis_synthesis_blockin` – meaning of this is unclear. This appears to mean "take the data in the vorbis working block, and decode into an internal state buffer". Specifically, take the data in the `vorbisWorkingBlock` attribute and pass ownership to the `vorbisDSP` attribute. ***Without documentation for `libvorbis` , this is conjecture.*** This does match the other decode layers - of having buffers for passing data around and having buffers for work. Normally this would mean read a block of data into the vorbis synthesis buffer.
- `vorbis_synthesis_pcmout` takes the decoded data out of the `vorbisDSP` attribute's internal buffer, and passes it back to the user in the form of a point to an array of array of `float` values. We pass the address of a pointer to `libvorbis`, which allocates a buffer, and modifies our pointer to point at the decoded PCM data buffer.

  The decoded PCM data is an array of array of `float` values, not integer PCM. `libvorbissimple` converts this data to integer PCM data.
- `vorbis_synthesis_read` is uses the `vorbisDSP` data attribute to tell `libvorbis` how many bytes of decoded data we consumed. This is necessary because `libvorbis` sends back a shared buffer and allows the client of the library to consume as many or as few bytes as needed. This may or may not be the entire buffer.

5.4.4 Decode/Passback Attributes

`ogg_int16_t convbuffer[BUFSIZE]`

This is the buffer used in the final steps of decode to place the data into for us to use. This buffer is kept private to `libvorbissimple`. However when implementing your own decoder using `libvorbis` directly, you would need to allocate this buffer and use it for accepting data from `libvorbis`.

`bool warnClipping`

Simple flag to allow the user to determine if clipping occurred at all during the decode of the current block. Clipping is defined as values that are generated by the vorbis library which are beyond the levels that would normally fit inside a 16 bit value space.

`int convsize`

This defaults to the size of the buffer divided by the number of channels. Essentially it's a byte count of the amount of bytes each channel takes up. (This becomes important because the system tracks the number of samples it takes out, and this is limited by the size of the conversion buffer.)

`char * buffer`

This is the buffer we wrap up and pass back to the user.

`int bufferlen`

This is the size of the buffer.

5.4.5 Support Code and Functional Overview

`newOggVorbisContext(void)`

Creates an initialised `ogg_vorbis_context` . If this succeeds it returns the fully initialised structure. If this fails, return `NULL`. Specifically, the `decodeState` is set to `STARTSTATE` , and all other attribute values are set to zero, `NULL` or `FALSE`.

The following functions also operate on the `ogg_vorbis_context` structure:

- `getAudio` - high level function that the user of `libvorbissimple` interacts with. Since this forms a simple state machine, the majority of other functions in `libvorbissimple` that operate on an `ogg_vorbis_context` may fail due to needing more data.
- `readPage` - Attempts to read an `ogg_page` out of the buffer managed by `ogg_sync_state`.
- `readPacket` - Attempts to read an `ogg_packet` out of the buffer managed by `ogg_stream_state`.
- `checkStreamOggVorbis` - Looks at the first `ogg_packet` in the data stream to verify the data is indeed ogg vorbis.
- `readCommentAndCodebookHeaders` - Reads the 2rd and 3rd ogg packets from the data stream in order to set up the decode correctly.
- `decodeDataStream` - Performs most of the work in actually decoding the data stream, one packet at a time.
- `PCMfloatsToIntPCM` - Converts the float based PCM generated by `libvorbis` into integer based PCM. It needs the number of audio channels from the `ogg_vorbis_context` in order to do this.
- `clearBuffer` - resets the `ogg_vorbis_context` buffer attributes, freeing up any memory used by the buffer in the process.
- `bufferResults` - takes the decoded results out of the buffer shared with the `libvorbis` library, and places it into the buffer shared with the user.

These all relate to different layers in the decode process. Since they make up the bulk of `libvorbissimple` they will be dealt with layer by layer.

5.5 `libvorbissimple` by layers

This section describes first of all the top level state machine that the decoder implements. This is followed by a description of the top layer of decode functions that the top level state machine steps through.

5.5.1 The Top Level State Machine

The top level state machine is implemented in `getAudio`. It is relatively simple, and essentially steps through the major top level steps needed for decoding vorbis files. Specifically it consists of the following states:

`STARTSTATE`

Clearly we start here. At this point in time, the system is initialised, but has no data to work from. As a result it creates a `decode_buffer` with no data, and sets the status to `NEEDDATA`. This is returned to the user. We expect to be sent data after this, so we set our next state to `CHECKSTREAMOGGVORBIS`.

