Appendix D: An Introduction to Real-time Audio IO with PortAudio

Victor Lazzarini

Most modern operating systems will provide Application Programming Interfaces[1] (APIs) for accessing audio devices (soundcards) present in the system for input/output (IO). For instance, on Windows we have the MME (Multimedia Extensions) and DirectX, among other APIs, for this purpose; on OS X, these services are offered by the CoreAudio framework; and on Linux, the most common 'backend' API is ALSA (Advanced Linux Sound API). In addition, we also have other higher-level cross-platform APIs for audio IO that often are built on top of the basic sound systems, such as the Jack Connection Kit (Linux and OS X, mostly), PulseAudio (Linux and Windows) and PortAudio (OS X, Windows and Linux).

The major drawback of writing programs using a platform-specific API is that portability becomes an issue, often involving major re-implementation of the audio IO parts. For this reason, using cross-platform APIs is often a good idea, even though sometimes using the backend API might allow a slight better performance. This text will introduce some basic aspects of realtime audio processing using the PortAudio API for audio IO. PortAudio is part of a set of cross-platform libraries for media, PortMedia and it can be downloaded and installed from http://portmedia.sourceforge.net/.

D.1 Preliminaries

Programming real-time audio, using whatever API, has a number of basic elements that should be considered first. This book already discusses all of these elements in great detail, so we will not need to cover them here. However it is good to remind us about a few things that will be heavily in use here.

First of all, what form does the data we are manipulating take? Basically, audio signals will be a stream of individual numbers, *samples*, produced by the soundcard at a rate of *sr* (sampling rate) *samples per seconds* (for each audio channel). Each one of these will be a representation of the amplitude of a waveform at a given time and they will be encoded into a certain data type, that can be a single byte, a short, a int, or a floating-point (single or double-precision). Audio samples of more than one channel are grouped into a frame, interleaved, so for multichannel, we will have *sr* frames of samples per second (instead of single samples). Our job will be to take in blocks of frames (or single samples) from the soundcard in an array, do something with them and send an array of processed frames to the soundcard for output. The size (number of frames) in this buffer will be linked to the IO latency, i.e. the time it takes for the sound to get in and out of the computer. In general, we will try to make this buffer as small as possible, but this will depend on the system. With smaller buffers the program might start to drop samples ('drop-outs') and we will get clicks and interruptions in the audio stream.

The previous paragraph is a summary of probably everything we need to know to start programming real-time audio IO with the tools we will study here. We will need to set the parameters for our signal (sampling rate, encoding and number of channels), reserve some memory to place the audio frames (arrays of data) and process the audio sample by sample. Generally speaking, after initialisation, we will open the soundcard for IO using the parameters

described above. PortAudio will then allow us to start a data stream from/to the soundcard. When we are done, we will just stop the stream, close the device and terminate the PortAudio session.

D.2 Programming with PortAudio

We will now go through the basic steps of using the PortAudio API. As with other such IO systems, this will take four steps: initialisation, device opening, IO operations and termination.

D.2.1 Initialization

The PortAudio library requires a single header to be placed at the top of the program:

```
#include <portaudio.h>
```

Before its use, the library needs to be initialised with a call to `Pa_Initialize()`. This returns an error code of the type `PaError`:

```
PaError err;
...
err = Pa_Initialize()
```

The error code is equal to the constant `paNoError` if the operation was successful. Otherwise the error string can be retrieved with `Pa_GetErrorText(err)`:

```
printf("%s \n", Pa_GetErrorText(err));
```

D.2.2 Opening the Devices

Before we open a device, we can use a function to discover and list all devices in the system; the following code fragment performs this:

```
ndev = Pa_GetDeviceCount();
for(i=0; i<ndev; i++){
  info = Pa_GetDeviceInfo((PaDeviceIndex) i);
  if(info->maxOutputChannels > 0) printf("output device: ");
  if (info->maxInputChannels > 0) printf("input device: ");
  printf("%d: %s\n", i, info->name);
 }
```

A ‘logical’ device (i.e. the one that gets listed) can be either uni or bi-directional. We can open bi-directional devices for input and/or output, but we cannot open a uni-directional input device for output and vice-versa, as the logic dictates. A single hardware device might get listed several times, uni and bi-directionally, as different logic devices. As PortAudio can sometimes

use more than one backend API, devices can often get listed multiple times (for each of these backends).

