3 Working With Audio Streams

Gabriel Maldonado

3.1 Introduction

In the computer, a "stream" is a sequential flow of data that moves from one point to another. These two "points" can be two different hardware devices, such as the Random Access Memory (RAM) to the Hard Drive, or, as in the case of audio streams, the flow might move an audio signal that results from some signal processing algorithm to the Digital to Analog Converter (DAC) and then out of the computer to your loudspeakers.

As you are probably well aware, digital computers only deal with numbers (digital derives from the word "digit", a synonym for "numeric"). Thus, the only kind of stream allowed by a computer is a numeric stream - actually it is a flow of binary numbers, zeroes and ones, which are combined to represent different numeric types. To generate such streams, we can *sample* analog signals[1] to *digitize* audio (as well as any sort of varying physical quantity).

A *digital audio signal* is a sequence of numbers that precisely represents the variation of air pressure that, when it stimulates our ear, produces the phenomenon we call *sound*[2]. In this section we will deal with audio streams. In particular, we will learn how to generate an audio signal in the C/C++ language and to store that signal in a file or send this signal directly to a digital-to-analog converter (DAC, the computer soundcard) in order to listen at it. Since the code that I present can be applied to both C and C++, I will refer to the language used as "C/C++". Actually C++ is a complete superset of the C programming language. Most of what is valid and true for C is also valid and true for C++. We will avoid code that is only compatible with C++ in this text. Some understanding of these languages is expected from the reader, but this text will also provide some further support for the study of programming in general.

3.2 Synthesizing Audio Streams to an Output

In this section, 1 will introduce a simple synthesis program *hellosine.c*[3], and compare it with the famous *helloworld.c*[4] code. Our program will generate a sequence of ***samples***[5] from a sine wave, and then stream the signal to the ***standard output***[6].

There are several ways to generate such signals, and in this chapter we will use the following two:

1. Using a standard C library function that calculates the *sine* of a number;
2. Calculating a single cycle of the wave, storing it in an area of the computer's memory called a *table*[7], and using the so called *table-lookup method* or *wave-table synthesis* technique to generate the signal. In this case, any waveform stored in the table could be played, not only a sine wave, but a wave of any shape.

3.2.1 HelloWorld and HelloSine: A Comparison

This section introduces our simple audio-stream example, written in analogy to famous *helloworld.c* program that simply prints a string of text to the console and terminates:

```
/* helloworld.c */
#include <stdio.h>
main()
{
     printf("Hello, world!\n");
}
```

Actually this program (a single *C function* called `main()`) does nothing more than *stream* a simple string to the ***standard output***. In the UNIX operating system, (and in DOS, the Windows console, and Linux) almost all devices, such as the screen, the keyboard, the printer, a file, etc., are considered to be a stream or a file (meaning a set of sequential data). By default, the ***standard output*** is associated with the console (and consequently the output is directed to the screen). But it is possible to ***redirect*** the standard output to another device, for example to a file, by simply calling the executable with the *redirection* character followed by the device name (in the case of a file, the name of the file itself). The redirection character is **'>'**. So, by executing the program with the console line:

```
$ helloworld > pluto.txt
```

the program will not print anything to the screen, but rather, to a file named *pluto.txt*, which is created with the output from the program.

Our program *HelloSine* will do exactly the same thing, but instead of streaming a sequence of characters, it will stream a sequence of samples (as text):

```
/* hellosine.c */
#include <stdio.h>
#include <math.h>

#define SAMPLING_RATE 44100
#define NUM_SECONDS 3
#define NUM_SAMPLES (NUM_SECONDS * SAMPLING_RATE)
#define PI  3.14159265
#define FREQUENCY 440

main()
{
     int j;
     for(j = 0; j < NUM_SAMPLES; j++) {
     float sample;
     sample = sin(2 * PI * FREQUENCY * j / SAMPLING_RATE);
     printf("%f\n", sample);
     }
 }
```

We can produce an audio stream, which in this case will be text containing a sequence of numbers, to a file by running our program (called `hellosine`) as in:

```
$ hellosine > sound.txt
```

3.2.2 A Functional Analysis of *hellosine.c*

In this section, we will discuss what *HelloSine* does and see how it accomplishes its task. My goal is to provide you with some insights that will make your approach to software design more effective. When you are writing a program, it is important to begin with a clear list of ideas on what it will have to do. Therefore, I encourage the reader to begin their own programs by writing a set of assertions regarding the task(s) of each program. At the beginning, these assertions don't need to consider the peculiarities of the particular programming language that will be used to develop the program. Rather, they should be quite generic. In the case of *HelloSine* the first assertion is:

> *Develop a program that writes the samples of a sine wave to the standard output.*

When a programmer attempts to realize this sentence in the C language, they quickly discover that the idea is quite clear, but that the information provided by the sentence is far from complete. "*How many samples should it write, or, how many seconds?"* If you have some notion about digital audio, you would ask yourself: "*Will the sine wave be heard? If so, what will be its frequency and amplitude? What will it have to be its sampling frequency?".*

All these questions must have concrete answers, because the computer only does what the programmer asks, it is unable to take a decision by itself. Such information can be provided in two ways:

1. in a fixed way, i.e. embedded in the program itself;
2. in a parameterized way, i.e. it can be set by the user each time the program is run.

In order to be as simple as possible, our program adopts the first way (fixed).

Now the assertions can be completed:

Develop a program that will:

- *write a stream of samples of a sinusoidal wave to the standard output;*
- *write 3 seconds of audio samples in text format;*
- *be sampled at the same sampling frequency of a commercial audio CD (44100 Hz);*
- *correspond to the pitch of the orchestral tuning note – middle A (440 Hz);*
- *have a monophonic output (1 channel).*

The information is now more complete. But I am only able to use the word '*complete'* because I know that there is an existing C language library that contains a function able to calculate the sine of a number (i.e. the `sin()` function). If I was unaware of this function, then I would have had to figure out how to implement a sine function and add these steps to my previous outline. This set of assertions is often called the "specifications" of a program, or more briefly the "specs".

Previous assertions correspond only to the first pass of the design phase, i.e. ***what*** the program should do. The second pass is: figure out ***how*** to do it. And then, only after answering the second question(s) is it possible to proceed with the third pass: ***implement*** it!

Here are my three phases of software development:

1. outline **what** your program needs to do?
2. figure out **how** to do it?
3. then **do** it?

So then, how do we proceed to the ***do*** phase? Well, old programming books often suggest that a graphic scheme called a "***flow-chart***" should be done before beginning to write the code. Sometimes this kind of approach can be useful for beginners or for small programs, but personally, I think that in most cases it is a waste of time, especially for programs written by a single person, because a complete flow-chart can turn out to be longer and complex than the code itself. Furthermore, flow-charts force people to think in a particular programming paradigm, that is "structured programming", but they don't apply so well to the newer paradigms such as "object-oriented" or "declarative programming." At best, I feel that flow charts are quite useful for work-groups. Here it makes sense that everyone should literally be "on the same page," i.e reading from the same "flow-chart."

Despite my having said this, I don't intend to say that a flow-chart is useless. I simply want to recommend that programmer not get too attached to a particular style, such as traditional flow-charting style. Rather than using a flow-charting approach, I prefer visualizing a program as a series of input-output blocks connected by cables. This is a particularly appropriate model for audio programming, in which we are quite used to connecting audio devices like microphones to mixers to recorders to amplifiers to loudspeakers. Actually, this approach is embodied in the very user interface of a number of visual programming applications that deal primarily with audio - Max/MSP, PD, Reactor, etc.

This kind of visualization is suited for any process dealing with signals (not only audio signals, but also control signals). These diagrams deal with the interconnection of modules or blocks, but hide the internal mechanism of each block from the programmer. Each of these signal modifying, signal processing, or signal generating blocks can be considered a "***black box***". And in a black box paradigm such as this, the programmer only knows the input(s), the output(s) and the behavior of the box, but knows nothing of the algorithm that is employed to make it work. This way of thinking is sometimes called "encapsulation" or "implementation hiding". These are two of the basic principles of ***object-oriented*** paradigm. The most basic means for information hiding in C and C++ is the function. But normally a single function is not powerful or rich enough to speak of it as a form of object-orienting. For now, we are dealing with the "***structured programming***" or "***imperative programming***" paradigm. Figures 3.1 and 3.2 show a comparison between the flow-chart and a block (black box) scheme.

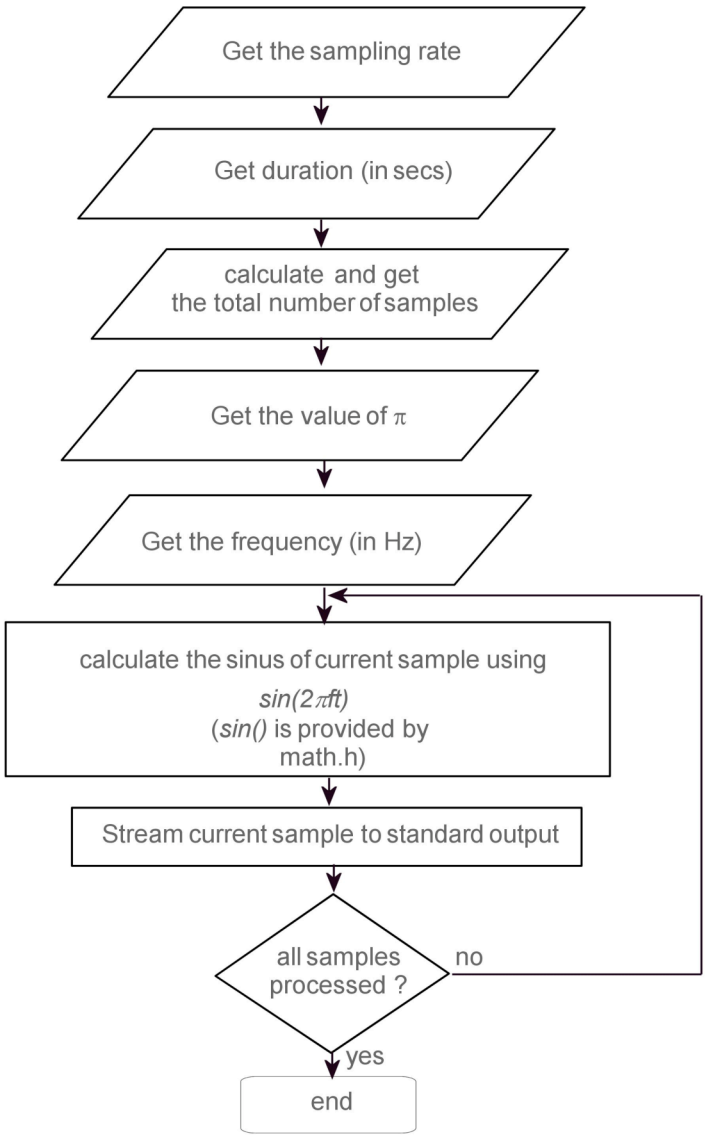

Figure 3.1 A flow-chart of hellosine.c

Clearly, the flow-chart is closer to describing what actually happens in *hellosine.c*, whereas the block-diagram give us a better overview of the structure of the system and all of its connections while hiding what happens internally (in this case, there is only a block made up of the sine-wave generation mechanism).