`CHECKSTREAMOGGVORBIS`

In this state, the system uses the call `checkStreamOggVorbis` to try and identify the incoming data stream. This will always cause a need to read data at least once - since the initial packets must be read. As a result if `checkStreamOggVorbis` returns `NEEDDATA` , then the top level state machine stays in the same state - i.e. in the `CHECKSTREAMOGGVORBIS` state, and the status in the returned `decode_buffer` is set to `NEEDDATA`. If the result of `checkStreamOggVorbis` is success (any positive value), then the stream is indeed ogg vorbis, and the system flips into the `READCOMMENTANDCODEBOOKHEADERS` state.

`READCOMMENTANDCODEBOOKHEADERS`

In this state, the system uses the call `readCommentAndCodebookHeaders` to try and set up the decoder for the incoming data stream. This may or may not cause a need to read data, depending on packet/page sizes in the stream relative to internal buffers. The valid return values for this are `COMMENTANDCODEBOOKHEADERSREAD` and `NEEDDATA`. In the former case, this means that the system is able to enter the `MAINDATADECODELOOP` state. In the latter case, the system remains in the `READCOMMENTANDCODEBOOKHEADERS` state. All other return values are errors, and the decoder enters the `FAILSTATE`.

MAINDATADECODELOOP

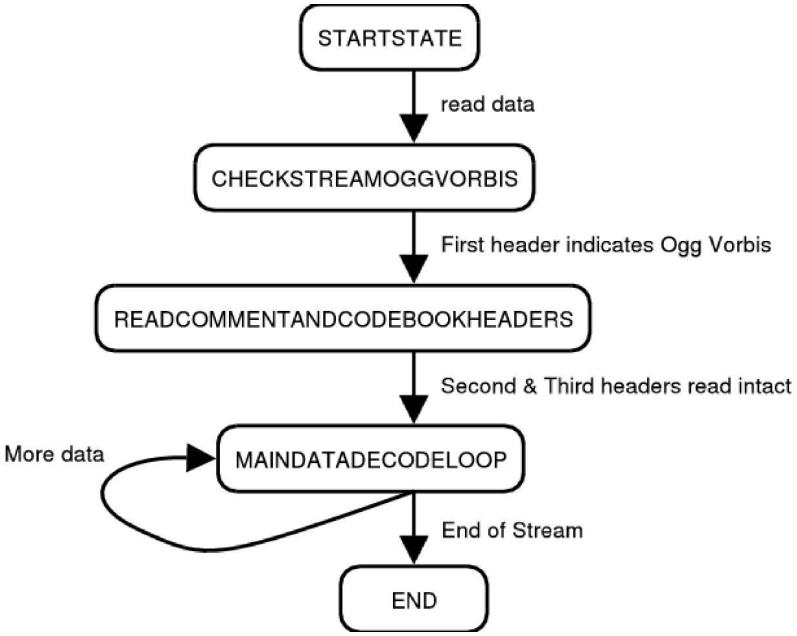
This is where the bulk of the time is spent in the code. The system makes a call to `decodeDataStream` and this has 2 logical successful outcomes:

- The system doesn't have enough data for decode, and needs some. It therefore sends back a `NEEDDATA` return status via the `decode_buffer` object.
- The system has enough data to perform at least one decode step, and successfully decodes a packet. In this scenario we copy the pointers we created and pass those back to the user along with a `HAVEDATA` return status via the `decode_buffer` object.

FAILSTATE

This state is used symbolically to indicate the system has finished decoding due to a failure.

The following diagram summarises the statemachine, and indicates the simplicity involved:



5.5.2 Top Level State Functions

A summary of the top level functions called in the top level state machine follows.

`checkStreamOggVorbis`

This function performs the following tasks:

- Initialises the `vorbisInfo` and `vorbisComment` structures
- Attempts to read a packet
    - If this fails, the return value is thrown back up the chain, without checking for what the error was.
- Otherwise, it attempts to initialise the contents of the `vorbisInfo` and `vorbisComment` structures using the current contents of the `oggPacket` structure.
    - If this fails, the stream isn't ogg vorbis, and a message to this effect is returned (`NOTOGGVORBIS`).
- Otherwise the stream *is* vorbis and a message is returned to that effect.