After devices are chosen for input and output, we will proceed to set the parameters for the IO data streams, which will include the selected devices. Filling in some fields in a PaStreamParameters structure, as shown below does this:

```
PaStreamParameters inparam, outparam;
...
    memset(&inparam, 0, sizeof(PaStreamParameters));
    inparam.device = devin;
    inparam.channelCount = 1;
    inparam.sampleFormat = paFloat32;

    memset(&outparam, 0, sizeof(PaStreamParameters));
    outparam.device = devout;
    outparam.channelCount = 1;
    outparam.sampleFormat = paFloat32;
```

These stream parameters are set separately for input and output. We start by initialising all the structure fields to 0 (using `memset()`). Then we set the chosen device number (`devin`, `devout`), the number of channels and the sample data type. For generality and simplicity, we will use a single-precision floating-point number to hold a sample (`paFloat32`), and leave PortAudio to select the IO precision depending on the soundcard and platform. This also means that the maximum amplitude of our digital signal will always be 1.0 (0dB full-scale). In addition to the stream parameters used here, there are other fields in the PaStreamParameters data structure. These however are not required for our current purposes (you can check them out in the `portaudio.h` header file).

Once the stream parameters are decided, we can proceed to open the streams. We can open both input and output streams with one single function call:

```
PaStream *handle;
...
err = Pa_OpenStream(&handle,&inparam,&outparam,SR,BUF,paNoFlag,NULL,NULL);
```

If successful, the open stream is represented by a handle[2] (the first parameter). The second and third parameters are used for the input and output parameter streams, respectively. If one of them is `NULL`, only one direction is open. Then we have the sampling rate `SR` and the data buffer size `BUF` in frames. This is followed by an optional IO flags (`paNoFlag` for no flags). The last parameters are reserved for one of the two possible modes of IO: asynchronous (or callback[3]-based) and blocking[4]. If the former is used, then we will pass a callback function here and a pointer to some user data as the last parameter. For blocking mode, these parameters are set to `NULL`. If the streams are successfully opened, we can start streaming with the call:

```
err = Pa_StartStream(handle);
```

D.2.3 Blocking IO

The blocking mode is possibly the simplest to understand and implement. It works in a very similar form to file IO: two functions are provided, that will take a buffer and read/write this data from/to a device. The functions will block if there is no data to be read or if the writing cannot be straight away. These functions are used as follows:

```
float buf[BUF];
...
err = Pa_ReadStream(handle, buf, BUF);
err = Pa_WriteStream(handle, buf, BUF);
```

For instance, a loop to read data from a soundcard and write it back without modification can be coded like this (`END` is the duration in seconds):

```
while(Pa_GetStreamTime(handle) < END){
    err = Pa_ReadStream(handle, buf, BUF);
    if(err == paNoError){
       err = Pa_WriteStream(handle, buf, BUF);
       if(err != paNoError)  printf("%s \n", Pa_GetErrorText(err));
     } else  printf("%s \n", Pa_GetErrorText(err));
 }
```

The function `Pa_GetSreamTime()` can be used to check the current time of a particular stream. We use it here to keep the processing loop going for the desired duration.

D.2.4 Callback IO

Callback IO is a little more involved. It requires the programmer to supply a function that will be called by PortAudio whenever data is available for IO. The form of the callback is:

```
int audio_callback(const void *input, void *output, unsigned long frameCount,
               const PaStreamCallbackTimeInfo *timeInfo,
               PaStreamCallbackFlags statusFlags, void *userData)
```

