# Low- and high-level audio programming using respectively C++ and PureData

This report demonstrates two approaches to program audio applications: a low-level approach using C++, the PortAudio API and the Libsndfile library, and a high-level approach using PureData and the OSC protocol. For the first part of the task, the low-level approach, we have implemented a record-and-playback program. For the second part of the task, the high-level approach, we have designed a mixer mixing four pre-recorded .wav files and a live input, with individual controls such as low- and high-pass filters, volume and reverb.

## Introduction

Audio programming comes with its own set of challenges. Working from the system level up, programs dealing with audio must conform to strict timing requirements – sound cannot simply wait for another process to finish. Working on a more abstract high level, programmers nowadays can avoid dealing with low-level concerns but have to figure out how to translate their creative ideas into code. In this project we created two programs to acquaint ourselves with some of the challenges, pitfalls and rewards of various programming approaches to audio.

### Implementing a low-level player and recorder

We created a basic-functionality audio player and recorder in C++ using the Portaudio library. C++ was appropriate for this project as it exposes the lower levels of abstraction needed to fully grasp what is going on under the hood. It was also a language that some members in our group already had familiarity with, ensuring we could focus primarily on audio paradigms as opposed to dealing with learning a new language. Portaudio was the audio library of choice because it provides an interface to audio programming that works across operating systems, allowing us to work on one operating system in the safety that the audio programming aspects will continue to work on other platforms.

### Why we chose PureData, OSC

Wdlkvjl wvhwk.

### Group workflow

Working as three people with different backgrounds meant allocating various tasks to different group members. None of us had much experience with system programming, but as Jonathan at least had some knowledge of C++ he took the first stab at the problem in order to get some of the boilerplate code in place. Meanwhile, Marcus and Nicolas began to get acquainted with PureData. As the project progressed, we used group meetings not just to track progress but to teach each other what we had learned, so that by the final weeks the two programs were a collaborative effort. Working separately at first and growing towards each other allowed us to take advantage of each others’ skills to learn more ourselves, and was more enriching than a simple divide-and-conquer would have been, or a project where we felt stuck with a technology we could not understand at all.

## Record and playback of audio using C++

### Overview of the main problem

In low-level audio programming, timing issues are paramount. In order to ensure smooth recording or playback without glitches, the program must deal with the audio device (be it microphone or speaker) with total regularity to avoid any breaks in the flow of samples. This means careful programming needs to avoid audio having to wait for a process to end, or for a resource to be freed by another thread. In the case of short segments of audio, a program can get away with storing audio in a buffer in memory, allowing for rapid access. This is not feasible as sound files grow longer, if one wants to access files on disk, or if the amount of memory is not yet determined (eg. when recording the live input of a microphone for an unspecified amount of time.)

Our program has to deal with two main issues. Firstly, we need to ensure that audio is served to or from the audio device within the appropriate time. Secondly, we need to interact with the filesystem to read from and write to disk, without interfering with the timing of the audio. We do both of these things by using the architecture Portaudio provides us with, as well as threading our program in a safe way.

### Structure of the program

Instead of forcing the problem into objects and classes, we approached this program from an imperative paradigm and divided various components of the problem into modules.

#### AudioWorker

The AudioWorker module functions as the main audio processing unit of our program. It is here that the most top-level functions are called, being *record\_file(char filename, SF\_INFO soundinfo)* or *play\_file(char filename, double startingpoint)*. By default, *record\_file* will record to a 16-bit mono PCM Wav file at 44.1 kHz, but these parameters can be changed by adapting the *SF\_info* struct (which comes from the libsndfile library). By default, *play\_file* will start playing from the beginning of the sound file, but given a *startingpoint* will seek in the file and begin playing at that moment. Another function exposed in AudioWorker’s header file is *record\_short*, which was simply intended to record a five-second buffer for testing purposes.

While the interface of AudioWorker is intentionally simple, this module is where the main usage of Portaudio lies. In order to get through audio with the library, we have to register an audio callback. This is a function with a signature specified by Portaudio, which will be called by the engine whenever the operating system makes a call for more audio. It is in this function that time is crucial, as it is called regularly by the operating system’s interrupt handler. Therefore, certain operations are forbidden, such as file IO, mutex operations, or even most other Portaudio API calls. We deal with these issues by keeping the audio callback itself as clean and simple as possible: we read or write from a buffer in memory, avoiding locks or file IO operations entirely; and we do only the most essential real-time audio processing (in this case, the numerical computations for the level meter.)