`readCommentAndCodebookHeaders`

This function operates in a very similar way.

- First of all, it checks a guard - it's only meant to deal with 2 packets. If it's called after 2 packets have been successfully read without creating a new `ogg_vorbis_context`, it returns a `FAILSTATE` value.
- It then enters a loop, which is exited one of three ways:
    - 2 packets are successfully read, and used to populate the `vorbisInfo` and `vorbisComment` attributes of the `ogg_vorbis_context`.
    - An extra packet is needed and the return value `NEEDDATA` is returned by `readPacket`.
    - Some other error occurs while reading a packet.
- After the loop, the conversion size (used in buffer management) is calculated based on the number of channels in the stream, and the `vorbisDSP` and `vorbisWorkingBlock` attributes are initialised.
- If the function succeeds, the function returns the integer 1.

`decodeDataStream`

This consists largely of a loop, which the system tries to loop through forever. Clearly this will not be the case, and the exit conditions are:

- An extra packet is needed and the return value `NEEDDATA` is returned by `readPacket`.
- Some other error occurs while reading a packet.
- We have successfully decoded some vorbis audio data

Assuming success for every single call in the function, the following happens:

- We read a packet from the vorbis buffers

- We take this packet and use it to initialise/populate the `vorbisWorkingBlock` attribute
- We use the attribute to prime the vorbisDSP attribute. Specifically, it looks like it attempts to move the `vorbisWorkingBlock` into a buffer owned by `vorbisDSP` - freeing up `vorbisWorkingBlock` for reuse.
- The next step is to take as many samples out of the buffer owned by `vorbisDSP` as possible. This is done as follows:
    - Make a call to `vorbis_synthesis_pcmout` passing over a buffer we own for `libvorbis` to populate.
    - Take the samples provided, and convert the from float based PCM samples into integer based PCM samples.
    - These samples are then buffered and we attempt to get more samples to decode
- At some point we will exit the loop and assuming we entered the loop, return `HAVEDATA`
- There are various failure points along the way:
    - Reading a packet may require more data, or simply fail
    - We may not be able to get a block out of the ogg packet. (Corruption perhaps)
    - We may not get audio out of the buffer managed by the `vorbisDSP` attribute
- Aside from the errors previously discussed these are either managed by the decoder or attempted to be skipped over.

Aside from these functions there are a small number of other functions used to support the decode process - for converting float PCM to integer PCM, clearing buffers, appending to buffers. These are covered in section *5.5.5 Decode Support Functions*.

5.5.3 Reading Packets

This is handled by one function - `readPacket` . The logic in it might seem a little odd, but it's designed this way to handle some odd edge cases. Specifically when we initially call the function, there may already be sufficient data in the buffer referenced by `oggStream` to retrieve many packets at once. As a result we do the following:

- Try to extract a packet
- If this succeeds we set a flag `packetRead` to avoid entering the loop
- If there is an error other than insufficient data, we return the `CORRUPTPACKET` value.
- Then if we haven't successfully read a packet yet, we enter a loop to try and put sufficient `ogg_page` structures into the buffer managed by `oggStream` until we can extract a packet successfully. Since this relies on calls to `readPage` which may in turn need more data from the client of the library, this may fail. In this case, we need to return the `NEEDDATA` value back to the caller to propagate up the chain.

5.5.4 Reading Pages

This is handled by one function - `readPage`. The structure of this is trivial really. Using `ogg_sync_pageout` it attempts to take a page out of the buffer owned by the `oggSync` attribute. If this fails we either return `NEEDDATA` or `CORRUPTPAGE`.

If the stream is not initialised (eg this is the first packet) some further processing occurs. Using `ogg_stream_init` it places the page in the buffer owned by the `oggStream` attribute. At the same time a call is made to `ogg_page_serialno`. At this point, the stream is initialised, so the attribute flag `streamInitialised` is set to true.

Assuming successful page read, the system returns the value `PAGEREAD`.

5.5.5 Decode Support Functions

The following functions are used to support the decode process:

`PCMfloatsToIntPCM`

Converts the floating point PCM to integer PCM. The float PCM is passed through as a pointer to and array of array of floats, and the result is stored in the conversion buffer in the `oggVorbisContext`. The conversion happens one audio channel at a time. Whilst the float values **should** be between -1 and 1 in practice `libvorbis` produces values outside this range, so the PCM values need to be clipped to 16 bit signed integer values.

