libbw64 Documentation

Release 0.10.0

Institut für Rundfunktechnik GmbH

Jan 23, 2023

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Requirements and dependencies

- compiler with C++11 support
- CMake build system (version 3.5 or later)

1.2 Installation

1.2.1 macOS

On macOS you can also use homebrew to install the library. You just have to add the IRT's NGA homebrew tap and can then use the usual install command.

```
brew tap irt-open-source/homebrew-nga
brew install libbw64
```

1.2.2 Manual

Alternatively clone the Git repository and install the library system wide using the CMake build system. See the following instructions for *nix systems.

```
git clone git@github.com:irt-open-source/libbw64.git
cd libbw64
mkdir build && cd build
cmake ..
make
make install
```

1.2.3 Copy headers

The `libbw64` is a header-only library so installing the library is not by all means necessary. It is also possible to just copy the content of the `include` directory to your project and make sure, that the `bw64` folder is in your `PATH`, that the header files can be found by the compiler.

1.3 CMake

As the library uses CMake as a build system it is really easy to set up and use if your project does too. Assuming you have installed the library, the following code shows a complete CMake example to compile a program which uses the `libbw64`.

```
cmake_minimum_required(VERSION 3.8)
project(libbw64_example VERSION 1.0.0 LANGUAGES CXX)

find_package(bw64 REQUIRED)

add_executable(example example.cpp)
target_link_libraries(example PRIVATE bw64)
```

If you prefer not to install the library on your system you can also use the library as a subproject. You can just add the library as a CMake subproject. Just add the folder containing the repository to your project and you can use the `bw64` target.

```
cmake_minimum_required(VERSION 3.5)
project(libbw64_example VERSION 1.0.0 LANGUAGES CXX)

add_subdirectory(submodules/libbw64)

add_executable(example example.cpp)
target_link_libraries(example PRIVATE bw64)
```

Note: If `libbw64` is used as a CMake subproject the default values of the options

- `BW64_UNIT_TESTS`
- `BW64_EXAMPLES`
- `BW64_PACKAGE_AND_INSTALL`

are automatically set to `FALSE`.

CHAPTER TWO

TUTORIAL

In this tutorial we will create a simple application which adjusts the level of all channels in a BW64 file and writes the output to a new file. We assume that the `include` path of the library is added to the `PATH`.

2.1 Basic structure

Let us start with the basic structure of our programme.

```
#include <iostream>
#include <bw64/bw64.hpp>

const unsigned int BLOCK_SIZE = 4096;

int main(int argc, char const* argv[]) {
  if (argc != 2) {
    std::cout << "usage: " << argv[0] << " [INFILE]" << std::endl;
    exit(1);
  }
  auto inFile = bw64::readFile(argv[1]);
  std::vector<float> buffer(BLOCK_SIZE * inFile->channels());
  while (!inFile->eof()) {
    auto readFrames = inFile->read(&buffer[0], BLOCK_SIZE);
    // TODO: process samples
  }
  return 0;
}
```

We include the header and open the file we want to read using the *`bw64::readFile()`* function and add a `while` loop in which we read the samples in a block buffer. The `bw64::Bw64Reader::read()` expects a float array and the number of frames, the function should try to read. One frame contains one sample for each channel. So if the `bw64::Bw64Reader::read()` function should try to read `N` frames, the buffer must be at least `N * CHANNELS` big. The samples are written into the buffer in a channel interleaved order, as illustrated in the following table.

Index	0	1	2	3	4	5	6	7	8	9	10
Channel	0	1	0	1	0	1	0	1	0	1	0
Sample	0	0	1	1	2	2	3	3	4	4	5

Note that we don't need to close our file at the end of the programme. This will be done automatically when `inFile` is destroyed at the end of the programme.

2.2 Write files

As a next step we also prepare our output file to write the samples.

```
#include <iostream>
#include <bw64/bw64.hpp>

const unsigned int BLOCK_SIZE = 4096;

int main(int argc, char const* argv[]) {
  if (argc != 3) {
    std::cout << "usage: " << argv[0] << " [INFILE] [OUTFILE]" << std::endl;
    exit(1);
  }
  auto inFile = bw64::readFile(argv[1]);
  auto outFile =
      bw64::writeFile(argv[2], inFile->channels(), inFile->sampleRate(),
                      inFile->bitDepth(), inFile->chnaChunk(), inFile->axmlChunk());

  std::vector<float> buffer(BLOCK_SIZE * inFile->channels());
  while (!inFile->eof()) {
    auto readFrames = inFile->read(&buffer[0], BLOCK_SIZE);
    // TODO: process samples
    outFile->write(&buffer[0], readFrames);
  }
  return 0;
}
```

We use the information from the input file we opened to initialize our output file. We also need to add the `chna` and `axml` chunks from the input file to the output file during initialization. We can directly use the buffer we passed to the `bw64::Bw64Reader::read()` in the `bw64::Bw64Writer::write()` function to write the unmodified samples. So also the `bw64::Bw64Writer::write()` expects the order of the samples to be interleaved as described above.