The callback will be passed the input and output data arrays (of whatever type the stream was opened) as the first and second parameters, both of which will contain frameCount frames (the next parameter). It will also be passed a time stamp data structure containing the current time, as well as the input/output data time references (which are going to depend on the audio latency). The next parameter is a series of stream status flags, which can be checked to ascertain the status of the data stream. The last parameter is the location of the user data[5] passed to the `Pa_OpenStream()` function. The equivalent code to the blocking IO shown in the previous section would be (note that in this case we do not have any need for user data, so we pass `NULL`):

```
int audio_callback(const void *input, void *output,unsigned long frameCount,
               const PaStreamCallbackTimeInfo *timeInfo,
               PaStreamCallbackFlags statusFlags, void *userData){

  int i;
  float *inp = (float *) input, *outp = (float *) output;
  for(i=0; i < frameCount; i++) outp[i] = inp[i];
  return paContinue;

}
```

with a stream open as this:

```
err = Pa_OpenStream(&handle,&inparam,&outparam,SR,BUF,paNoFlag,
      audio_callback, NULL);
```

The program will also require that we make sure it keeps running for the required duration in seconds. So we have to place a time-counting loop that just waits until we reach the requested end time (`END` now is the duration in seconds), using `Pa_GetSreamTime()` to check it:

```
while(Pa_GetStreamTime(handle) < END);
```

If we had not included this loop, the program would just finish immediately, as there is nothing preventing or blocking the execution. PortAudio calls the callback function, so we do not have direct control of it.

D.2.5 Finalizing

When we are done, we just need to stop and close the stream. If we are ending our program, we also terminate the PortAudio session:

```
Pa_StopStream(handle);
Pa_CloseStream(handle);
Pa_Terminate();
```

D.3 A Full Example: Echo

We will now present a real-time audio processing example, an echo program using a very common DSP structure, the comb filter (Figure D.1). Programming a comb filter is relatively easy; all we need is some memory for the delay (an array) and a read/write pointer that will access it circularly. We place the audio data in the delay and retrieve at the other end, feeding it back into the delay line. This is shown in the following code fragment:

```
float delay[SR/2], out = 0.f;
```

```
...

 for(j=0; j < BUF; j++) {
  out = delay[rp];
  delay[rp++] = buf[j] + out*0.5;
  if (rp == SR/2) rp = 0;
  buf[j] = out;
}
```

In the example above, our delay is SR/2 samples long, or ½ second. The comb filter algorithm first writes output sample into a temporary variable, then it fills in the delay position vacated by the output sample (with an input sample and the fed-back output) and increments the read/write index. This is wrapped-around if it reaches the end of the delay line. Finally the output buffer (which is the same as the input one) is filled with the current output sample. The feedback gain is 0.5, which means that each echo will be ½ of the amplitude of the previous.

We will present two versions of this example, one using the blocking IO and the other using the callback interface. They should perform quite similarly, but this might vary from platform to platform. Also the buffer size BUF can be adjusted according to the platform/backend API used. If drop-outs are heard (likely with Windows & MME), it can be increased; for lower latency it can be decreased, if your system allows.

The major difference in the code is of course the presence of the callback in the second example. Because we will do the processing in the callback, the delay line and read/write index will have to be passed to it as *user data*. We will place these two elements in a data structure, which will be dynamically allocated in the main program, then we will pass the pointer to it as the last argument to the Pa_OpenStream() function. The callback will then be able to use it in its processing. In the blocking example, instead, everything we need is inside the main function.

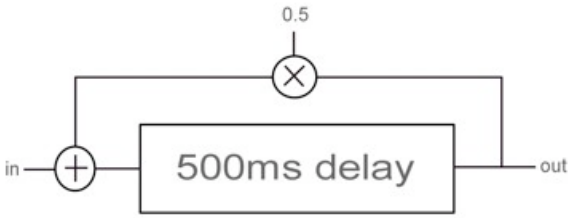

Figure D.1 A conceptual flowchart for the *echo* program.