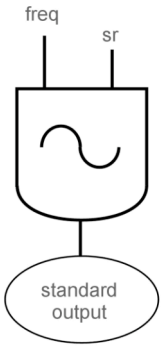

Figure 3.2 A block diagram of hellosine.c

For the graphic representation, one could easily substitute a verbal description of what the program will do to accomplish its task:

1. fill in all the external information that the program needs to run;
2. based on the total duration (in seconds) and the sampling rate (in Hz), calculate how many samples the program will generate;
3. initialize the time to zero;
4. calculate the first sample by using the formula:

$$sin(2\pi ft) \qquad (1)$$

(where ***f*** is the frequency in Hz and ***t*** is the instantaneous time in seconds);

5. output it to the standard output;
6. increment instantaneous time ***t*** by a sample period (i.e. 1/samplingRate) in order to advance to the next sample;
7. repeat steps (4) through (6) until all samples have been computed and sent to the standard output;
8. exit the program.

Now that I have described what and how the program should proceed to do its task, we can move on to the implementation stage. Let's continue with our analysis. As was stated previously, the inputs of our program are embedded, and so, they can't be considered as true inputs from the actual point of view. If the programmer has to change any of these inputs, they would have to recompile the program before running it. They are implemented by means of C macros and are: `SAMPLING_RATE`, `NUM_SECONDS`, and `FREQUENCY` (at the top of the program code). There are two further macros: `NUM_SAMPLES`, (the number of samples to be sent to the standard output), that is obtained by using two previous arguments (i.e. NUM_SECONDS multiplied by the SAMPLING_RATE), and PI (which is nothing more than the famous math constant π – the ratio between the circumference and the diameter of a circle) used in the formula, that was previously defined by the programmer.

In order to calculate and output all the samples, we need a loop iterating/repeating NUM_SAMPLES times. In the body of this loop, the temporary variable 'sample' is filled each

time with a new value, that is obtained by incrementing the index of the loop (the variable `j`) each cycle.

Using the input macro constants, notice that the math formula: $sin(2\pi ft)$ is translated into:

```
sin(2 * PI * FREQUENCY * j / SAMPLING_RATE)
```

In particular, the instantaneous time ***t*** is incremented by a sample period by means of the `j/SAMPLING_RATE` expression, in which the `j` variable expresses both the index of the loop and current sample number.

3.2.3 Table-lookup Oscillators

In this section we will continue to work with audio streams and we will be introduce the ***table-lookup method*** for generating waveforms. This method is also called ***wavetable synthesis*** and it can be used for both artificial sounds, generated entirely by computer calculations, and for acoustic sounds, recorded with a microphone.

Wavetable synthesis is a technique based on reading data that has been stored in blocks of *contiguous computer-memory locations,* called ***tables***. This sound-synthesis technique was one of the very first software synthesis methods introduced in the Music I – Music V languages developed by Max Mathews at Bell Labs in the late 50's and early 60's. Back then, computers were incredibly slow and reading pre-computed samples from memory was much faster than calculating each sample from scratch (as in the case of previous *hellosine.c*, in which the `sin()` function was called for each sample of the synthesized sound).

With table-lookup synthesis, it is sufficient to calculate only a single cycle of a waveform, and then store this small set of samples in the table where it serves as a template. In order to playback the actual sound, this stored single cycle of the waveform must be re-read in a loop at the desired frequency. Among other things, this method provided a huge advantage over the analog oscillators used in radio and electronic music studios of early sixties because analog oscillators were capable of producing but a limited set of wave-shapes (sinusoid, triangle, sawtooth and square), whereas with the table-lookup method, virtually any wave-shape could be synthesized.

Three methods were devised to allow for the specification of arbitrary frequencies:

1. varying the reading-speed (such as in the case of a tape recorder); or
2. reading samples at a fixed rate, but varying the table length for each frequency (using a bigger table for lower frequencies and a smaller for higher frequencies); or
3. reading samples at a fixed rate from a fixed length table, but varying the *increment* of the *indexes* of the samples to be read from the table.

The method actually used for wavetable synthesis (and the most convenient) is the third one. In order to understand it, let's suppose that, given a fixed reading-rate (called the ***sampling rate***), the *'natural'* frequency of a waveform is 100 Hz, when all subsequent samples (or elements) of the table are read, one a time. Each subsequent element of the table is numbered by an ***index***, starting from 0 up to the length of the table (minus 1). The first sample of the table has index 0, the second has index 1, and so on, up the last sample that is table length - 1. So, in a table made up of 1000 samples, the last sample has an index of 999.

The job begins by reading the first element, and sending its value to the output, then the second, then the third, etc. After last element (having index 999) is read, the cycle restarts again from index 0 and so on.

To have a 'natural' frequency of 100 Hz with a table of, suppose, 1000 elements, all 1000 samples must be read in a hundredth of second, so each sample must be read in 1/100/1000 = $1/100000^{th}$ of second. This slice of time is called the *sampling period*. The sampling-rate (actually the reading speed) is the inverse of the sampling period, so in our case, is 100000 Hz. In most digital audio applications, the sampling rate is fixed.

If you wanted the waveform to "oscillate" at twice the frequency (200 Hz), it would be sufficient to read only samples with even indices, skipping the odd ones and reading one value from every other location in the table (i.e. index 0, 2, 4, ... etc.). Again, to have a frequency of 300 Hz it is sufficient to read one sample every three indices, skipping two contiguous indexes each time (i.e. index 0, 3, 6, 9, ..., etc.).

How to obtain frequencies less than 100 Hz? Simply by reading the same sample more than once. For example, to play the waveform at a frequency of 50 Hz, it is sufficient to read the value from each sample twice and then read the next (i.e. index 1, 1, 2, 2, 3, 3, 4, 4, ..., etc.) and to listen to that waveform at 1/3 the 'natural frequency' (33.33... Hz), it is sufficient to read the same sample 3 times before moving along to the next value in the table (i.e. index 1, 1, 1, 2, 2, 2, 3, 3, 3, ..., etc.).

You might be wondering at this point, if this method is capable of obtaining any frequency. The answer is no. Given the technique as explain thus far, the only frequencies that can be obtained from our table-lookup oscillator are those that are integer multiples or integer fractions of the 'natural' frequency of the table. But there is a method that would allow us to obtain any frequency. In order to understand it, we have to introduce the concept of ***phase***, (which for table-lookup oscillators has an extended meaning), and the concept of a ***sampling increment***.

In wavetable synthesis, the phase is a variable that acts as a pointer to the index of a current table element (note: this not to be confused with a *C-language pointer*, we will cover this topic in depth later on). The specific phase value is advanced by the amount of the sampling increment for each sampling period, in order to point to the appropriate table-index at each subsequent reading. In the first of previous examples, a frequency of 100 Hz was obtained by incrementing the phase value by 1 for each sampling period; 200 Hz was obtained with a sampling increment of 2; 300 Hz with a sampling increment of 3 and so on. But, in order to obtain a frequency of 50 Hz, you needed a sampling increment lower than 1, actually 1/2 or 0.5; to obtain a frequency of 33.33... Hz, you needed a sampling increment equal to 1/3 or 0.333 and so on.

Actually, the value of the sampling increment determines, and is proportional to, the frequency of the wave. In most cases, the sampling increment is a ***fractional number,*** not an integer. The phase value is also a real number. So, if the instantaneous value of the phase is, for instance, 10.92843, the actual table element that is extracted has an index of 10 because the integer part of the phase is taken as a pointer to an index of table elements. And obviously, the possible range of values of our phase pointer can only be from 0 up to the table-length minus 1 (a 100 element table goes from 0 – 99). And when the phase value reaches or surpasses the table length, it needs to wrap back around to the beginning in order to restore the phase pointer to a useful range, and this is done by subtracting the length of the table from the calculated sampling increment (if our table is 100 samples long, and our sampling increment is 105, then our phase is

set to a value of 5 and points to a table-index back at the beginning of the table). The method described here, is the 'raw', un-interpolated method and this obviously introduces some error and some "noise" in the signal. To obtain a better sound quality, (a more accurate representation of the data/curve stored in the table) one of several ***interpolation*** methods are usually employed with the most common being the *linear interpolation* method.

3.2.4 A Wavetable Example: HelloTable

In this example, named ***hellotable.c***, we will send a stream of samples to an output, just as we did in *hellosine.c,* but in this case, the output will be to a location other than our standard output because sending samples to the standard output is useless if we want to actually hear the sound. In order to listen to the generated sound, we will have to send the samples to a *device* (either a binary file in one of a number of audio file formats, such as *.wav* or *.aiff*, or a physical device such as the computer's DAC) that allow us to hear it directly, in *real-time* or in *deferred-time* (in the latter case, deferring our listening until "after" the .aiff or .wav file containing the output samples has been written to the hard drive).

As we learned previously, we'll start with a simple verbal description of the requirements of our program:

Write a program that:

- *will output a stream of samples to a destination (device) that can be chosen by the user, at compile-time, from the following: the standard output (like previous hellosine.c); a raw binary file made up of 16-bit integer samples; a Windows standard wav-format file; or the DAC of our computer or on an audio card in real-time*
- *will use the wavetable-synthesis method to generate the samples*
- *will store a single cycle of a waveform in a table*
- *will allow the user to set the following parameters at run-time: the frequency of the waveform in Hertz; the duration of the waveform in seconds (during which the program will reiterate the single wave-cycle stored in the table the corresponding number of times); and the waveshape, chosen from: sinusoid, sawtooth, triangle or square .*

Here is the (partial) source code of *hellotable.c*:

```
#include <stdio.h>
#include <math.h>

#define SAMPLING_RATE 44100
#define PI  3.14159265

#define TABLE_LEN 512
#define SINE      0  /* macros to make array index to mean something */
#define SQUARE    1
#define SAW       2
#define TRIANGLE  3
```

```
float table[TABLE_LEN  ]; /* array (table) to be filled with a
         waveform */

/* ... OTHER CODE HERE (containing functions that fill the table
 with a waveform le, initialization-cleanup, and appropriate output
      according to the chosen out-tination), see later ... */

void main()
{
 int waveform;
 const float frequency, duration;

 printf("Type the frequency of the wave to output in Hz, and press
            ENTER: ");
 scanf("%f", &frequency);

 printf("\nType the duration of tone in seconds, and
          press ENTER: ");
 scanf("%f", &duration);

 wrong_waveform:
 printf("\nType a number from 0 to 3 corresponding to the "
              "waveform you intend to choose\n");
 printf("(0 = sine, 1 = square, 2 = sawtooth , 3 = triangle), "
              "and press ENTER: ");
 scanf("%d", &waveform);
 if ( waveform < 0 || waveform > 3) {
 printf("\nwrong number for waveform, try again!\n");
  goto wrong_waveform;
 }

    /*------ FILL THE TABLE -----------*/
 switch (waveform) {
    case SINE:
    printf("\nYou've chosen a SINE wave\n");
    fill_sine();
    break;
    case SQUARE:
    printf("\nYou've chosen a SQUARE wave\n");
    fill_square();
    break;
    case SAW:
    printf("\nYou've chosen a SAW wave\n");
    fill_saw();
    break;
    case TRIANGLE:
    printf("\nYou've chosen a TRIANGLE wave\n");
    fill_triangle();
    break;
    default: /* impossible! */
    printf("\nWrong wave!! Ending program.\n");
    return;
 }

 init();
 /*------ SYNTHESIS ENGINE START ---------*/
 {
```

```
    int j;
    double sample_increment = frequency * TABLE_LEN / SAMPLING_RATE;
    double phase = 0;
    float sample;

    for (j = 0; j < duration * SAMPLING_RATE; j++) {
    sample = table[(long) phase];
    outSample(sample);
    phase += sample_increment;
    if (phase > TABLE_LEN) phase -= TABLE_LEN;
    }
 }
 /*------ SYNTHESIS ENGINE END -----------*/
 cleanup();
 printf("End of process\n");
}
```

This is only a part of the code of file *hellotable.c*, the other parts (indicated by the comment `OTHER CODE HERE`) will be viewed later. Here are the steps that the program has to follow in order to accomplish its task:

1. provide default settings and "built-in" information (for example, sampling rate, symbolic tags representing the various wave shapes, etc.) in the form of *#define* constants
2. define a ***table*** in the form of a ***global array*** of floats, that will contain the waveform samples
3. provide a set of four user-defined ***functions*** that can fill the table with one of the possible waveshapes (not shown in previous code)
4. provide user-defined functions that initialize and cleanup the output device (not shown in previous code)
5. provide a user-defined function that sends each sample to the output device chosen at compile-time
6. ask the user to provide the run-time information (frequency, duration and wave shape) by typing some numbers to the console
7. store the information typed by the user into numeric variables (floating-point for frequency and duration, integer for the waveshape)
8. if the user makes a mistake when typing the waveshape number, repeat the operation again (this is called "defensive programming")
9. based on the user's choice, fill the table with a waveform cycle by selecting one of previous user-defined functions
10. initialize the output device by calling a user-defined function
11. initialize the processing variables and calculate the sample-increment by using the expression:

$$frequency * table\_length/sampling\_rate \qquad (2)$$

12. start cycling the wavetable-synthesis engine loop by using the following steps:
    a) initialize the sample index to zero;
    b) read a sample from the table according to current phase

c) output the sample to the device chosen at compile-time by calling a user-defined function;
d) increment the phase by a sample step;
e) if the new phase value is greater than table length: subtract the table length from the phase value itself (to wrap around to the beginning);
f) increment the sample index by a unit;
g) if the sample index is less than the value corresponding to the overall duration, continue looping by jumping again to step b);

13. cleanup the output device and exit program

Note: In order to make *hellotable.c* a little more readable at this point, I have omitted the definitions of the functions `fill_sine()`, `fill_square()`, `fill_saw()`, `fill_triangle()`, `init()`, `cleanup()` and `outSample()`, which will be seen later

Let's look at the `main()` function. The integer variable `waveform` serves to contain a symbolic identifier, whose value is chosen by the user among those defined by previous macros – `SINE`, `SQUARE`, `SAW` and `TRIANGLE`. The float variables frequency and duration will be used to hold the other user-supplied parameters.

These variable are filled by the `scanf()` function (from the *stdio.h* library). To fill a variable (or a set of variables), this function is passed the addresses of target variables as function arguments, by means of the `&` operator. Syntactically, the address of a variable is equivalent to a pointer, and such pointer is passed as an argument to the `scanf()` function. After this call, the pointed target of such address is filled with an item coming from the ***standard input*** (by default, the computer keyboard). Thus, the program pauses at the different `scanf()` lines and waits for the user to type a number corresponding to the frequency, duration or waveform, respectively, and carries on when the *enter* key is hit.

Notice that this part of the program contains a ***label***, `wrong_waveform`. A label marks a point in the source code. It is made up of a text identifier followed by the colon ***':'*** character. Labels are used with two C/C++ statements, `goto` and `case` . The `goto` statement is used to jump to the point were corresponding label is located, inside a function. Good programming style suggests that `goto` statements should be avoided, or at least reduced to a minimum, because they make the program difficult to read and messy. However there are situations in which `goto` is very useful, such as in our *hellotable.c* in which some lines are repeated until the user provides a value in the 0 to 3 range, (i.e. it avoids wrong user inputs, that could cause the program to crash.)

The other statement that works with labels is `switch()...case`, that appears next in the program:

```
switch(waveform) {
      case SINE:
            ...something here...
            break;
      case SQUARE:
            ...something here...
            break;
      case SAW:
```

```
            ...something here...
            break;
      case TRIANGLE:
            ...something here...
            break;
      default:
            ...something here...
}
```

The switch()...case statement allows a selection among multiple sections of code, depending on the value of an expression, in this case the content of the `waveform` variable. The expression enclosed in parentheses, is the "controlling expression," and must be an integer variable or constant. The switch statement causes an unconditional jump to a label belonging to a set of labels, called *case labels* enclosed in curly braces. Such labels are different from `goto` labels, because they express an integer value, while `goto` labels are only a text literal. Depending on the value of the controlling expression, the jump goes to the corresponding matching label. The values of the case labels must be integer constants. The statements in the switch body are labeled with `case` labels or with the `default:` label, which will be used if nothing matches.

In our case, the labels are macros defined at the top of *hellotable.c*, serving as identifiers of our waveshapes. So if our variable `waveform` has been filled, for instance, with 1, after the `switch()` controlling expression, program flow will jump to 'case SQUARE: (line 143) and will execute the two following lines. After that, the ***break*** statement (line 146) will exit the switch block, continuing the execution at line 158.

In the `switch()` block of *hellotable.c* there are four cases (plus one default case), each contains a call to a user-defined function whose purpose is to fill the array `table[]` with a single cycle of a wave. The functions are `fill_sine()` (at line 141 of *hellotable.c*), `fill_square()` (line 145), `fill_saw()` (line 149) and `fill_triangle()` (line 153). These functions will be examined later (together with the initialiazation and clean-up functions `init()` and `cleanup()`**)**

The synthesis engine is enclosed in a curly-brace-delimited instruction block. The reason for this enclosure is to be able to declare some local variables placed inside the engine zone itself. The integer variable `j` is a looping index. The variable `sample_increment` (declared as a *double* in order to increase its precision), contains the sampling increment, which depends on the frequency:

```
double sample_increment = frequency * TABLE_LEN / SAMPLING_RATE;
```

The `phase` variable is also a `double`; it is initialized to zero, and will be used as a storage space for the temporary phase value during the performance of the synthesis engine. Finally, the `sample` variable is declared as a `float` and will contain the instantaneous value of the audio wave.

The synthesis engine performance begins at line 168 with a `for()` loop whose body is reiterated for all samples of the output. The actual number of samples is calculated with the expression `duration * SAMPLING_RATE`, where `SAMPLING_RATE` is the number of samples per second. During each cycle of the loop, the `sample` variable is filled with a table element whose index is determined by the integer truncation of the phase (line 169). Since an

array only accepts integer numbers as indexes, this truncation is done by converting the `phase` variable, which is a `double`, into a `long`, by means of the ***cast*** operator.

In line 170, the current value of the `sample` variable is sent to the (previously chosen) output device by calling the user-defined `outSample()` function. (We'll examine this function shortly.) The current phase value is then incremented by the sampling increment, in order to allow the index of the table to point to the appropriate array element at the next cycle. When the value of the `phase` variable exceeds `TABLE_LEN`, a new cycle is starting, and the `phase` variable must be decremented by that amount. This pass is done by the `if()` conditional at the end of the code block.

After the loop block has generated the appropriate number of samples, the program calls the user-defined `cleanup()` function and ends . Here is the listing of remaining code of *hellotable.c* (what was omitted from the previous listing for clarity sake):

```
void fill_sine()/*fills the table with a single cycle of a sine wave */
{
     int j;
     for(j = 0; j < TABLE_LEN; j++)
     table[j] = (float) sin(2 * PI * j/TABLE_LEN);
}

void fill_square()
{
     int j;
     for( j = 0; j < TABLE_LEN/2; j++)
     table[j] = 1;
     for( j = TABLE_LEN/2; j < TABLE_LEN; j++)
     table[j] = -1;
}

void fill_saw() /* descending ramp */
{
     int j;
     for( j = 0; j < TABLE_LEN; j++)
     table[j] = 1 -  (2 * (float) j / (float) TABLE_LEN) ;
}

void fill_triangle()
{
     int j;
     for( j = 0; j < TABLE_LEN/2; j++)
               table[j] =  2 * (float) j/ (float) (TABLE_LEN/2) - 1;
     for( j = TABLE_LEN/2; j < TABLE_LEN; j++)
        table[j] = 1 - (2 * (float) (j-TABLE_LEN/2) / (float) (TABLE_LEN/2));
}

#ifdef REALTIME /* uses Tiny Audio Library  */
     #include "tinyAudioLib.h"
#elif defined(BINARY_RAW_FILE)
     FILE* file;
#elif defined(WAVE_FILE) /* uses portsf library */
     #include "portsf.h"
     PSF_PROPS props;
     int ofd;
```

```
#endif

void outSample(float sample)
{
#ifdef REALTIME /* uses Tiny Audio Library */
    outSampleMono(sample);
#elif defined(BINARY_RAW_FILE)
    short isample = (short) (sample * 32000);
    fwrite(&isample,sizeof(short),1,file);
#elif defined(WAVE_FILE) /* uses portsf library */
    psf_sndWriteFloatFrames(ofd,&sample,1);
#else /* standard output */
    printf("%f\n", sample);
#endif
}


void init()
{
#ifdef REALTIME /* uses Tiny Audio Library */
    tinyInit();
#elif defined(BINARY_RAW_FILE)
    file = fopen("hellotable.raw","wb");
#elif defined(WAVE_FILE) /* uses portsf library */
    props.srate = 44100;
    props.chans = 1;
    props.samptype = PSF_SAMP_16;
    props.format = PSF_STDWAVE;
    props.chformat = STDWAVE;
    psf_init();
    ofd = psf_sndCreate("hellotable.wav", &props, 0, 0, PSF_CREATE_RDWR);
#else /* standard output */
    printf("\n...Nothing to initialize...\n");
#endif
}

void cleanup()
{
    printf("cleaning up... ");
#ifdef REALTIME /* uses Tiny Audio Library */
    tinyExit();
#elif defined(BINARY_RAW_FILE)
    fclose(file);
#elif defined(WAVE_FILE) /* uses portsf library */
    {
        int err1, err2;
        err1 = psf_sndClose(ofd);
        err2 = psf_finish();
        if(err1 || err2)
        printf("\nWarining! An error occurred writing WAVE_FILE file.\n");
    }
#else /* standard output */
    printf("nothing to clean up... ");
#endif
}
```

User-functions `fill_square()`, `fill_saw` and `fill_triangle()` simply fill the `table[]` array with the corresponding single-cycle (non-bandlimited) waveshapes square, sawtooth and triangle. Notice that, in these cases, the `for()` loop is used without curly braces because the statement is the assignment of the calculated sample to each element of the table. In C, if the `for()` loop contains a single C statement curly braces are optional. This time, the ***cast*** operator is used to convert an `int` variable to a `float`.

In programs made up of a single source file, the `main()` function is typically placed after all the other user-defined functions, as in *hellotable.c*. The reason for this practice is that the compiler needs to know the types of the function arguments, and the types of function return values before the lines in which such functions are called from the `main()`. Alternatively, it is typical to precede the function calls with ***function prototypes***. These typically appear in the `#include` (or header) files of libraries. In this case, the bodies of such functions are usually pre-compiled into the library binary file.

Next in the code we see these lines:

```
#ifdef REALTIME /* uses Tiny Audio Library */
#include "tinyAudioLib.h"
#elif defined(BINARY_RAW_FILE)
FILE* file;
#elif defined(WAVE_FILE) /* uses portsf library */
#include "portsf.h"
PSF_PROPS props;
int ofd;
#endif
```

Here we see some C pre-processor statements, followed by lines of normal code (global variable declarations). The `#ifdef`, `#elif defined()`, `#else` and `#endif` statements relate to *conditional compilation*. They are used to check to see if a certain macro has been defined. Depending on this, the pre-processor tells the compiler to compile or not the different chunks of code between `#ifdef` or `#elif` up to next `#elif` or `#else` statement.