`clearBuffer`

This function is called every time we enter `decodeDataStream`. What it does is this:

- It frees up the memory owned by the `oggVorbisContext buffer` attribute if it currently pointing at allocated data.
- Sets the `buffer` attribute to `NULL`
- Sets the `bufferlen` attribute to `0`. This ensures that the memory we use is freed up regularly, reducing the risk of memory leaks. This is also why users should not expect buffered data to be valid between invocations to `getAudio`.

`bufferResults`

This function is used by `decodeDataStream` to append the most recently decoded data - in the `convbuffer` attribute of the `oggVorbisContext` . As a result the data which we pass back to the user may be the result of several conversions - which is why we do not simply pass back the `convbuffer` attribute.

6 Summary

This document has discussed briefly what ogg vorbis is and provides an overview of the ogg vorbis decode chain. Programs that use `libvorbis` directly are expected to create, manipulate and push the decode chain along at all points in time. This was followed by a discussion of life cycle of a program using `libvorbis` directly.

The remainder of the document detailed a sample decoder designed for readability and simple reuse. The decoder was divided into two halves:

- A common decoder section comprising of the functionality most programmers are generally interested in - specifically just decoding audio.
- A reusable source code library designed for inclusion into other programs verbatim and for modification/extension.

This discussion was divided into the following areas:

- An overview of the architecture - to explain why the structure is as it is.
- A walk through a sample client of the reusable source code library.
- A walk through the dependencies, constants, data structures, their support functions, and then through the functionality of the reusable source code library layer by layer.

The reason for the detail here is largely due to the current lack of documentation of `libvorbis`. Hopefully this document and accompanying code goes someway to rectifying this issue.

