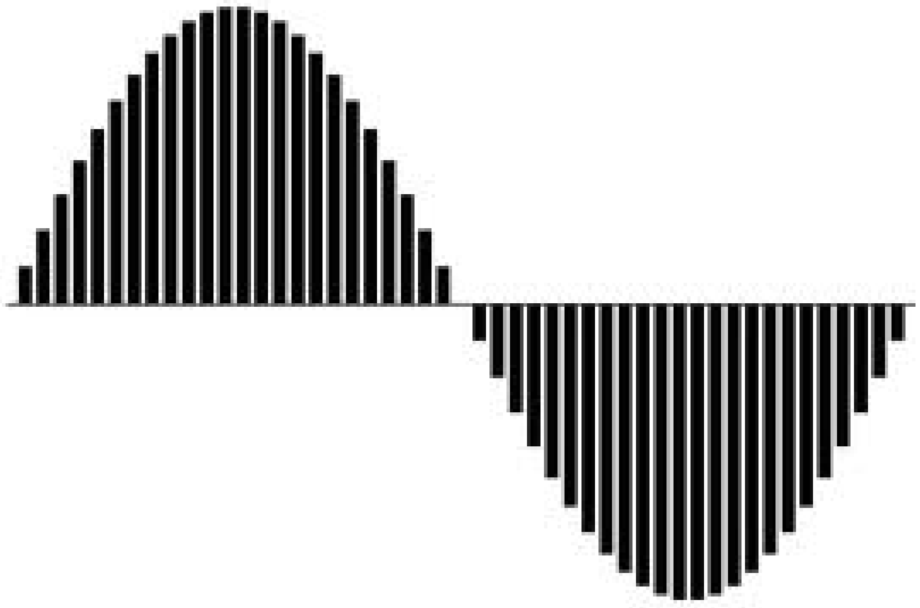

DT2410: Audio Technology

Audio Programming with C#, PortAudio and SuperCollider

Jesper Annebäck	anneback@kth.se
Sam Lewis	svrlewis@kth.se
Jesper Norberg	jenor@kth.se

Examiner: Sten Ternström

Abstract

The following project contains the development process of a C# program using mainly PortAudio, but also Wav-encoding from an external library called NAudio. The program uses callback function and a double buffer to read and write a sound file. Focus was on understanding the fundamentals of how sound is handled in the computer by the double buffer. The paper also shows how you can use the high-level programming language SuperCollider for manipulating sound in real-time.

Contents

1 Introduction

1.1 Goal

The task is divided into two parts A and B. The first part is to implement a program that can play a sound file using a double buffer, show the dB-levels in a intuitive way and also record a sound file while checking for available disk space. The other part is to modify a signal in real time in some interesting way, using a high-level programming language.

1.2 Acknowledgements

Due to conflicting schedules, we have split the project into three main areas, which we have then divided up between us. The main areas were: Task A - Sam Lewis, Task B - Jesper Norberg, the Report - Jesper Annebäck. Everyone has participated in each of the three areas, but the designated person has done more than the others in his specific area.

2 Background

2.1 Double Buffer

The computer system can only do one thing at a time, e.g. when it reads a file from the hard disk it can only do that. In order to read (playback) and or write (recording) to a sound file correctly the computer needs to read the samples from the hard disk, which takes time, at the same time as it passes them to output. This can be done with a double buffer system, see Figure 1. The input