2.3 Add signal processing

To make our example complete, let us add some basic signal processing and adjust the gain of all channels.

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <bw64/bw64.hpp>

const unsigned int BLOCK_SIZE = 4096;

int main(int argc, char const* argv[]) {
  if (argc != 4) {
    std::cout << "usage: " << argv[0] << " [INFILE] [OUTFILE] [GAIN]"
              << std::endl;
    exit(1);
  }
  auto inFile = bw64::readFile(argv[1]);
  auto outFile =
      bw64::writeFile(argv[2], inFile->channels(), inFile->sampleRate(),
                      inFile->bitDepth(), inFile->chnaChunk(), inFile->axmlChunk());
```

(continues on next page)

(continued from previous page)

```
  std::vector<float> buffer(BLOCK_SIZE * inFile->channels());
  float gain = atof(argv[3]);
  while (!inFile->eof()) {
    auto readFrames = inFile->read(&buffer[0], BLOCK_SIZE);
    std::transform(buffer.begin(), buffer.end(), buffer.begin(),
                   [gain](float value) { return value * gain; });
    outFile->write(&buffer[0], readFrames);
  }
  return 0;
}
```

CHAPTER THREE

CHANGELOG

3.1 Unreleased

This version fixes a number of buffer overruns, integer overflows, and uses of uninitialised data which may be triggered by reading malformed files; all users are advised to upgrade.

3.1.1 Added

- library can now easier be used as a CMake subproject
- new CMake option `BW64_PACKAGE_AND_INSTALL`

3.1.2 Changed

- Renamed CMake library target name from `libbw64` to `bw64`
- Renamed CMake option `UNIT_TESTS` to `BW64_UNIT_TESTS`
- Renamed CMake option `EXAMPLES` to `BW64_EXAMPLES`
- `FormatInfoChunk::formatTag` now matches the formatTag in the file, rather than always returning 1
- fmt parsing is stricter – the chunk size must match the use of cbSize, and the presence if extra data is checked against the formatTag

3.1.3 Fixed

- Fix sample rate parameter type in `writeFile()` and `BW64Writer` ctor to support 96k samplerates
- fmt extra data is now written correctly

3.2 0.10.0 - (January 18, 2019)

3.2.1 Added

- Additional unit tests

3.2.2 Changed

- Use `Catch2` instead of `Boost.Test` for unit testing

3.2.3 Fixed

- Fix `Bw64Reader::seek()` and `Bw64Reader::tell()` implementation
- RIFF chunk size calculation

3.3 0.9.0 - (July 23, 2018)

Initial release

CHAPTER

FOUR

MAIN FUNCTIONS

`bw64::Bw64Reader` and `bw64::Bw64Writer` classes usually should not be created manually. Instead the two builder functions to either read or write a file should be used.

std::unique_ptr<*Bw64Reader*> `bw64::`**`readFile`**`(`**`const`** std::string &*filename*`)`

Open a BW64 file for reading.

Convenience function to open a BW64 file for reading.

Parameters

- `filename`: path of the file to read

Return `unique_ptr` to a *Bw64Reader* instance that is ready to read samples.

std::unique_ptr<*Bw64Writer*> `bw64::`**`writeFile`**`(`**`const`** std::string &*filename*, uint16_t *channels* = 1u, uint32_t *sampleRate* = 48000u, uint16_t *bitDepth* = 24u, std::shared_ptr<*ChnaChunk*> *chnaChunk* = nullptr, std::shared_ptr<*AxmlChunk*> *axmlChunk* = nullptr`)`

Open a BW64 file for writing.

Convenience function to open a new BW64 file for writing, adding `axml` and `chna` chunks.

If passed to this function, the `axml` and `chna` chunks will be added to the BW64 file *before* the actual data chunk, which is the recommended practice if all components are already known before writing a file.

Return `unique_ptr` to a *Bw64Writer* instance that is ready to write samples.

Parameters

- `filename`: path of the file to write
- `channels`: the channel count of the new file
- `sampleRate`: the samplerate of the new file
- `bitDepth`: target bitdepth of the new file
- `chnaChunk`: Channel allocation chunk to include, if any
- `axmlChunk`: AXML chunk to include, if any

CHAPTER

FIVE

BW64 FILE CLASSES

class Bw64Reader

Representation of a BW64 file.