Compiling these examples is very simple. Assuming PortAudio is installed, as default in `/usr/local/lib`, with headers in `/usr/local/lib`, we have the following command (on all systems):

```
gcc -o echo echo.c -I/usr/local/include -L/usr/local/lib -lportaudio
```

D.3.1 Blocking Example

```
#include <stdio.h>
#include <string.h>
#include <portaudio.h>
```

```
#define SR 44100.
#define BUF 512  /* increase this if you have drop-outs */
#define END 60


int main(int argc, char** argv){

  PaError err;
  PaDeviceIndex devin,devout, ndev;
  const PaDeviceInfo *info;
  PaStreamParameters inparam, outparam;
  PaStream *handle;
  int i, rp = 0;
  float buf[BUF], out = 0.f;
  float delay[(int)SR/2];

  memset(&delay,0,sizeof(float)*SR/2);
  err = Pa_Initialize();
  if( err == paNoError){
    ndev = Pa_GetDeviceCount();
    for(i=0; i<ndev; i++){
      info = Pa_GetDeviceInfo((PaDeviceIndex) i);
      if(info->maxOutputChannels > 0) printf("output device: ");
      if (info->maxInputChannels > 0) printf("input device: ");
      printf("%d: %s\n", i, info->name);
    }

    printf("choose device for input: ");
    scanf("%d", &devin);
    printf("choose device for output: ");
    scanf("%d", &devout);

    memset(&inparam, 0, sizeof(PaStreamParameters));
    inparam.device = devin;
    inparam.channelCount = 1;
    inparam.sampleFormat = paFloat32;
    memset(&outparam, 0, sizeof(PaStreamParameters));
    outparam.device = (PaDeviceIndex) devout;
    outparam.channelCount = 1;
    outparam.sampleFormat = paFloat32;

    err = Pa_OpenStream(&handle,&inparam,&outparam,SR,BUF,paNoFlag,
                  NULL, NULL);
    if(err == paNoError){
      err = Pa_StartStream(handle);
      if(err == paNoError){
      while(Pa_GetStreamTime(handle) < 60){
        err = Pa_ReadStream(handle, buf, BUF);
        if(err == paNoError){
            for(i=0; i < BUF; i++) {
                    out = delay[rp];
                      delay[rp++] = buf[i] + out*0.5;
                  if (rp >= SR/2) rp = 0;
                      buf[i] = out;
          }
          err = (int) Pa_WriteStream(handle, buf, BUF);
          if(err != paNoError)  printf("%s \n", Pa_GetErrorText(err));
```

```
        } else  printf("%s \n", Pa_GetErrorText(err));
      }
      Pa_StopStream(handle);
      } else  printf("%s \n", Pa_GetErrorText(err));
      Pa_CloseStream(handle);
    } else  printf("%s \n", Pa_GetErrorText(err));
    Pa_Terminate();
  }  else  printf("%s \n", Pa_GetErrorText(err));

  return 0;
}
```

D.3.2 Callback Example

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <portaudio.h>

#define SR 44100.
#define BUF 512  /* increase this if you have drop-outs */
#define END  60

typedef struct _mydata{
  float delay[(int)SR/2];
  int rp;
} mydata;


int audio_callback(const void *input, void *output,
               unsigned long frameCount,
               const PaStreamCallbackTimeInfo *timeInfo,
               PaStreamCallbackFlags statusFlags,
             void *userData){


  mydata *p = (mydata *)userData;
  int i, rp = p->rp;
  float out, *delay = p->delay;
  float *inp = (float *) input, *outp = (float *) output;
  for(i=0; i < frameCount; i++){
    out = delay[rp];
    delay[rp++] = inp[i] + out*0.5;
    if(rp >= SR/2) rp = 0;
    outp[i] = out;
  }
  p->rp = rp;
  return paContinue;