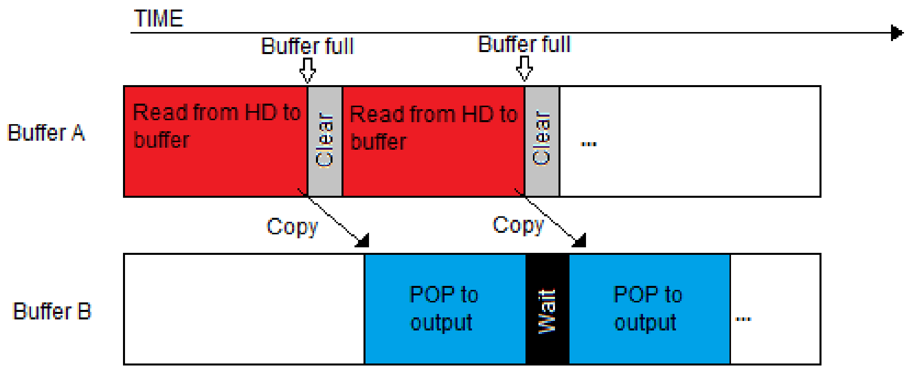

Figure 1: A simple double buffer system.

buffer (Buffer A) reads from the hard disk (HD) to its memory address on the ram, until it is full. When the system announces that the buffer is full it copies it to the output buffer (Buffer B) which POPs the data to the next component in line. The POP is rather fast so the output buffer will need to wait for a new full buffer, shown i Figure 2.[1]

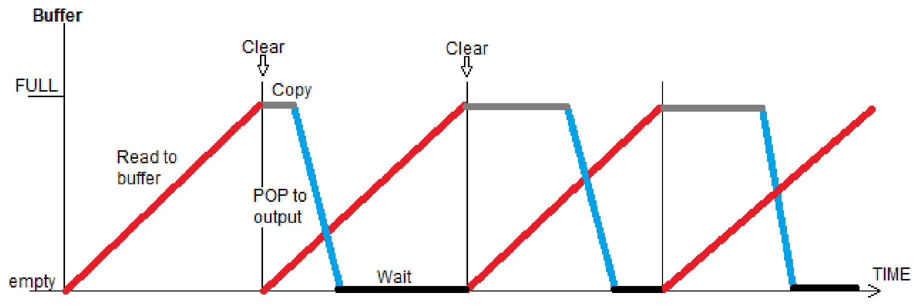

Figure 2: The buffers' content over time.

In the context of audio programming, double buffers are needed to ensure smooth playback. By being ahead of the output stream by a single buffer there is some level of insurance in case the processor needs to perform another task while the audio is being output. As long as the output does not catch up to the second buffer then playback will remain constant and smooth. In practice, more than two buffers can be used to improve reliability at the cost of memory.

2.2 Callback Function

A callback function is a piece of executable code that is passed as an argument to another function in the program. The callback function's purpose is to not *hold up* the program execution, while one function is running. It is used when a program needs to execute different function base on its current state.

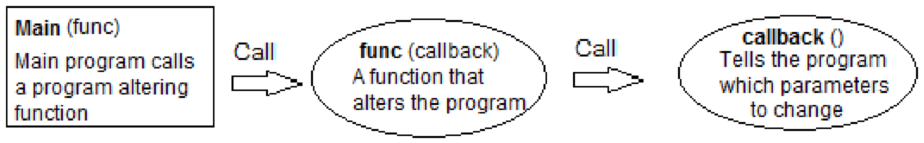
Main (func)
Main program calls a program altering function
Call
func (callback)
A function that alters the program
Call
callback ()
Tells the program which parameters to change

Figure 3: The Main program will be on hold since the altering function will only be executed when the callback function is invoked.

For our task the callback function is essential. This is as audio needs to be played at a constant rate and in a precise order. By having a callback function that fires at a high priority when audio is needed, it ensures that the playback is smooth and glitch free.

3 Method

Since Java is not good for handling the buffer process, the language of choice for the audio input and output program, the first task, was C# and the cross-platform open source library PortAudio[2] and also NAudio[3]. C# syntax's is pretty similar to Java's and made the decision easy, also one group member already had decent experience with it. PortAudio is a library which allows developers to use simple callback functions for recording and or playback audio.

In order to create the program the IDE of choice was the Microsoft Visual Studio. In order to make the whole group develop the program simultaneously and time independent Github[4] was used for revision control.