#### Ringbuffer

Keeping work in the callback to a minimum works without an issue while the buffers are small enough to keep entirely in memory (such as in *record\_short)*. However, when files get too long to load entirely into memory, or if recording for an indeterminate amount of time, we need to do some extra work to ensure the timeliness of the audio callback. Instead of providing a simple array to the callback, we provide it with a ringbuffer, or circular buffer. This data structure acts as a FIFO-queue, allowing pushes on one end while popping from the other. Its implementation depends on a read and a write pointer cycling through a buffer of fixed size, wrapping around the end of the buffer. This setup, with data being continuously streamed into the same area of memory, makes it particularly appropriate to deal with audio, as it is conceptually easy to understand how audio fills the buffer and leaves it again in a constant stream.

While Portaudio provides a circular buffer implementation, it is not by default exposed and required a lot of internal dependencies to get working. Instead, we opted for TPCircularBuffer, by Michael Tyson.[[1]](#footnote-1) This data structure uses a very basic interface: *TPCircularBufferHead* and *TPCircularBufferTail* returns a pointer to the next data to write and read respectively, and *TPCircularBufferProduce* and *TPCircularBufferConsume* perform the respective write and read operations, freeing the memory as they go along. An important feature of this data structure as it is implemented is that it is thread-safe with one producer and one consumer. This is crucial as it allows both the audio callback to work with the CircularBuffer on one end while at the other end a separate thread performs slower File IO operations.

#### FileIO

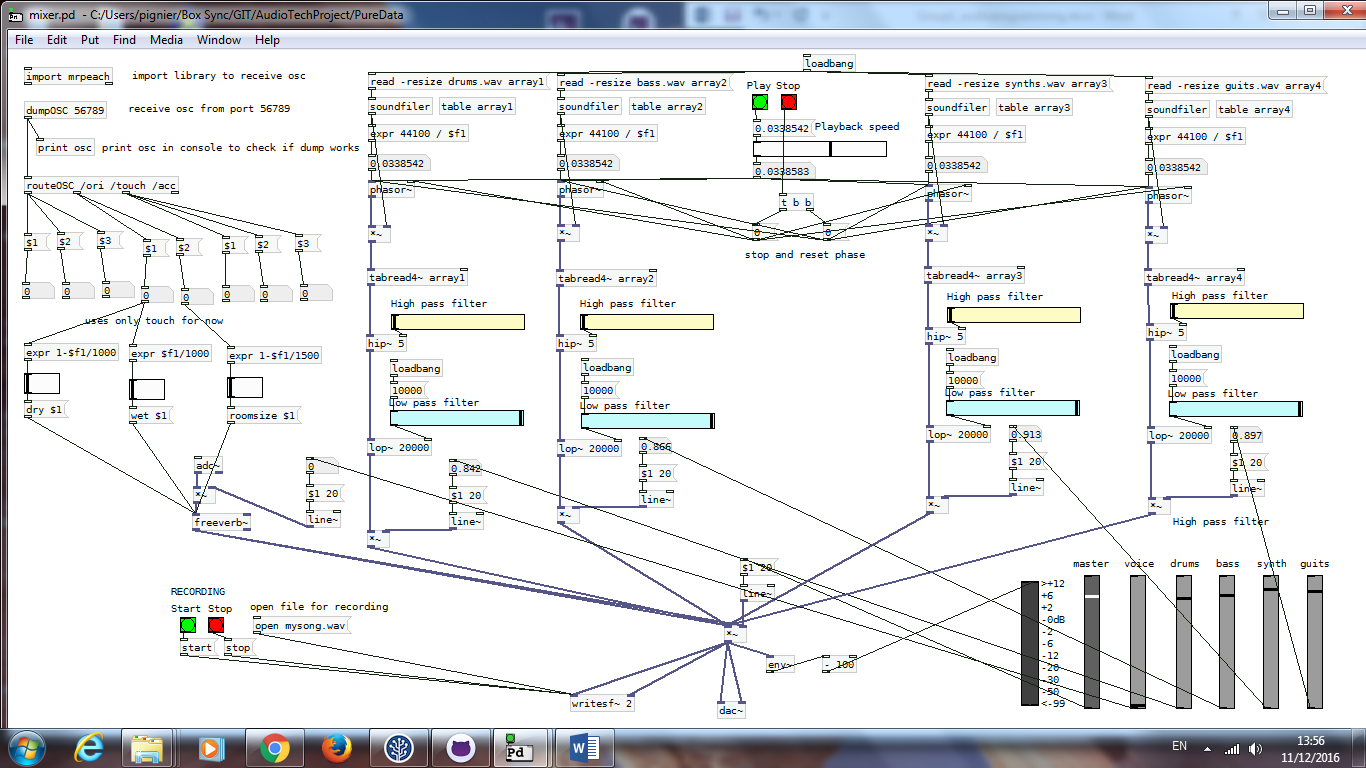
This module provides the bridge between our program and the file system. It mainly derives its functionality from the library libsndfile, which allows for a number of different file formats to be processed. Many of the FileIO functions wrap neatly around libsndfile functions, such as *file\_open, read\_from\_file, write\_to\_file,* and *seek*. Additionally, though, there are two functions, *read\_file\_threadworker* and *write\_file\_threadworker* which are ready to act as the second player in the audio callback scheme. These take in an AudioFile (a struct keeping together SNDFILE pointer and SF\_INFO struct from libsndfile) and a buffer and will write one to the other (depending on whether we are reading or writing to or from disk). In practice, AudioWorker will initialize a thread with this function, handing it the ringbuffer as its buffer argument. If we’re reading from a file, then the AudioFile will be the already opened file on disk; if we’re writing, this will be a temporary file opened for writing. Handling the reading and writing to and from file in a different thread, interacting with the audio thread through a thread-safe data structure relaxes the time requirements on the file IO operations. We have no idea how slow these will be in practice, but at least now we have faith in knowing these operations will not hold up the audio callback.