In our case, the C pre-processor checks if the `REALTIME` macro has been defined somewhere (either in a header or as an option to the compiler). If it has been defined, the pre-processor tells the compiler to compile the line

```
#include "tinyAudioLib.h"
```

ignoring the rest. Otherwise it checks if the `BINARY_RAW_FILE` macro has been defined, and skips the other conditions, etc. Thanks to conditional compilation, the *hellotable* program allows the programmer to choose, from a set of different devices, an output device at *compilation time* (not at *runtime*). Such devices are:

1. Realtime: sends the output directly to the DAC in order to listen to the sound while the processing stage is happening. This choice uses the *Tiny Audio Library*, (a simple library that is a wraps the ***Portaudio pablio*** library in a way that makes it much easier to use by beginners).
2. Binary Raw File: the output is written to a raw audio file (16 bit mono), that can be played later with any wave players or editors that supports the specification of

information about the format of the data in the raw audio file, such as the sampling rate, the number of channels (stereo or mono), and the sample resolution (8, 16 or 24 bit, or even 32 bit floats). The advantage of using a Raw Audio File is that in this case hellosine.c doesn't require us to link in any special libraries. We can write the file by simply using the *stdio* library.

3. Wave File: the output is written to a standard Windows .wav file (based on Richard Dobson's *portsf* library). This format is automatically recognized by most wave players. Thus, it might prove easier to use than raw files, because the user doesn't have to remember the special format of the audio file.
4. Standard Output: the default choice. Sends each output sample to the *standard output* and provides numeric text strings expressing the fractional values of each sample (like *hellosine.c*). This choice is valid if the `REALTIME`**,** `BINARY_RAW_FILE` or `WAVE_FILE` macros have **NOT** been defined.

It is worth noting that the programmer need only provide the libraries corresponding to the compilation choice, and can ignore all other libraries. Conditional compilation can help to successfully build a program that might be missing some libraries are those that might have included libraries that were platforms or operating system specific. In fact, in the second and fourth cases of conditional compilation of *hellotable.c* (i.e. Binary Raw File and Standard Output), the programmer doesn't have to provide any special library because ***stdio*** is available on all platforms.

Next in the source code, there is the definition of the `outSample()` function. This function accepts a `float` as an input and sends the number, from the `sample` argument variable, to an output device chosen by the user at compilation time.

Again, there are some conditional compilation statements:

1. Given the `REALTIME` choice, the input argument is simply fed as an argument to the `outSampleMono()` function (line 63) that belongs to the *Tiny Audio Library*. This function sends the *float* to the DAC in realtime, in order to listen at it.
2. Given the `BINARY_RAW_FILE` choice, the *short* variable `isample` (a 16 bit integer) is filled with the `sample` input argument value, after rescaling it between the ranges of -32000 to +32000 (the original range was -1 to 1). Also, in order to truncate eventual fractional values, a type conversion is done by the `cast` operator. In the following line, there is a call to the `fwrite()` function from the *stdio* standard library that writes a chunk of contiguous binary data to a previously opened file. The three input arguments required by this function are: the initial address of the data chunk operator, (in our case this is simply the address of the `isample` variable), the size of one of the data elements to be written, (in our case this is a *short* and its size is obtained by means of the `sizeof( )` operator), the total number of data elements to be written, and finally, a pointer to a previously-opened file (in our case the file is opened in the ***init()*** user-defined function, but its pointer was previously declared as a global variable).
3. Given the `WAVE_FILE` choice, the input argument is simply fed as an argument to the `psf_sndWriteFloatFrames()` function that belongs to the *portsf* library.

4. Given the default case, a simple `printf()` function call feeds the `sample` argument to the *standard output.*

The next definition seen in the code is of `init()`. At compilation time, it serves to initialize the chosen output device in the appropriate way. Here again, there are some conditional compilation statements:

1. In the case of the REALTIME (i.e. *Tiny Audio Library*) choice, a ***tinyInit()*** function call is provided (actually this is not necessary because the *Tiny Audio Library* automatically intializes itself on the first call to an input or output function).
2. In the case of the `BINARY_RAW_FILE` choice, a file is opened by calling the `fopen()` function (belonging to *stdio* standard library) that gets, as its input arguments, a string containing the name and path of the file to be opened, and a string containing some specifiers that set the mode and operations to be applied to the corresponding file (in our case, the string "`wb`" stands for 'write' and 'binary'). The `fopen()` function returns a FILE pointer that fills the `file` global variable.
3. In the case of the `WAVE_FILE` choice, the fields of the global ***props*** structure belonging to the user-defined type `PSF_PROPS` (defined inside the ***"portsf.h"*** header file belonging to the *portsf* library), are filled, one-a-time. Then a call to the `psf_init()` function intializes some of the hidden stuff in the *portsf* library. At last, a call to the `psf_sndCreate()` function fills the `ofd` file descriptor.
4. In the default case, nothing has to be done in this pass.

Finally, we have the definition of the `cleanup()`. In an appropriate way, it serves to close the output device and terminate all pending actions. In this case, conditional compilation statements are:

1. `REALTIME`: call the `tinyExit()` function of *Tiny Audio Library,* that closes opened devices (Note: not really necessary because the *Tiny Audio Library* automatically calls this function at exit if it is not called explicitly).
2. `BINARY_RAW_FILE`: call the `fclose()` function of the *stdio* standard library, that closes the corresponding file. It accepts a pointer to a previously-opened file as an argument.
3. `WAVE_FILE`: calls two functions: the `psf_sndClose()` that takes the `osf` file descriptor as an argument and the `psf_finish()` function. Return values are provided that express some information that indicates the presence or absence of errors. Such return values, used to show specific warnings, are stored in the `err1` and `err2` variables.
4. Default (standard output): no operation is needed to clean up.

3.3 Playing an Audio File

In this section we will write a set of *waveplayer* programs using the *Tiny Audio Library* and the *portsf* library, for playing *.wav* and *.aif* files. This application will perform two basic actions:

- read a block of samples from an audio file stored in the hard disk;
- send samples to the DAC

with a *streaming* read and play operation. This means that it is not necessary to load the entire file in memory to play it. Rather, we read a small amount of it while playing a previously-read chunk. This kind of operation involves the concept of 'buffering'. However, in these programs, most of buffering mechanism is hidden from the programmer by library functions, so that user doesn't need to worry about them. In more complex applications, the programmer will have to deal with the buffering mechanism directly.

3.3.1 A Simple Command Line Audio File Player

The first example, *player.c* is a program that can play a mono or stereo audio file from the command line. The command line syntax is the following:

```
$ player  filename
```

where *player* is the executable name and filename is the name of a *.wav* or *.aif* file.

Here is a verbal description of the steps that the program will cover to accomplish its task:

- *include* the needed libraries[8];
- *make declarations:* declare and define an array of floats that will act as the sample streaming buffer (see later); declare the file handle (actually a descriptor in **portsf** library) and a structure containing some information about the opened audio file; declare a long variable that will contain the number of read sample frames;
- *check* the command line syntax;
- *initialize* the libraries and open the audio file*; check* if it has been opened correctly; *exit* with an error if the number of channels of the audio file is greater than 2 (only mono and stereo files are allowed);
- *perform* the loop in which a block of samples is read into a buffer from the file, then such block is played by sending the buffer to the real-time output;
- *repeat* the loop body until all samples of the file are played;
- *close* the file*, finish* library operations and *exit* the program.

And here is the source code of *player.c*:

```
#include <stdio.h>
#include "tinyAudioLib.h"
#include "portsf.h"

#define FRAME_BLOCK_LEN 512
void main(int argc, char **argv)
{
 float buf[FRAME_BLOCK_LEN * 2]; /* buffer space for stereo (and
                  mono) */
 int sfd; /* audio file descriptor */
 PSF_PROPS props; /* struct filled by psf_sndOpen(), containing
            audio file info */
```

```
 long nread; /* number of frames actually read */

 if ( argc != 2 ) {  /* needs a command line argument */
      printf("Error: Bad command line. Syntax is:\n\n");
      printf("     player  filename\n");
      return;
 }

 psf_init(); /* initialize portsf library*/
 sfd = psf_sndOpen(argv[1], &props, 0); /* open an audio file
                        using portsf lib */
 if (sfd < 0) { /* error condition */
      printf("An error occurred opening audio file\n");
      goto end;
 }
 if (props.chans > 2) {
      printf("Invalid number of channels in audio file\n");
      goto end;
 }

      /*========= ENGINE =========*/
 do  {
      nread = psf_sndReadFloatFrames(sfd, buf, FRAME_BLOCK_LEN);
      if (props.chans == 2)  /* stereo */
            outBlockInterleaved(buf, FRAME_BLOCK_LEN);
      else  /* mono */
            outBlockMono(buf, FRAME_BLOCK_LEN);
 } while (nread == FRAME_BLOCK_LEN);
      /*======== ENGINE END ======*/
 end:
 printf("finished!\n");
 psf_sndClose(sfd);
 psf_finish();
}
```

We start by including the necessary header files[9] for the libraries used in this code. The macro `FRAME_BLOCK_LEN` defines the length of the buffer block, used for the temporary storage of samples, expressed in ***frames***. A *frame* is similar in concept to the *sample*, and is actually the same thing in the case of ***mono*** audio files. For multi-channel files, a *frame* is the set of samples for all channels, belonging to a determinate sample period. For example, in the case of a stereo file, each frame contains 2 samples, the first sample representing the left channel, the second one representing the right channel. In the case of a quad file a frame will contain 4 samples and so on.

Next, we have the main() function head. This time the main() function takes two arguments:

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

These two arguments serve the `main()` function to handle the command-line arguments typed by the user at the operating system shell in order to run the program. The `argv` argument is a pointer to `char` (a `char` is an 8-bit integer, often used to express a character) that actually

points to an array of strings (remember that a `string` in the C language is nothing more than an array of `char`). So, the `argv` pointer is used to handle the command-line arguments provided by the user when starting a program from the text-based console shell. Command-line argument access can be accomplished in the following way:

- `argv[0]` returns the address of a string containing the name of program executable;
- `argv[1]` returns the address of a string containing the first command-line argument;
- `argv[2]` returns the address of a string containing the eventual second argument
- ... and so on

The `argc` argument is an `int` (integer) containing the number of arguments that the user typed at the console to run the program (this number must also include the program executable name which is also considered to be an argument). From the console, each argument must be separated from others by means of blank spaces or tabs. In our case, the user has to type two arguments (the program name and the name of the audio file to be played), so `argc` has to contain *'2'*. If it doesn't, an error message will be displayed and the program is stopped.

Next, an array of 1024 *float* elements (`FRAME_BLOCK_LEN * 2`) is declared and defined:

```
float buf[FRAME_BLOCK_LEN * 2];
```

This array is actually the buffer that will contain the block of 512 frames read from the audio file. It is dimensioned to 1024, in order to fit 512 stereo frames. In the case of mono frames, the exceeding space will not be used, but this will not affect the correct functioning of the program.

Following, this we declare the file identifier `sfd` and the `props` variable, which is a structure `PSF_PROPS`, defined in *portsf.h*. This structure contains information about an audio file opened with *portsf* lib, and has the following fields:

```
typedef struct psf_props  {
long        srate;
long        chans;
psf_stype   samptype;
psf_format  format;
psf_channelformat chformat;
} PSF_PROPS;
```

where the `srate` member variable contains the sample rate of the audio file, the `chans` variable contains the number of channels, the `samptype` variable (an *enum* type declared in *portsf.h*) contains the format of samples (for example, 16-bit integers, 8-bit integers, 24-bit integers, 32 bit floating-point values, and so on), the `format` variable contains the format of the audio file (for example .wav or .aiff), and the `chformat` variable contains information about channel displacement (not particularly useful in our current case). The last variable to be declared is `nread`, a `long` that will contain the number of frames read from the file by the loop engine (see later).