7 Appendix 1: Cross Reference Diagram

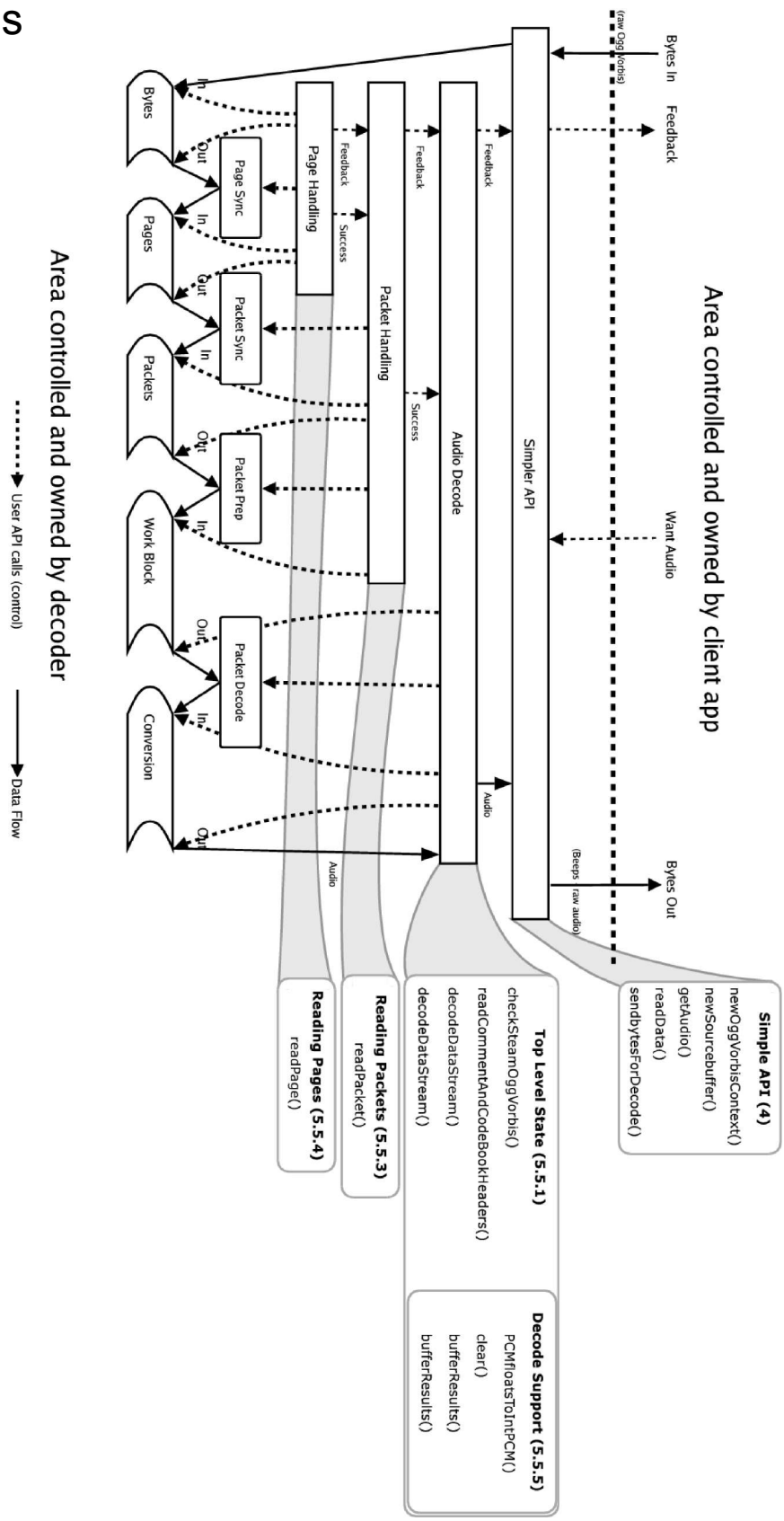
Key libogg/libvorbis by decode stage

Bytes
- ogg_sync_init()
- ogg_sync_buffer()
- ogg_sync_wrote()

Pages
- ogg_sync_pageout()
- ogg_page_serialno()
- ogg_stream_init()
- ogg_stream_pagein()

Packets
- ogg_stream_packetout()
- vorbis_synthesis_headerin()
- vorbis_comment_init()
- vorbis_info_init()
- vorbis_block_init()

Block decode
- vorbis_synthesis_init()
- vorbis_synthesis()
- vorbis_synthesis_blockin()
- vorbis_synthesis_pcmout()
- vorbis_synthesis_read()



8 Appendix 2: libvorbis API Description

As noted in several places, this has been based on working back from the code. In some places the descriptions are conjecture.

`vorbis_comment_init` initialises `vorbis_comment` structures. This appears to be a memory allocation step primarily.

`vorbis_info_init` initialises `vorbis_info` structures. This appears to be a memory allocation step primarily.

`vorbis_synthesis_init` uses the bit stream information contained in the `vorbis_info` structure to initialise a `vorbis_dsp_state` attribute ready for decode.

`vorbis_synthesis` uses an `ogg_packet` structure to initialise a `vorbis_block` structure. This appears to copy the contents of the `ogg_packet` into the `vorbis_block`. ***This is conjecture.*** Furthermore this also appear to be the actual main decode step. ***Again, this is conjecture.*** It might mean "prepare for synthesis". Either way, it has to happen before `vorbis_synthesis_blockin`.

`vorbis_synthesis_headerin` uses the first 3 `ogg_packet`s from the data stream as a data source to populate the `vorbis_info` and `vorbis_comment` structures.

`vorbis_block_init` uses the `vorbis_dsp_state` structure to initialise a `vorbis_block` attribute for decode. In all likelihood this actually allocates memory/buffer space for decode. ***This is conjecture.***

`vorbis_synthesis_blockin` The meaning of this is unclear. This appears to mean "take the data in the vorbis working block, and decode into an internal state buffer". Specifically, take the data in a `vorbis_block` structure and pass ownership to the `vorbis_dsp_state` structure. This does match the other decode layers - of having buffers for passing data around and having buffers for work. Normally this would mean read a block of data into the vorbis synthesis buffer. ***This is conjecture.***

`vorbis_synthesis_pcmout` takes the decoded data out of the `vorbis_dsp_state` structure's internal buffer, and passes it back to the user in the form of a point to an array of array of `float` values. We pass the address of a pointer to `libvorbis`, which allocates a buffer, and modifies our pointer to point at the decoded PCM data buffer.

The decoded PCM data is an array of array of `float` values, not integer PCM.

`vorbis_synthesis_read` uses the `vorbis_dsp_state` structure as a handle to tell `libvorbis` how many bytes of decoded data we consumed. This is necessary because `libvorbis` sends back a shared buffer and allows the client of the library to consume as many or as few bytes as needed. This may or may not be the entire buffer.