In order to avoid a redundantly high number of write operations to the filesystem, we

#### Interface

#### LevelMeter

## Programming a mixer in PureData with live remote controls via OSC



## Possible improvements and conclusion

## Code

All the code can be found at <https://github.com/jonathanadam/AudioTechProject> .

# Using the template

This chapter describes how the template works, and how you are supposed to use it. The content is mostly self-referential. The template contains two chapters, but the article should be one chapter. So: under the chapter heading there should be a short summary of what the article is about, in one paragraph. Support mechanisms in Word such as *styles*, *captions*, *cross-references*, and automated numbering should be used throughout. This summarising paragraph has the *style* **Chapter introduction**. The chapter heading (the title of your article) must have *style* **Heading 1**, or the numering will not work.

## Section headings have style Heading 2

This template file is called AudioTemplate2015.dot.

### Subesection headings have style Heading 3

This is body text. The first paragraph after a heading should have style **Normal no indent**. This gives a straight left margin on the first line. Note that **Normal no indent** is automatically selected after **Heading 3**.

Body text in the following paragraphs use the style **Normal**. It is selected automatically after **Normal no indent** when you enter a new paragraph (with Enter). Body text has font Garamond, size 11. But, always let the styles control the formatting: font, size, indent, alignment, margins, line spacing etc. Do not format the text yourself, unless the text requires a special typography. If you do, it is much harder to reformat the text later.

### Numbering of items in the text

With automatic numbering of the document it becomes much easier to reuse the articles. So please let Word assign the numbers wherever necessary: sections, figures, tables etc. Don’t number anything manually. This actually makes the writing easier too.

Use **Insert | Caption…** to number figures, tables and exercises. This makes it easy to have the chapter number follow. Every paragraph or inserted object (figures etc) in Word can be equipped with a caption, as follows.

1. Select the paragraph or the object with the mouse.
2. Choose **Insert | Reference… > Caption**.
3. Choose **Figure** or **Table**. Alternatives in Swedish are also found in this template.
4. Press the button **Numbering** and select **Include chapter number**, and choose Arabic numerals throughout. Choose **OK**.

The paragraphs below have been captioned in this way.

Fake figure

Figure 2‑1. Example of an automatically numbered figure and its associated caption text.

|  |  |  |
| --- | --- | --- |
| Fake table |  |  |
|  |  |  |

Table 2‑1. Example of an automatically numbered table, and its associated caption text.

Example 2-1: This is an automatically numbered example. I positioned the cursor at the start of the paragraph, without marking any text. Then I chose **Insert | Reference… > Caption** and selected **Example** to insert a number. Then I applied the paragraph formatting style **Normal no indent** to avoid the boldface font in style **Caption.**

### Numbering of equations

The Caption function is not practical for numbering equations, because a caption is always placed above or below the object to which it refers. Equation numbers should be right-aligned. Below we have an ”equation paragraph” that makes this easier. To insert an equation with numbering on the form *chapternr*.*sequencenr*, follow these steps:

1. Copy the paragraph with the equation, below. To select the whole paragraph, double-click in the left margin.
2. Paste the paragraph at the place where you want the equation.
3. Double-click the equation itself. It is an embedded object, that invokes the Microsoft Equation Editor.
4. In the equation editor, change the equation to what it should be.
5. Click outside the equation. This closes the Equation Editor and returns you to the document.