Normally, you will create an instance of this class using *bw64::readFile()*.

This is a RAII class, meaning that the file will be openend and initialized (parse header, format etc.) on construction, and closed on destruction.

Public Functions

Bw64Reader(**const** char **filename*)

Open a new BW64 file for reading.

Opens a new BW64 file for reading, parses the whole file to read the format and identify all chunks in it.

Note For convenience, you might consider using the `readFile` helper function.

~Bw64Reader()

Bw64Reader destructor.

The destructor will automatically close the file opened in the constructor.

uint32_t **fileFormat**() **const**

Get file format (RIFF, BW64 or RF64)

uint32_t **fileSize**() **const**

Get file size.

uint16_t **formatTag**() **const**

Get format tag.

uint16_t **channels**() **const**

Get number of channels.

uint32_t **sampleRate**() **const**

Get sample rate.

uint16_t **bitDepth**() **const**

Get bit depth.

uint64_t **numberOfFrames**() **const**

Get number of frames.

uint16_t **blockAlignment**() **const**

Get block alignment.

std::shared_ptr<*DataSize64Chunk*> **ds64Chunk**() **const**

Get 'ds64' chunk.

Return `std::shared_ptr` to *DataSize64Chunk* if present and otherwise a nullptr.

std::shared_ptr<*FormatInfoChunk*> **formatChunk**() **const**

Get 'fmt ' chunk.

Return `std::shared_ptr` to *FormatInfoChunk* if present and otherwise a nullptr.

std::shared_ptr<*DataChunk*> **dataChunk**() **const**

Get 'data' chunk.

Warning This method usually should not be called, as the acces to the *DataChunk* is handled seperately by the *Bw64Reader* class .

Return `std::shared_ptr` to *DataChunk* if present and otherwise a nullptr.

std::shared_ptr<*ChnaChunk*> **chnaChunk**() **const**

Get 'chna' chunk.

Return `std::shared_ptr` to *ChnaChunk* if present and otherwise a nullptr.

std::shared_ptr<*AxmlChunk*> **axmlChunk**() **const**

Get 'axml' chunk.

Return `std::shared_ptr` to *AxmlChunk* if present and otherwise a nullptr.

std::vector<ChunkHeader> **chunks**() **const**

Get list of all chunks which are present in the file.

bool **hasChunk**(uint32_t *id*) **const**

Check if a chunk with the given id is present.

void **seek**(int32_t *offset*, std::ios_base::seekdir *way* = std::ios::beg)

Seek a frame position in the *DataChunk*.

template<typename T, typename std::enable_if< std::is_floating_point< T >::value, int

Read frames from dataChunk.

Return number of frames read

Parameters

- `[out] outBuffer`: Buffer to write the samples to
- `[in] frames`: Number of frames to read

uint64_t **tell**()

Tell the current frame position of the dataChunk.

Return current frame position of the dataChunk bool **`eof`**`()`

Check if end of data is reached.

Return `true` if end of data is reached and otherwise `false`

`class Bw64Writer`

BW64 Writer class.

Normally, you will create an instance of this class using *bw64::writeFile()*.

This is a RAII class, meaning that the file will be opened and initialized (required headers etc.) on construction, and closed and finalized (writing chunk sizes etc.) on destruction.

Public Functions

`Bw64Writer``(`**`const`** char **filename*, uint16_t *channels*, uint32_t *sampleRate*, uint16_t *bitDepth*, std::vector<std::shared_ptr<*Chunk*>> *additionalChunks*`)`

Open a new BW64 file for writing.

Opens a new BW64 file for writing, initializes everything up to the `data` chunk. Afterwards, you may write interleaved audio samples to this file.

If you need any chunks to appear *before* the data chunk, include them in the `additionalChunks`. They will be written directly after opening the file.

Warning If the file already exists it will be overwritten.

Note For convenience, you might consider using the `writeFile` helper function.

`~Bw64Writer``()`

Finalize file.