The program goes on to initialize *portsf* lib and open an audio file:

```
psf_init();
sfd = psf_sndOpen(argv[1], &props, 0);
```

This function attempts to open the audio file whose name is contained in the string pointed to by `argv[1]`, and returns a file descriptor in the `sfd` variable, if that file has been correctly opened. In the case of the `psf_sndOpen()` function, a valid file descriptor is a positive integer value. It will be used by further functions that access the corresponding file as a unique identifier. If the file doesn't exist, or any other error occurs, the `sfd` variable will be filled with a negative value.

Normally, C language functions are only able to return a single argument, but there is a trick to make it possible to actually return any number of arguments. We use this trick with the second input argument of the `psf_sndOpen()` function. It behaves as an output argument, or better, as eight output arguments, given that eight is the actual number of member variables belonging to the `props` structure. Having an input argument behave as an output argument is done by passing the *address* of the corresponding variable (that will be filled with the return value), instead of passing the variable itself to the called function. So, the address of the `props` is passed as an input argument (even if it will contain a return value), and this structure is filled by the `psf_sndOpen()` function internally (with the return value itself, i.e. the filled `props` structure). The third argument is a *rescaling* flag; that is not set in our case (because we don't want to rescale the samples); so it is set to zero.

We then test if the audio file has been correctly opened, by evaluating the sign of the `sfd` variable. If it is negative, an error message is printed to the console; and in this case, the program is ended by the `goto` statement, that forces program flow to the `end:` label at the bottom of the `main()` function. We will also test the member `chans` of `props`. If the opened file has a number of channels greater than 2, the program prints an error message ends, because such types of files are not supported by the program.

The next section is the synthesis engine loop. Because the conditional has to be placed after at least one iteration has been done, a `do...while()` statement block is used. Look at line 32:

```
nread = psf_sndReadFloatFrames(sfd, buf, FRAME_BLOCK_LEN);
```

This function reads a block of frames and places them in the `buf` array. The input arguments are, in order, the file descriptor (`sfd`), the buffer address (`buf`), and the number of frames to be read into the buffer itself (`FRAME_BLOCK_LEN`). It returns the number of samples actually read, equal to `FRAME_BLOCK_LEN`, unless the end of the audio file has been reached.

Inside the loop, there is a conditional that switches two different lines, depending on whether the audio file is stereo or mono. Actually, the Tiny Audio Lib offers support to various buffer formats, i.e. mono or stereo, separate channels or interleaved frames, and so on. The `outBlockInterleaved()` function accepts a buffer made up of multi-channel interleaved frames, whereas `outBlockMono()` accepts only mono buffers.

Finally, we have the loop conditional. When `nread` contains a value different from `FRAME_BLOCK_LEN`, it means that the end of file has been reached, and so the loop stops. The program then closes the audio file, and terminates the *portsf* library operations.

3.3.2 A Command Line Player with some Options

A new, slightly different player is presented in this section. This player will allow us to specify both the starting point and the duration of the file to play. It will also be an opportunity to see how to handle the positions of a block of audio data. The command-line shell syntax of this player will be:

```
$ player2  [-tTIME] [-dDUR] filename
```

where `player2` is the name of the executable; `-t` is an optional flag that must be followed by an integer or fractional number (without inserting blank spaces), denoting the starting *TIME* point expressed in seconds (note: square brackets indicate that the corresponding item is optional and does not have to be typed in command line); `-d` is another optional flag followed by a number denoting the *DUR*ation in seconds of the chunk of sound actually played; `filename` is the name of the .wav or .aiff audio file to play.

Here is our usual verbal description of the steps the program has to do to accomplish its task:

1) include the necessary library headers;
2) define a tiny function that displays the command-line syntax of the program in several error-condition points;
3) define the length of the array containing the streaming buffer;
4) in the `main()` function, declare (and eventually define) most local variables used in the program;
5) check if the optional command line flags are present, and evaluate corresponding arguments;
6) check if the syntax is correct, otherwise display an error message and exit the program;
7) initialize everything and open the audio file;
8) display some audio file information;
9) calculate play-start and play-stop points in frames;
10) seek the calculated play-start point in the opened file;
11) start the playing engine loop;
12) read a block of frames and output it to the DAC;
13) repeat step 12 until the play-stop point is reached;
14) close the audio file, cleanup library and end the program.

Here is the listing of the *player2.c* file:

```
#include <stdio.h>
#include <math.h>
#include "tinyAudioLib.h"
#include "portsf.h"

void SYNTAX(){
      printf ("Syntax is:\n          player2 [-tTIME] [-dDUR] filename\n");
}
```

```
#define FRAME_BLOCK_LEN 512
void main(int argc, char **argv)
{
float buf[FRAME_BLOCK_LEN * 2]; /* buffer space for stereo
  (and mono) */
int sfd; /* audio file descriptor */
int   opened = 0; /* flag telling if audio file has been
 opened */
PSF_PROPS props; /* struct filled by psf_sndOpen(),
   containing audio file info */
long counter; /* counter of frames read */
long length;      /* length of file in frames */
long endpoint; /* end point of playing */
extern int arg_index; /* from crack.c */
extern char *arg_option; /* from crack.c */
extern int crack(int argc, char **argv, char *flags, int ign);int flag,
timflag=0, durflag=0; /* flags */
long nread; /* number of frames actually read */
double startime, dur;

while (flag = crack(argc, argv, "t|d|T|D|", 0)) {
      switch (flag) {
            case 't':
            case 'T':
            if (*arg_option) {
                  timflag=1;
                  startime = atof(arg_option);
            }
            else {
                  printf ("Error: -t flag set without"
                          "specifying a start time in seconds.\n");
                  SYNTAX();
              return;
            }
            break;
            case 'd':
            case 'D':
            if (*arg_option) {
                  durflag=1;
                  dur = atof(arg_option);
            }
            else {
                  printf ("Error: -d flag set without"
                      "specifying a duration in seconds\n");
                  SYNTAX();
                  return;
            }
            break;
            case EOF:
            return;
            }
}

if ( argc < 2 ) {  /* needs a command line argument */
      printf("Error: Bad command line arguments\n");
      SYNTAX();
```

```
      return ;
}

psf_init(); /* initialize portsf library */
sfd = psf_sndOpen(argv[arg_index], &props, 0); /* open an audio
           file using portsf lib */
if (sfd < 0) { /* error condition */
      printf("An error occurred opening audio file\n");
      goto end;
}

printf("file \'%s\' opened...\n", argv[arg_index]);
printf("sampling rate: %d\n", props.srate);
printf("number of chans: %d\n", props.chans);
length = psf_sndSize(sfd);
printf("duration: %f\n", (float) length / (float) props.srate);

if (timflag)
      counter = (long) (startime * props.srate); /* length in
                        frames */
else
      counter = 0; /* beginning of file */

if (durflag) {
      endpoint = (long) (dur  * props.srate + counter);
      endpoint = (endpoint < length) ? endpoint : length;
}
else {
      endpoint = length;
      dur = (double) (endpoint-counter) / (double) props.srate;
}

if (props.chans > 2) {
      printf("Invalid number of channels in audio file, max 2
              chans allowed\n");
      goto end;
}

psf_sndSeek(sfd, counter, PSF_SEEK_SET); /* begin position at the
             appropriate point */

printf("Playing the file from time position %0.3lf for %0.3lf
              seconds...\n", startime, dur);

/*========= ENGINE =========*/
do {
      nread = psf_sndReadFloatFrames(sfd, buf, FRAME_BLOCK_LEN);
      if (props.chans == 2)  /* stereo */
            outBlockInterleaved(buf, FRAME_BLOCK_LEN);
      else  /* mono */
            outBlockMono(buf, FRAME_BLOCK_LEN);
      counter+=FRAME_BLOCK_LEN;
} while (counter < endpoint);
      /*======== ENGINE END ======*/
end:
printf("finished!\n");
psf_sndClose(sfd);
```

```
psf_finish();
}
```

Since most of code of this new file has been taken from previous examples, all lines will not be explained; only new concepts will be clarified. In the `main()` function, a flag `opened` is declared. In a program source, a flag is a variable (normally an `int` variable) that can contain a limited number of ***states*** that affect the program's behavior during execution. (Don't confuse *flag variables*, in the source code, with the *option flags,* provided at the command line. Even if they could sometimes be related, they are two very different concepts). The states of a flag are expressed by means of the value of the variable itself. Each possible state is determined by a unique integer number which is assigned to the flag variable. And these numbers are actually symbols of some strictly non-numeric concept, and so, they will not be used as true numbers (for example, arithmetic operations on flags are senseless).

Very often, as in our case here, a flag is ***boolean.*** This means that its possible states are either *true* or *false.* Boolean flag values are *non-zero* to indicate the *true* state and *zero* to indicate the *false* state. It is initially set to zero (false), to indicate that the file is not opened.

We also declare a *long* variable named ***counter***, to contain the count of read frames. It is declared as *long*, not as *int*, because in some platforms `int` might express a 16-bit integer that would not be sufficient for most file lengths. The Variable `length` (a `long`) will contain the file length and `endpoint` will hold the end point of playback, both expressed in frames.

There are three `extern` declarations: the variable `arg_index`, an index to the command-line arguments; the string pointer `arg_option`, that will point at the currently-parsed command-line argument, and the function prototype of `crack()`, a function defined in another source file (actually a library function, even if in this case the library contains only this function, located in file *crack.c*, so is compiled together with *player2.c*). Being a library function, `crack()` will be used as a black-box, i.e. we have only to know what it does, not how it functions internally.

Finally, we have the declarations of three flags: the `flag` variable, that is used as temporary storage of starting option-flags provided by the user when starting the program at the command-line shell; the `timflag` variable, that signals if the `-t` option has been provided; and the `durflag` variable, that signals if the `-d` option has been provided. These last ones are linked to to `startime` and `dur`, which will contain the start time and duration optionally provided by the user from the command line.

The program execution contains a loop that checks the command-line options by evaluating the `crack()` function. This function (authored by Piet van Oostrum and provided with his `mf2t` open-source program), receives the command line by means of its arguments:

```
flag = crack(argc, argv, "t|d|T|D|", 0)
```

Actually, this function has 1 output argument and 4 input arguments. The first two input arguments, `argc` and `argv` (coming from the `main()` function), pass the command line to the `crack()` function, in order to evaluate the command line itself and see if some option flags are present. According to a Unix-style convention, all option flags must be preceded by a hyphen '**-**'. In *player2.c*, valid flags are `-t` and `-d`; in fact, the third input argument of `crack()` is a string containing the letters allowed for option flags, that in this case must be followed by a

numeric datum indicating the starting point (for `-t` flag) and duration (for `-d` flag) in seconds. In the string provided in the third argument of `crack()`, allowed flags are `-t`, `-d`, `-T` and `-D` (uppercases meaning the same thing of lowercases). The **'|'** character indicates that other data must follow the corresponding flag letter, without inserting spaces between the flag letter and the datum in the command line.

The last input argument of `crack()` is a boolean flag indicating if `crack()` should raise an error message in case of unknown flags (when set to 0), or simply ignore them (when set to 1). In this case, the action of raising the error message when an unknown flag is found, is enabled.

Each time `crack()` is called, it puts a found flag character in the `flag` return variable. It also fills the global variables `arg_index` and `arg_option`. The `arg_index` variable is incremented each time a new flag is found, in subsequent calls to the `crack()` function. If some data follows a found flag, such data is contained in a string pointed to by the `arg_option` pointer. The `arg_option` pointer is updated with corresponding string at each subsequent call. More information about using `crack()` is available in the file *crack.c*[10].