For the second task the high-level programming language SuperCollider[5] was used. A bit of pre-study was needed since none of the members had work with it before. Good examples and documentation could be found on the developer's website.

Most of the early work of the assignment was developed in study session, were the group sat down and programmed together. Further into the course the work was divided among the members, due to very conflicting schedules.

4 Results

4.1 Implementation

4.1.1 C# and PortAudio: Program for Task A

The whole code is available in the Appendix. Our Main program is invoked when a user selects a wave file from the program. This creates a *wavFile* object.

When the *wavFile* object is played it splits into two threads, in the first thread there is a 'busy loop' to load blocks of samples from disk into a queue for the callback function to read out of. The busy loop should be performing work when there is not anything else to do. In this way, we can ensure that audio is processed and ready to be played ahead of when it is needed to be output.

The average dB value of the sample blocks is also calculated in the busy loop. This could also be calculated in the callback function however if, for whatever reason, it were to run slowly then this could cause glitches in the audio output. Care is also taken to assume that the busy loop can only get a certain amount of samples ahead of the samples that are being played. This is to restrict the amount of memory used by the program.

The second thread simply is created simply for the callback function to run in. This needs to be in a separate thread so that the GUI thread can still run and not block while the callback or busy loop are performing work.

For the recording, we approached the problem in the same way we approached playback, but in reverse, so to speak. It proved to be unsufficient, and as such we had to get down to a more low-level approach. At the time of writing, we are able to receive and display the mic input, but unfinished with the handling of the file creation.

4.1.2 SuperCollider: Program for Task B

The whole code is available in the Appendix. The code can be executed from the SuperCollider IDE only, by calling on each related row. The program will then alter mic input in real time.

We wanted to implement classic sound alteration effects, in order to demonstrate the potential of the language. To accomplish this we used both in the language pre-defined sound alteration functions, as well as implementing our own.

In our implementation we have implemented the effects Clipping, Folding, Reverb, Flanger and Granulation, as well as various combinations of the previous. Each effect has its own "function".

5 Discussion

Since our backgrounds in the field differed, our experiences with the software tools were also different. We had some amount of struggle with ensuring that everyone contributed the same amount. Another factor that also affected our work was the different personal study schedules, it was quite hard to plan for study sessions further into the course.

Working with a GUI proved initially to be difficult. When first written, the GUI was inaccessible when a sound was playing. This is as the busy loop that read the wav file from disk was running in the same thread that the GUI was displayed in. This was fixed by splitting the busy loop and GUI parts of the program into separate threads.

Finding the right size of the queue to send sound buffers to the callback event also was challenging. At first, a small queue size was implemented in order to both constrain the memory use of the program and also to limit the overhead time needed to fill the queue when a song is initially started. However, a small queue size caused the audio output to be non constant and glitchy. This was easily fixed by increasing the queue size.

6 Conclusion

The programming assignment gave a very good perspective of how audio I/O works on a lower level in a computer. The whole procedure is kind of a reinvention of functions that already exists in many of the sound libraries that can be found today.

Writing software at this lower level gives a greater amount of control in the way in which it is output but also adds another level of difficulty in writing the software. Design decisions need to be made with an extra amount of care as audio needs to be output in a precise and on-time manner. The consequences of putting a block of code that takes a long time to run in the wrong place can be disastrous.

We are overall happy with our work. Each member worked well and the end result is, though not perfect, pretty close to home. Given more time, we would have finished the file handling of the recording as well.

7 References

[1] Ternström, Sten. Professor of Music Acoustics, Kungliga Tekniska högskolan. Stockholm, Lecture: *Software architectures for audio, part1.* 2013-12-06.

[2] PortAudio. http://www.portaudio.com/ (viewed 2013-12-11)

[3] NAudio. http://naudio.codeplex.com/ (viewed 2013-12-11)

[4] Github. http://github.com/ (viewed 2013-12-11)

[5] SuperCollider. http://supercollider.sourceforge.net/ (viewed 2013-12-11)