This destructor will write all yet-to-be-written chunks to the file and will also finalize all required information, i.e. the final chunk sizes etc.

uint16_t **`formatTag`**`()` **`const`**

Get format tag.

uint16_t **`channels`**`()` **`const`**

Get number of channels.

uint32_t **`sampleRate`**`()` **`const`**

Get sample rate.

uint16_t **`bitDepth`**`()` **`const`**

Get bit depth.

uint64_t **`framesWritten`**`()` **`const`**

Get number of frames.

bool **`isBw64File`**`()`

Check if file is bigger than 4GB and therefore a BW64 file.

void **`useRf64Id`**`(`bool *state*`)`

Use RF64 ID for outer chunk (when >4GB) rather than BW64.

uint32_t **`chunkSizeForHeader`**`(`uint32_t *id*`)`

Get the chunk size for header.

uint64_t **riffChunkSize**()
Calculate riff chunk size.

void **writeRiffHeader**()
Write RIFF header.

void **finalizeRiffChunk**()
Update RIFF header.

template<typename **ChunkType**>
void **writeChunk**(std::shared_ptr<*ChunkType*> *chunk*)
Write chunk template.

template<typename **ChunkType**>
void **overwriteChunk**(uint32_t *id*, std::shared_ptr<*ChunkType*> *chunk*)
Overwrite chunk template.

template<typename T, typename std::enable_if< std::is_floating_point< T >::value, int
Write frames to dataChunk.

Return number of frames written

Parameters

- `[out] inBuffer`: Buffer to read samples from
- `[in] frames`: Number of frames to write

CHAPTER

SIX

CHUNKS

class Chunk

RIFF chunk base class.

Subclassed by *bw64::AxmlChunk*, *bw64::ChnaChunk*, *bw64::DataChunk*, *bw64::DataSize64Chunk*, *bw64::FormatInfoChunk*, *bw64::JunkChunk*, *bw64::UnknownChunk*

Public Functions

virtual uint32_t **id**() **const** = 0

Get FourCC id.

virtual uint64_t **size**() **const** = 0

Get the size of the chunk.

virtual void **write**(std::ostream &*stream*) **const** = 0

Write the chunk to a stream.

class FormatInfoChunk : **public** bw64::*Chunk*

Class representation of a *FormatInfoChunk*.

Public Functions

FormatInfoChunk(uint16_t *channels*, uint32_t *sampleRate*, uint32_t *bitDepth*, std::shared_ptr<*ExtraData*> *extraData* = nullptr, uint16_t *formatTag* = 1)

Simple *FormatInfoChunk* constructor.

Parameters

- `channels`: number of channels
- `sampleRate`: sample rate of the audio data
- `bitDepth`: bit depth used in file
- `extraData`: custom *ExtraData* (optional, nullptr if not custom)
- `formatTag`: format tag, defaults to PCM

uint32_t **id**() **const**

Get FourCC id.

uint64_t **size**() **const**

Get the size of the chunk.

uint16_t **formatTag**() **const**
FormatTag getter.

uint16_t **channelCount**() **const**
ChannelCount getter.

uint32_t **sampleRate**() **const**
SampleRate getter.

uint32_t **bytesPerSecond**() **const**
BytesPerSecond getter.

uint16_t **blockAlignment**() **const**
BlockAlignment getter.

uint16_t **bitsPerSample**() **const**
BitsPerSample getter.

const std::shared_ptr<*ExtraData*> **extraData**() **const**
ExtraData getter.

void **write**(std::ostream &*stream*) **const**
Write the chunk to a stream.

class ExtraData
Class representation of the *ExtraData* of a *FormatInfoChunk*.

Public Functions

ExtraData(uint16_t *validBitsPerSample*, uint32_t *dwChannelMask*, uint16_t *subFormat*, std::string *subFormatString*)
ExtraData constructor.

uint16_t **validBitsPerSample**() **const**
ValidBitsPerSample getter.

uint32_t **dwChannelMask**() **const**
DwChannelMask getter.

uint16_t **subFormat**() **const**
SubFormat getter.

std::string **subFormatString**() **const**
SubFormatString getter.

class DataChunk : **public** bw64::*Chunk*
Class representation of a *DataChunk*.

Public Functions

uint32_t **id**() **const**
Get FourCC id.

uint64_t **size**() **const**
Get the size of the chunk.

void **write** (std::ostream&) **const**
Not to be used write chunk to stream.

Warning As the data chunk is usually not written in one piece the override for this function is not used. Using this method will throw an exception.