The *crack()* function is called repeatedly until the last command-line flag has been read. The *switch()* block at line 28 offers two choices: if the return value of *crack()* contains the flag ***-t*** (or uppercase ***-T***), then `timflag` is set to *1* (i.e. *true*, that means that user has provided a custom play-start time) and the `startime` variable is filled with the corresponding value in seconds (line 33). Notice that the standard library function `atof()` has been used to convert the play-start datum from the original representation (i.e. a character string set by `arg_option` variable) into a `double` floating-point value. If the ***-t*** flag was specified without providing a corresponding datum, an error message will rise. In this case, the command-line syntax will be displayed by means of a call to the user function `SYNTAX()`, defined at the start of the program and the program will be stopped. If the return value of `crack()` contains the flag ***-d*** (or ***-D***), then the `durflag` is set to 1 (*true*) and the `dur` variable is filled with the corresponding playing duration in seconds. Again, if the user forgot to provide the datum, the corresponding messages will be displayed and the program will end. If the return value of *crack()* contains a special value defined by the `EOF` macro constant, the program will terminate (after `crack()` itself displayed an error message). This condition will happen if the user provides some unknown flag, different from ***-t*** and ***-d***.

If the user has not provided any audio file name, an error message is displayed, (the command line syntax) and the program terminates. Otherwise the *portsf* library is initialized and we open the file, as in *player.c*. Once this is done, the program prints some information about the opened file to the console, such as its name and sampling rate. The `length` variable is filled with the length of the file in frames, by the *portsf* library function `psf_sndSize()` and this is converted to seconds and printed to the console.

Next, the program sets the counter to the appropriate initial value. (Notice how the state of `timflag` determines the conditional flow). The endpoint is also assigned the appropriate value, with the state of `durflag` determining the conditional flow. The play-stop point (i.e. the `endpoint` variable) is calculated by adding starting time (the value of counter variable) to the duration, whose unit is converted from seconds to frames by multiplying it by sampling rate. We then need to check whether this variable exceeds the length of the file, using a conditional assignment:

```
endpoint = (endpoint < length) ? endpoint : length;
```

The right member of the assignment is a conditional that evaluates the expression inside the round braces and, if the result is true, then we assign the value immediately following the *'?'* symbol, else, we assign the value following the *':'* symbol. Such expressions are used quite often in the C language.

In case the user did not provide the `-d` option flag, endpoint is assigned the total length of the file and the `dur` variable is set to the corresponding duration in seconds. Notice that the two members of this division are previously converted to double in order to produce a reliable result.

As in the previous example, if the opened file has a number of channels greater than two, the program is ended because it it only supports mono and stereo files. The *portsf* library function call:

```
psf_sndSeek(sfd, counter, PSF_SEEK_SET);
```

This call moves the current file pointer to the location corresponding to the second input argument (the `counter` variable). The `PSF_SEEK_SET` argument is a macro that indicates that the file pointer has to be moved by counting from the beginning of the file. (Other option macros are: `PSF_SEEK_CUR`, that moves the pointer starting from current position; and `PSF_SEEK_END`, that counts backwards from the end of file.) The `psf_seek()` function is similar to the `fseek()` function from the `stdio` library.

Complementing the program, we have the performance engine loop of *player2.c*. It is very similar to that of *player.c*, but in this case the exit-loop conditional compares the current counter value with the play-end point of the file (from the `endpoint` variable). At each loop cycle, `counter` is incremented by the block length.

3.4 Critical Real-time Issues

In this section a number of special techniques will be explained that are used for streaming audio or MIDI data in realtime either from an input or to an output. So far, our audio data has been simply output by means of a single C library function (***Tiny Audio Lib***). We have treated this library as a black-box without being aware of what is really happening inside.

Actually the audio engine of some of the previous applications (*player.c, player2.c, playerGUIsimple.cpp* and *playerGUI.cpp*) was realized by means of a simple loop, something like this:

```
do {
      buf = get_new_data();
      outBlock(buf, FRAME_BLOCK_LEN);
} while (buf != NULL);
```

In this hypothetical function[11] that gets sample data from an eventual input and returns a pointer to each data block, `buf` is a pointer to a block of samples returned by `get_new_data()`. The `buf` pointer is then used as an argument of the `outBlock()` function.

The concept of buffering is inherent in this *do-while* loop. A buffering technique is encapsulated inside the function `outBlock()`, so that the user doesn't have to worry about it. The user's only job is to fill the buffer with the proper data items, (in musical cases these items are sample frames, MIDI messages, or control signal frames), and to provide the address of the buffer. The `outBlock()` function call doesn't return until the block of sample frames has been sent to the target device. If this device is an audio interface, the call to `outBlock()` freezes the program until the block of samples is actually played by the DAC. Each time a streaming operation has successfully finished, `outBlock()` returns, the program is unfrozen and continues to fill another block with the new items. Finally, the stream terminates when a `NULL` pointer is returned by the `get_new_data()` (obviously, the `outBlock()` function should be able to interpret a `NULL` pointer correctly by doing an appropriate action, such as, doing nothing or cleaning-up corresponding device).

3.4.1 FIFO Buffers

If the previous example were used for realtime audio streaming, and if the block of samples were completely played, there would be a time lag that the program itself would take, in order to calculate the new block of samples. If this time lag surpasses the duration of a sample period, the audio interface would remain, for some moments, without samples to send to the output. This situation would cause a *drop-out*[12] in the sound, that would be perceived by the listener as an annoying and disturbing click.

To avoid drop-outs such as this, the solution is to use more than a single block of samples. In this case, the block passed to the function *outBlock()* would be copied to another internal block (often hidden to the user), before the samples are actually played by the DAC, and the program would be free to calculate another block while the audio interface was converting the samples stored in the previously copied block. This technique is often called '***double buffering***'. If necessary, even more than one block can be filled before the audio converter stage. So we can have 'triple buffering', 'quadruple buffering' and so on. In these cases, the items of such buffering techniques would be the block of samples[13].

Buffering is an important technique for realtime streaming, and the buffering scheme used for streaming is referred to as a ***FIFO*** (**F**irst-**I**n **F**irst-**O**ut) *queue* or a *circular buffer*[14]. Realtime streaming buffers can be synchronous or asynchronous. When the rate of the items is constant (for example, in the case of audio or video streams), we deal with a synchronous stream, whereas, if the rate is not constant (for example in MIDI) we deal with an asynchronous stream.

There are two types of *FIFO* buffers, ***input buffers*** and ***output buffers***. *Input buffers* receive data from an input device and feed the program with such data. The data is then processed by the program, and eventually send to an output *FIFO* buffer, which sends such data to an output device. An input *FIFO* buffer is made up of a memory block and a routine that inserts the items coming from the input source into the ***head*** of the buffer (this action is normally called '***push***'), continuing to do such operation until there is no more buffer space available.

An input-buffer-full situation can happen if the average processing speed of the program is not fast enough[15] to extract and process the incoming data. In fact, each item should remain in the buffer only for a limited amount of time, then it must be extracted and removed from the ***tail*** of the buffer by the program itself, in order to process it (this action is normally called '***pop***'). If the input buffer is completely empty, the program temporarily stops processing, waiting for more data to be available.

In the case of *output buffers*, the data items are generated by the program and *pushed* to the ***head*** of the buffer. The output device receives the data items from the ***tail*** of this buffer and sends them to the output when they are needed. The processed data is *popped* from the buffer by the output device. In case the buffer is completely full (this is a normal case), the program stands-by and waits until the buffer is able to receive more data. Be aware that, in the case of output buffers, if the program provides the processed data items more slowly than they are sent to the output the buffer can become empty. And in the case of synchronous realtime processes, a situation of an empty output buffer is an abnormal, pathologic situation in which the items (for example, samples in the case of an audio DAC, or in the case of recording, a CD in your CD-burner) must maintain a strictly regular period when transferred. This situation is also called *'buffer underrun'*.

In short, there are two kinds of *FIFO* buffers: input and output buffers:

1. with an *input buffer*, data items are received from the outside, and *pushed* into the buffer by the input device. The program has the task to *pop* such data items from the buffer in order to process them. In realtime processes, the program has to maintain this buffer empty, or, at least, not full. If the program is not fast enough at popping data, the buffer could became full and some incoming data items could be lost;
2. with an *output buffer*, data items are generated by the program, and *pushed* into the output buffer. In realtime synchronous streaming, the output device *pops* such data items from the buffer at a strictly constant rate. It is a duty of the program to maintain a full buffer, or, at least, not empty. If the program is not fast enough in pushing data, the buffer could became empty and drop-outs could occur, breaking the regularity of the streaming operation.

The larger that the buffer is[16], the greater the security that "buffer underruns" can be avoided. Although there can never be absolute certainty that such a pathological situation won't occur, you can trust that, at a reasonable buffer size, buffer underruns cannot occur in normal situations.

So, why don't we just use a huge buffer for streaming audio in realtime for total security? There is a drawback in using large buffers for realtime processing, and such drawback is directly proportional to the buffer size: it is the so-called '***latency***'. The *theoretical* latency[17] is the average time lag that passes between a push action of a data item in the FIFO buffer and the subsequent pop action of that particular data item. Such latency is directly proportional to the buffer size and inversely proportional to the sampling rate, in the case of audio streams. Normally the latency is calculated in milliseconds, so the maximum latency can be calculated by the following formula:

$$maxLatency = 1000 * bufferSize * blockSize / sampleRate \quad (3)$$

where *bufferSize* is the size of the FIFO buffer (for example, in the case of triple buffering is equal to 3), the *blockSize* is the number of sample frames of each block (for example, 512 stereo frames), and *sampleRate* is the sampling rate (for example, 44100 Hz). In this case the maximum latency is:

$$maxLatency = 1000 * 3 * 512 / 44100 = 34.83\sim ms \quad (4)$$

that is 34.83 ms (maximum latency delay). Now we have to calculate the minimum latency with the following formula:

$$minLatency = 1000 * (bufferSize-1) * blockSize / sampleRate \quad (5)$$

that is:

$$minLatency = 1000 * 2 * 512 / 44100 = 23.22\sim ms \quad (6)$$

The theoretical average latency is obtained by the mean value between maximum and minimum latency:

$$averageLatency = (minLatency + maxLatency) / 2 \quad (7)$$

that is:

$$averageLatency = (34.83\sim + 23.22\sim) / 2 = 29.025\sim ms \quad (8)$$

Remember that this is a theoretical value, the real latency delay is always bigger than the theoretical one. Furthermore, in the case of simultaneous input and output data transfer (full-duplex, sometimes required for digital effects such as guitar fuzz boxes, etc.) the theoretical latency is the sum of the latencies obtained by the input and output buffers and block sizes.

The choice of the appropriate buffer size and block size is a compromise between latency and security from buffer underruns. So, most commercial audio programs, leave it up to the user to customize the optimal buffer size, and this can only be achieved by means of trail and error tests.

Summing up, instead of sending data items directly to the output, there are at least three reasons for using FIFO buffers:

- In synchronous streaming operations, the physical output (or input) device has to guarantee a perfectly constant rate and precise timing, but a computer tends to process data at irregular speeds, so by filling the buffer with a certain number of data items, there will always be at least an item ready to be sent out.
- In both synchronous and asynchronous streaming operations, the data items could be fed faster than the output device is able to handle them, so some items could be lost. By storing the 'overfed' items in the buffer, we guarantee that all items will be recognized, in turn by the output device.
- processing and transferring blocks of data is often faster than processing single data elements one a time.

By accumulating data into the buffer, the processing engine is able to process blocks of data at the maximum speed, without the necessity of waiting for the output port to be ready to receive the datum.