 (2.1)

The *style* of the above paragraph with the equation is **Numbered equation**. Its formatting contains center tab, right tab, and space above and below the equation. The parnetheses contain a cross-reference to the current chapter number, a full stop, and a field code { **seq** ekv \\* Arabic \s 1 }. The option "\s 1" restarts the numbering of the sequence "ekv" every time a paragraph with style **Heading 1** is encountered, that is, the chapter heading. The *style* of this gray-panel paragraph is **In depth** and is intended for advanced material or for curiosities.

Thus, we use the function “numbered sequence” to number equations. To apply automatic numbering to other items, use **Insert** | **Field** -> **Numbering** -> **seq** *sequencename* . The sequence name is something you make up yourself.

### Cross-references

To reference figures, tables and equations, use **Insert | Reference… > Cross-reference**. To reference an unnumbered object, attach a so-called **Bookmark** to the object, and insert a cross-reference to this Bookmark. Then the pagination of the reference will be updated automatically.

### Footer

The page footer is generated automatically; do not edit it directly. Till vänster visas kursnamnet (se **Error! Reference source not found.**). The chapter heading is displayed centered in the footer. The page number preceded by the chapter number is shown to the right. The chapter number will stay at 1 until your article is combined with the others.

### Header

The page header is not intended for display but only as a convenience to those editing the document. The Header is generated automatically. Before printing the file, erase the contents of the Header, or mark it as Hidden Text (Ctrl-Alt H).

### Symbols as reading directives



Little symbols in the margin are nice, but they can be difficult to put in place. This template contains a trick for doing it. Follow these steps:

1. Choose an icon in Tabell 2-1.
2. Make a new paragraph that contains only the corresponding ordinary character in the third column.
3. Place the cursor next to the character.
4. Press Ctrl+§ (the paragraph character near top left on the keyboard).

The keystroke Ctrl-§ is configured to apply the style**Icon at left** to the current paragraph. The paragraph font is Wingdings. To get other icons from the Wingdings font, choose Insert Symbol with the font Wingdings, instead of steps 1 and 2 above.

|  |  |  |  |
| --- | --- | --- | --- |
| Icon | Directive | Char | ASCII |
| 👓 | Advanced | $ | 36 |
| 🕯 | Illuminating | ' | 39 |
| ☝ | Definition | G | 70 |
| 🕱 | Pitfall | N | 77 |

Tabell 2-1: Icons used for reading directives.

## Adapting existing texts

### Applying this template to an existing document

If you have a document that uses the styles Normal, Heading 1, Heading 2, and Heading 3; where Heading 1 is the document heading, then this might work:

1. Open the existing document.
2. Choose **Tools | Templates and Add-Ins… > Templates**
3. Check **Automatically update document styles**.
4. Choose **Attach…**
5. Browse to the file AudioTemplate.dot .

### Pasting existing text from another document

The template settings can easily be upset if text is pasted from other documents. To be on the safe side, don’t use the ordinary **Edit | Paste** (Ctrl-v) but always choose **Edit | Paste special…** > **Unformatted text** . You will have to reapply character formatting, but otherwise undesired styles may be imported from the other document.

### Pre-defined paragraph styles

These styles are specially (re-)defined in AudioTemplate.dot:

|  |  |
| --- | --- |
| **Style name** | **Intended for** |
| Heading 1 | Chapter heading |
| Heading 2 | Subheading |
| Heading 3 | Sub-subheading |
| Chapter introduction | The summary following the chapter heading |
| Normal no indent | Body text in the first paragraph after figures or tables |
| Normal | Body text |
| Numbered equation | Paragraph containing a numbered equation only |
| In depth | Advanced material or curiosity |
| Icon at left | Symbols in the margin |
| Console | Console typing that is entered by the user or is displayed on a screen, programme code. |
| Header | Page header |
| Footer | Page footer |
| Caption | Figure and table captions |
| Reference | Literature references |
|  |  |

1. M. (2016). A simple, fast circular buffer implementation for audio processing. Retrieved December 09, 2016, from https://github.com/michaeltyson/TPCircularBuffer [↑](#footnote-ref-1)