class JunkChunk : **public** bw64::*Chunk*
Class representation of a *DataChunk*.

Public Functions

uint32_t **id** () **const**
Get FourCC id.

uint64_t **size** () **const**
Get the size of the chunk.

void **write** (std::ostream &*stream*) **const**
Write the chunk to a stream.

class AxmlChunk : **public** bw64::*Chunk*
Class representation of an *AxmlChunk*.

Public Functions

uint32_t **id** () **const**
Get FourCC id.

uint64_t **size** () **const**
Get the size of the chunk.

void **write** (std::ostream &*stream*) **const**
Write the chunk to a stream.

class AudioId
Class representation of an *AudioId* field.

class ChnaChunk : **public** bw64::*Chunk*
Class representation of an *ChnaChunk*.

Public Functions

uint32_t **id** () **const**
Get FourCC id.

uint64_t **size** () **const**
Get the size of the chunk.

uint16_t **numTracks** () **const**
NumTracks getter.

uint16_t **numUids** () **const**
NumUids getter.

std::vector<*AudioId*> **audioIds**() **const**
AudioIds getter.

void **addAudioId**(*AudioId* *id*)
Add *AudioId* to *AudioId* table.

void **write**(std::ostream &*stream*) **const**
Write the chunk to a stream.

class DataSize64Chunk : **public** bw64::*Chunk*
Class representation of a DataSize64 chunk.

Public Functions

DataSize64Chunk(uint64_t *bw64Size* = 0, uint64_t *dataSize* = 0, std::map<uint32_t, uint64_t> *table* = std::map<uint32_t, uint64_t>())
DataSize64Chunk constructor.

uint32_t **id**() **const**
Get FourCC id.

uint64_t **size**() **const**
Get the size of the chunk.

uint64_t **bw64Size**() **const**
Bw64Size getter.

uint64_t **dataSize**() **const**
DataSize getter.

uint64_t **dummySize**() **const**
DummySize getter.

uint32_t **tableLength**() **const**
TableLength getter.

void **bw64Size**(uint64_t *size*)
Bw64Size setter.

void **dataSize**(uint64_t *size*)
DataSize setter.

void **dummySize**(uint64_t *size*)
DummySize setter.

const std::map<uint32_t, uint64_t> &**table**() **const**
Get table.

bool **hasChunkSize**(uint32_t *id*) **const**
Has chunkSize for id.

uint64_t **getChunkSize**(uint32_t *id*) **const**
Get chunkSize for id.

void **setChunkSize**(uint32_t *id*, uint64_t *size*)
Set or add a ChunkSize.

void **removeChunkSize**(uint32_t *id*)
Remove a ChunkSize from table.

void **clearChunkSizeTable**()
Clear ChunkSize table.

void **write**(std::ostream &*stream*) **const**
Write the chunk to a stream.

class UnknownChunk : **public** bw64::*Chunk*
Class representation of a custom chunk.

This class can be used to copy unknown chunks from an input file to an output file.

Public Functions

uint32_t **id**() **const**
Get FourCC id.

uint64_t **size**() **const**
Get the size of the chunk.

void **write**(std::ostream &*stream*) **const**
Write the chunk to a stream.

CHAPTER

SEVEN

UTILITIES

constexpr uint32_t `bw64::utils::`**fourCC**(char **const** *p*[5])

Convert char array chunkIds to uint32_t.

std::string `bw64::utils::`**fourCCToStr**(uint32_t *value*)

Convert uint32_t chunkId to string.

The libbw64 library is a lightweight C++ header only library to read and write BW64 files. BW64 is standardised as Recommendation ITU-R BS.2088 and the successor of RF64. So it already contains necessary extensions to support files which are bigger than 4 GB. Apart from that an BW64 file is able to contain the ADM metadata and link it with the audio tracks in the file. To do that a BW64 specifies two new RIFF chunks – the `axml` chunk and the `chna` chunk. To parse, create, modify and write the ADM metadata in the `axml` chunk you may use the libadm library.

CHAPTER EIGHT

FEATURES

- no dependencies
- support file sizes bigger than 4 GB (`ds64` chunk)
- read and write `axml` and `chna` chunks
- 16, 24, and 32 bit integer file formats

CHAPTER NINE

ACKNOWLEDGEMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 687645.

INDEX