3.4.2 Host-driven and Callback Mechanisms

Since a device driver most often hides the buffering mechanism from the user, the implementation is rarely an issue. Still, a basic knowledge of these techniques is necessary in order to use the device driver's audio or MIDI API. For example, when we used the functions `outBlockInterleaved()` and `outBlockMono()` in the programs *player.c, player2.c, playerGUIsimple.cpp* and *playerGUI.cpp,* there was actually a buffering mechanism hidden in these functions (belonging to the ***Tiny Audio Library***), that was invisible to the final application programmer. However, the programmer had to provide a routine that generated each block of samples and allocated the array that would host such block.

The kind of mechanism provided by the *Tiny Audio Library* is a ***blocking*** mechanism[18], that is, it blocks the program processing each time the output buffer is full (or the input buffer is empty), waiting until new data items are ready to be processed. An API for data in/out with a blocking mechanism is easy to use, because the program is automatically blocked when there is not enough room in the audio buffer for more data items. In this case the write function will just block the program until more room is available, in a transparent way for the user. But this mechanism, even if it is simple to use, is not the most efficient, since its waiting can waste many CPU cycles that could be applied to other jobs in the meanwhile.

There are other much more efficient mechanisms than blocking such as multi-threading, host-driven and callback mechanisms. Often these methods are combined in the same application. We have seen a simple way of using multi-threading in the *playerGUIsimple.cpp* and *PlayerGUI.cpp* programs. In these cases, the creation of a new execution thread has been done to separate the GUI part of the application from the audio engine. A new engine thread was created each time the *'play'* button was clicked, and such thread was terminated (or destroyed) each time the 'stop' button was pushed. Actually the engine itself was made up of a single thread, so it was not really an example of a multi-threading engine in these cases. In fact, a multi-threading engine would take advantage of buffer-full streaming stops for calculating new samples. Such an engine is quite complicated to develop, because any non-trivial multi-threading technique is quite tricky and advanced to develop

A callback mechanism is based on the concept of ***callback function***. A callback function is a function automatically called by the operating system (not by the main flow of the program) at a moment that is appropriate for its own purpose. There are several way to make the operating system calls a callback function, for example, in the Win32 operating system API there are timer objects that, when activated, are able to call a function written by the user at regular intervals of time. The user can set the duration of such time intervals. Other kinds of callback functions can be associated with computer hardware interrupts, or simply called in response to messages generated by a GUI event. We have already seen this sort of callback functions in the *playerGUIsimple.cpp* and *playerGUI.cpp* programs. Often, the best way to learn a new concept is to see it applied to a practical example. This is what we will do now, in order to learn to use the callback mechanism provided by the ***Portaudio*** library. In the next section we will deal with an example of callback mechanism applied to full-duplex audio streaming, based on ***Portaudio***.

3.4.3 Using the Portaudio Library

The *Portaudio* library[19] is a standard, open-source, multi-platform library, which is able to support realtime audio streaming very efficiently, thanks to its integrated callback mechanism. Portaudio wraps most hardware-dependent implementation code into a common API available

for several operating systems, such as Windows, Linux, Solaris, FreeBSD, MacOS and MacOS-X. Since version v.19, *Portaudio* provides multiple *host-API* support in the same binary. This means that, for example, under Windows it can provide *MME*, *DirectX* and *ASIO* support in a single binary build of an application. The final application user can choose, at runtime, between the different host-APIs. Portaudio provides both a *blocking* API[20] and a *callback-mechanism* API. The programmer is free to choose the most appropriate their specific application, even if the callback mechanism is the most efficient by far (but a bit more difficult to program than the blocking mechanism).

Up until this point, we have used the blocking mechanism in our programs; but there are two main advantages in using a callback mechanism instead of a blocking mechanism:

1. it frees the CPU from dead-cycles, when the output buffer is full or the input buffer is empty, thereby optimizing the overall machine performance and allowing the main thread of execution to continue processing while the i/o buffers are flushing (i.e. the main thread is not blocked by the buffer status).
2. helps to reduce latency to a minimum (especially when using low-latency-oriented host APIs such as ASIO or ALSA).

3.4.4 The Callback Mechanism in Practice: HelloRing

The application shown in this section is a console program that implements a realtime ***ring modulator***[21], in which the user sets the modulation frequency. Since ring modulation is such an extremely simple synthesis technique, it frees us to concentrate on the callback mechanism provided by Portaudio.

Our focus will be:

- how to write a full-duplex audio callback function (containing the audio engine) specially suited for Portaudio;
- how to get and display all the available audio devices of the Portaudio-supported host-APIs;
- how to choose a specific device for input and output;
- how to set the parameters of an audio stream (for example, number of channels, sampling rate, block size, and so on);
- how to initialize Portaudio and start the audio callback mechanism;
- how to stop the callback mechanism and cleanup Portaudio.

The user selects the audio input and audio output ports when the program starts, by choosing them from a list of available ports. Since the v19 version of Portaudio, the program user can choose from all the host APIs available on the target machine. This allows the user to chose *MME*, *DirectX* or *ASIO* under Windows, *OSS* or *ALSA* under Linux, and so on.

The program ***HelloRing*** will accept a stereo input: both channels will be modulated by a sinusoidal signal (whose frequency is chosen by the user), and the two modulated channels will be sent to a stereo output.

Here is a descriptive summary of the requirements of the program *helloRing.c*:

Write a console program that:

- *Accepts a stereo audio signal as input, and streams the processed signal to a stereo out in realtime, using the callback mechanism provided by the Portaudio v19 library.*
- *The input signal is processed in realtime by means of the ring-modulation technique, i.e. each channel of the input signal is multiplied by a sinusoidal signal that acts as a modulator. This signal is calculated by means of the sine function provided by the standard C library 'math.h'.*
- *The user of this program is allowed to choose:*
  1. *the frequency of the modulating sinusoid, in Hertz;*
  2. *the input and output audio device among the several host-APIs present in the target machine.*

And here is a description outline of the steps that we will cover to accomplish the task:

1. include the needed library headers;
2. make the external declarations:
   a. define the macro constants of the frame block length, the sampling rate, and the **2π** value (needed for the modulator frequency calculation);
   b. declare the portaudio stream pointer as global;
   c. declare the sampling increment as global;
3. define the callback function that contains the ring modulation engine (that is a loop which multiplies each channel by a sinusoid and outputs the result);
4. define all the initialization stuff inside a function. The initialization function has to:
   a. ask the user the modulator frequency and get it;
   b. calculate the sampling increment and store it in the corresponding global variable;
   c. initialize Portaudio for operation
   d. show all the available audio output devices for each host-API, and ask the user to choose one of them;
   e. fill the corresponding parameter structure. Such structure will contain information about the output stream, such as the chosen port, number of channels, sample format, etc. that will be used for opening the audio stream;
   f. show all the available audio input devices for each host-API, and ask the user to choose one of them;
   g. fill corresponding parameter structure;
   h. open the Portaudio stream;
   i. start the stream;

5. define all the terminate stuff inside a function. The termination function has to:
    a. stop the stream;
    b. close the stream;
    c. terminate Portaudio. This cleans up all internal stuff initiated by Portaudio;
6. define the main() function. The main function has to:
    a. call the init function at start;
    b. begin a dummy loop that stops only when a key is pressed by the user;
    c. call the termination function.

Here are the contents of *helloRing.c*:

```
#include <stdio.h>
#include <math.h>

#include "portaudio.h"

#define FRAME_BLOCK_LEN 256
#define SAMPLING_RATE 44100
#define TWO_PI  (3.14159265f * 2.0f)

PaStream *audioStream;
double si = 0;

int audio_callback( const void *inputBuffer, void *outputBuffer,
                    unsigned long framesPerBuffer,
                    const PaStreamCallbackTimeInfo* timeInfo,
                    PaStreamCallbackFlags statusFlags,
                    void *userData
                  )
{
float *in = (float*) inputBuffer, *out = (float*)outputBuffer;
static double phase = 0;
unsigned long i;
for( i=0; i < framesPerBuffer; i++ ) {
    float sine = sin(phase);
    *out++ = *in++ * sine; /* left channel */
    *out++ = *in++ * sine; /* right channel */
    phase  += si;
}
return paContinue;
}

void init_stuff()
{
 float frequency;
 int i,id;
 const PaDeviceInfo  *info;
 const PaHostApiInfo *hostapi;
 PaStreamParameters outputParameters, inputParameters;

 printf("Type the modulator frequency in Hertz: ");
 scanf("%f", &frequency); /* get the modulator frequency */
```

```
 si = TWO_PI * frequency / SAMPLING_RATE; /* calculate sampling
                                             increment */

 printf("Initializing Portaudio. Please wait...\n");
 Pa_Initialize();   /* initialize portaudio */

 for (i=0;i < Pa_GetDeviceCount(); i++) {
  info = Pa_GetDeviceInfo(i); /* get information from current device */
  hostapi = Pa_GetHostApiInfo(info->hostApi); /*get info from curr.
                  host API */
  if (info->maxOutputChannels > 0)/* if curr device supports output */
     printf("%d: [%s] %s (output)\n",i, hostapi->name, info->name );
  }

  printf("\nType AUDIO output device number: ");
  scanf("%d", &id); /* get the output device id from the user */
  info = Pa_GetDeviceInfo(id); /* get chosen device information
                                  structure */
  hostapi = Pa_GetHostApiInfo(info->hostApi);/*get host API struct */
  printf("Opening AUDIO output device [%s] %s\n",
       hostapi->name, info->name);
outputParameters.device = id; /* chosen device id */
outputParameters.channelCount = 2; /* stereo output */
outputParameters.sampleFormat = paFloat32;    /* 32 bit float output */
outputParameters.suggestedLatency =
            info->defaultLowOutputLatency; /*set default*/
outputParameters.hostApiSpecificStreamInfo = NULL; /*no specific info*/

for (i=0;i < Pa_GetDeviceCount(); i++) {
  info = Pa_GetDeviceInfo(i); /* get information from current device */
  hostapi = Pa_GetHostApiInfo(info->hostApi); /*info from host API */
  if (info->maxInputChannels > 0)  /* if curr device supports input */
      printf("%d: [%s] %s (input)\n",i, hostapi->name, info->name );
    }

printf("\nType AUDIO input device number: ");
scanf("%d", &id); /* get the input device id from the user */
info = Pa_GetDeviceInfo(id); /* get chosen device information struct */
hostapi = Pa_GetHostApiInfo(info->hostApi);  /* get host API struct */
printf("Opening AUDIO input device [%s] %s\n",
           hostapi->name, info->name);

inputParameters.device = id; /* chosen device id */
inputParameters.channelCount = 2; /* stereo input */
inputParameters.sampleFormat = paFloat32; /* 32 bit float input */
inputParameters.suggestedLatency = info->defaultLowInputLatency;
inputParameters.hostApiSpecificStreamInfo = NULL;

Pa_OpenStream(  /* open the PaStream object and get its address */
    &audioStream, /* get the address of the portaudio stream object */
    &inputParameters,  /* provide output parameters */
    &outputParameters, /* provide input parameters */
    SAMPLING_RATE,     /* set sampling rate */
    FRAME_BLOCK_LEN,   /* set frames per buffer */
    paClipOff,         /* set no clip */
    audio_callback,    /* provide the callback function address */
```

```
    NULL );            /* provide no data for the callback */

    Pa_StartStream(audioStream); /* start the callback mechanism */
    printf("running... press space bar and enter to exit\n");
}

void terminate_stuff()
{
Pa_StopStream( audioStream );    /* stop the callback mechanism */
Pa_CloseStream( audioStream );   /* destroy the audio stream object */
Pa_Terminate();                  /* terminate portaudio */
}

int main()
{
init_stuff();
while(getchar() != ' ') Pa_Sleep(100);
terminate_stuff();
return 0;
}
```

At the top, we have the headers and constants used in the code. Particularly important is `FRAME_BLOCK_LEN`, the length, in frames, of the frame block[22] used by the Portaudio callback function. We need to be aware of the fact that the larger the block size, the higher the latency[23]. The `audioStream` pointer (to a Portaudio stream object[24])) is declared global:

```
PaStream *audioStream;
```

This is a handle that encapsulates a relevant number of inner gears, most of which are completely transparent to the API user. Also made global is `si`, which will hold the sampling increment of the modulator.