}

int main(int argc, char** argv){
```

```
  PaError err;
  PaDeviceIndex devin,devout, ndev;
  const PaDeviceInfo *info;
  PaStreamParameters inparam, outparam;
  PaStream *handle;
  int i;
  mydata *data = (mydata *) calloc(sizeof(mydata),1);

  err = Pa_Initialize();
  if( err == paNoError){
    ndev = Pa_GetDeviceCount();
    for(i=0; i<ndev; i++){
      info = Pa_GetDeviceInfo((PaDeviceIndex) i);
      if(info->maxOutputChannels > 0) printf("output device: ");
      if (info->maxInputChannels > 0) printf("input device: ");
      printf("%d: %s\n", i, info->name);
    }

    printf("choose device for input: ");
    scanf("%d", &devin);
    printf("choose device for output: ");
    scanf("%d", &devout);

    memset(&inparam, 0, sizeof(PaStreamParameters));
    inparam.device = devin;
    inparam.channelCount = 1;
    inparam.sampleFormat = paFloat32;
    memset(&outparam, 0, sizeof(PaStreamParameters));
    outparam.device = (PaDeviceIndex) devout;
    outparam.channelCount = 1;
    outparam.sampleFormat = paFloat32;

    err = Pa_OpenStream(&handle,&inparam,&outparam,SR,BUF,paNoFlag,
                  audio_callback, data);

    if(err == paNoError){
      err = Pa_StartStream(handle);
      if(err == paNoError){
      while(Pa_GetStreamTime(handle) < END);
      Pa_StopStream(handle);
      } else  printf("%s \n", Pa_GetErrorText(err));
      Pa_CloseStream(handle);
    } else  printf("%s \n", Pa_GetErrorText(err));
    Pa_Terminate();
  }  else  printf("%s \n", Pa_GetErrorText(err));

  free(data);
  return 0;
}
```

D.4 Final Words

Realtime audio programming with a cross-platform API such as PortAudio can be made simple. The examples shown in this text demonstrate the basic ideas behind the use of that library.

However, for more complex systems, other issues may arise, which are beyond the scope of this text. Nevertheless, by offering a plain introduction to realtime audio programming, it is hoped that the user will be able to build on the acquired knowledge, so that more involved systems can be tackled. Some further examples are provided with the PortAudio source code. Information on how to download and install this library, as well as reference documentation, is provided at http://portmedia.sourceforge.net/.

D.5 Notes

[1] Roughly speaking, an API is a set of programming resources (functions, data structures, definitions, etc.) that support a certain development task.

[2] A *handle* is basically a pointer to an open stream. This is actually a pointer to a data structure that is 'opaque', i.e. we do not and should not need to know what it holds. We pass the address of the pointer to this data structure (a pointer to a pointer) because the open-stream function will create a variable for this data structure and make the handle point to it. Then this handle can be used as a reference on all operations on the stream, so we know what stream we are manipulating. These operations are for IO, to change and check the stream status, etc.

[3] A *callback* is a function that is given to a system for that system to invoke it ('call-back') at a later time. Setting up callbacks involve writing the code for the function and then passing it as an argument to a function call that 'registers' the callback with the system in question.

[4] A way to distinguish these two is to consider the concepts of synchronous and asynchronous IO. In the first case, the IO is 'in sync' with the rest of the program, so we wait for input, process it and then output, blocking the execution when we come to each one of the stages of IO (in practice, the blocking might only occur at the input), giving the rise to the name 'blocking IO'. In the latter case, the IO is made independent of the rest of the program, so there is no blocking of execution, and waste of CPU time. So in principle, this can be made more efficient.

In the case of callback asynchronous IO, the system will issue callbacks when data is ready for either input or output (or both), so the rest of the program is not idle waiting for data. To understand this, we can think that there are two parts of the program running in parallel: the main section, with the code we have written in sequential order, and a secondary one that keeps invoking the callback function whenever it is needed.

Although potentially more efficient, this can be more complex to program, as resources (e.g. memory) might need to be shared by the callbacks and the rest of the program. In this case, we might need to protect these resources so that no conflicts arise (such as the different parts of the program trying to assign to a variable at the same time).

[5] It is very common for asynchronous systems to have a means of passing some user-allocated data to the callback. The reason is that in many applications, we will need to have some means of passing data from the main part of the program to the callback and vice-versa. The 'user-data' argument is designed for that. All we need to do is to pass the pointer to any variable we would like to share between the two. Often this variable is a data structure holding several items we need to access in the callback or in the main program. The second example demonstrates this: the callback needs to access some permanent memory for the delay line, so we store it in a data structure, together with the read/write index and pass this as user data.