Next the audio stream callback[25] is defined, which contains the most important code: the engine of the ring modulator. This user-supplied function can have any name but must always have the same number, type and order of arguments as well as the same return type as expected by Portaudio. The callback is not invoked directly by our code, but automatically from the Portaudio inner mechanism. This function receives a pointer to the block of incoming samples and a pointer to the block of outgoing samples from the Portaudio mechanism. It is your task to read and, eventually, process the samples of the input block, as well as to calculate the samples of the audio output block at each function call. Notice that this function has a long list of arguments, and returns an *int*. This is the function head:

```
int audio_callback (void *inputBuffer,void *outputBuffer,
                    unsigned long framesPerBuffer,
                    const PaStreamCallbackTimeInfo* timeInfo,
                    PaStreamCallbackFlags statusFlags,
                    void *userData
                           )
```

However in our case, we only have to care about the first three arguments and the return value (i.e. `inputBuffer`, `outputBuffer` and `framesPerBuffer`), since the others are not used in this program. Nevertheless, the `Pa_OpenStream()` API[26] function requires the

address of a function having such a prototype as an argument (so we must declare it as above). That function will register our callback with Portaudio.

Concentrating now on the code of `audio_callback()`, let's review its code:

```
int audio_callback( const void *inputBuffer, void *outputBuffer,
                    unsigned long framesPerBuffer,
                    const PaStreamCallbackTimeInfo* timeInfo,
                    PaStreamCallbackFlags statusFlags,
                    void *userData
                  )
{
float *in = (float*) inputBuffer, *out = (float*)outputBuffer;
static double phase = 0;
unsigned long i;
for( i=0; i < framesPerBuffer; i++ ) {
    float sine = sin(phase);
    *out++ = *in++ * sine; /* left channel */
    *out++ = *in++ * sine; /* right channel */
    phase  += si;
}
return paContinue;
}
```

The arguments `inputBuffer` and `outputBuffer` are pointers that contain the addresses of the input and output blocks[27] of sample frames. Such variables are void pointers. In order to use them, they must be cast to a pointer of a concrete type. Since, in this case, we plan to deal with samples of 32-bit floating-point format[28], we have to cast them to `float` pointers (`in` and `out`). Since the audio stream was opened as a stereo stream, each buffer frame is made up of two interleaved[29] float samples. The `in` and `out` buffer can be considered as arrays of stereo frames, each having `framesPerBuffer` length. The current phase of the modulator is stored in a static double variable, initialized to zero at the start of the program[30].

The `for` loop is our audio engine, which is iterated `framesPerBuffer` times. Here, the modulator is obtained by calculating the sinus of current phase value, and the output is stored in the temporary `sine` variable. Then the left input channel is ring-modulated (i.e. multiplied) by the sine output and the output is stored to current buffer (i.e. frame block) location. Notice the indirection and post-increment operators of the `out` and `in` pointers:

```
*out++    =    *in++ ……
```

They are indirection operators (a unary prefix operator whose purpose is to access, for writing or reading, the location currently pointed to by the corresponding pointer). Don't confuse them with the asterisk of the multiplication operator (which is a binary infix operator) that is placed in between the `in` pointer and the `sine` variable:

```
*in++  *   sine
```

since this operator simply multiplies the two items. In this statement,

```
*out++ = *in++ * sine; /* left channel */
```

the value of the input buffer location, having the `in` pointer as an address, is extracted and multiplied by the current `sine` value. The result is copied to the output buffer location having the `out` pointer as an address. *After* this assignment, the `in` and `out` pointers are both incremented (be aware that the increment operation is done on the addresses contained in the pointers, not on the values of the pointed locations), and consequently point to the next locations of corresponding buffers. This statement is applied to the left channel. Given that the buffer is made up of interleaved frames, an identical statement is provided for the right channel. Because of its perceived efficiency, this use of buffer pointer increment is very common in audio applications[31].

At the last line of the loop, current phase value is incremented by the sampling increment. After the loop has processed all the frames of the input and output blocks in this way, the `audio_callback()` function returns the ***paContinue*** API enumerator constant[32].

Our `main()` function is minimal:

```
int main()
{
init_stuff();
while(getchar() != ' ') Pa_Sleep(100); /* wait
                        for space-bar and enter */
terminate_stuff();
return 0;
}
```

It calls the `init_stuff()` function, which gets the modulator frequency and the audio ports from the user, initializes everything, opens the audio stream, and starts the Portaudio callback process.

Then it starts a while loop that calls the `Pa_Sleep()` API function. This function blocks the main thread of execution for 100 milliseconds (this value represents its argument) and yields, for this time interval, all the CPU time to the other threads and processes (among which there is the Portaudio callback mechanism). This loop has, as a testing condition, `getchar() != ' '`. This function checks the terminal for keyboard input: if the user has pressed space bar followed by enter, the condition becomes false (i.e. `getchar()` returns the `space` character) and the loop is exited. In this case, `terminate_stuff()` function is called to stop the callback process and destroy the `PaStream` object.

Now let's look into the details of the `init_stuff()` function. It contains the code to create the `PaStream` object and to start the callback process. We get the frequency of the modulator from the user and calculate the sampling increment (`si`), used in `audio_callback()` to increment the phase of the modulator.

The Portaudio library is initialized by calling the `Pa_Initialize()` API function, which must be called before any other API function belonging to Portaudio. The loop:

```
for (i=0;i < Pa_GetDeviceCount(); i++) {
  info = Pa_GetDeviceInfo(i); /* get information from current device */
  hostapi = Pa_GetHostApiInfo(info->hostApi); /*get info from curr.
                  host API */
  if (info->maxOutputChannels > 0)/* if curr device supports output */
     printf("%d: [%s] %s (output)\n",i, hostapi->name, info->name );
```

```
}
```

is used to display all the available audio output devices. These devices include all available host APIs supported by Portaudio (for example, MME, DirectX and ASIO, in the Windows case, OSS and ALSA in Linux). The loop is iterated the number of times equal to the value returned by the `Pa_GetDeviceCount()` API function, which provides the number of available devices belonging to all host APIs.

At each loop iteration, the `info` pointer is filled with the (read-only, or const) address of a variable belonging of the type `PaDeviceInfo`, by calling `Pa_GetDeviceInfo()`.This contains details about a corresponding audio device. Here are all the members of a `PaDeviceInfo` structure[33]:

```
typedef struct PaDeviceInfo {
    int structVersion;
    const char *name;
    PaHostApiIndex hostApi;
    int maxInputChannels;
    int maxOutputChannels;
    PaTime defaultLowInputLatency;
    PaTime defaultLowOutputLatency;
    PaTime defaultHighInputLatency;
    PaTime defaultHighOutputLatency;
    double defaultSampleRate;
} PaDeviceInfo;
```

The relevant members in our case are `name`, `hostApi` and `maxOutputChannels`. The `name` member is a pointer to a string containing the name of current device; the `hostApi` member contains an *`int`* value that expresses the host API index of the current audio device. With this, we can get information about a host API by calling `Pa_GetHostApiInfo()`. This function returns a (read-only, or const) pointer to `PaHostApiInfo`, which contains some host API information. Here are all the members of `PaHostApiInfo` structure[34] `(of which we only need `name`):

```
typedef struct PaHostApiInfo {
    int structVersion;
    PaHostApiTypeId type;
    const char *name;
    int deviceCount;
    PaDeviceIndex defaultInputDevice;
    PaDeviceIndex defaultOutputDevice;
} PaHostApiInfo;
```

The code inside the loop also tests the `maxOutputChannels` member of `PaDeviceInfo`:

```
if (info->maxOutputChannels > 0)
 /* if curr device supports output */
```

```
printf("%d: [%s] %s (output)\n",i,
            hostapi->name, info->name );
```

This is done to know if the current audio device supports *output* channels. If this test is true, i.e., if there is a number of output channels greater than zero, a line of text is printed to the console, displaying: the audio device number, its host API, its name and whether it is an output device. On my personal computer, for instance, this is the output of the loop displaying the output devices:

```
7:  [MME] Microsoft Sound Mapper - Output (output)
8:  [MME] MOTU Analog 1-2 (output)
9:  [MME] MOTU Analog 3-4 (output)
10: [MME] MOTU Analog 5-6 (output)
11: [MME] MOTU Analog 7-8 (output)
12: [MME] MOTU TOSLink 1-2 (output)
13: [MME] Avance AC97 Audio (output)
21: [Windows DirectSound] Driver audio principale (output)
22: [Windows DirectSound] MOTU Analog 1-2 (output)
23: [Windows DirectSound] MOTU Analog 3-4 (output)
24: [Windows DirectSound] MOTU Analog 5-6 (output)
25: [Windows DirectSound] MOTU Analog 7-8 (output)
26: [Windows DirectSound] MOTU TOSLink 1-2 (output)
27: [Windows DirectSound] Avance AC97 Audio (output)
28: [ASIO] ASIO DirectX Full Duplex Driver (output)
29: [ASIO] ASIO Multimedia Driver (output)
30: [ASIO] MOTU FireWire Audio (output)
```

Notice how the number of the device begins with 7 (not with zero) and skips several numbers: these devices are skipped because they contain zero output channels (i.e. they don't support audio output). After the list is shown, the user is asked to type the number of the output device they intends to choose and the `id` variable is filled with that number. Information on the chosen device is then printed to the terminal.

Once we have an output device, then we can set it up. We do this by filling the data members of the `outputParameter` variable, which has the type `PaStreamParameters`[35]:

```
typedef struct PaStreamParameters {
    PaDeviceIndex device;
    int channelCount;
    PaSampleFormat sampleFormat;
    PaTime suggestedLatency;
    void *hostApiSpecificStreamInfo;
} PaStreamParameters;
```

The information contained in this structure is the following:

- `device`: an *int* number (chosen by the program user) that expresses the index of a supported audio device. Its range is 0 to (`Pa_GetDeviceCount() - 1`).
- `channelCount`: the number of audio channels to be delivered to the stream callback. It can range from 1 to the value of `maxInputChannels` in the `PaDeviceInfo` structure for the device specified by the device parameter.

- `sampleFormat`: the sample format of the buffer provided to the stream callback. It may be any of the formats supported by Portaudio[36].
- `suggestedLatency`: The desired latency in seconds. Be aware that an audio device could not support the value of latency set by the user, and so provide a latency quite different than this setting. The actual latency values for an already-opened stream can be retrieved using the *inputLatency* and *outputLatency* members of the `PaStreamInfo` structure returned by the `Pa_GetStreamInfo()` API function.
- `hostApiSpecificStreamInfo`: is an optional pointer to a host API specific data structure containing additional information for device setup and/or stream processing. Often not used, as in *helloRing.c*, in which case it must be set to `NULL`.

The address of `outputParameter` structure will be provided as an argument of the `Pa_OpenStream()` API function, which creates the audio stream object. In addition to output, we must select and set-up audio input. We will proceeed similarly by filling the `inputParameters` variable. We can then pass the addresses of both variables to the `Pa_OpenStream()` API function[37]:

```
Pa_OpenStream(  /* open the PaStream object and get its address */
    &audioStream, /* get the address of the portaudio stream object */
    &inputParameters,  /* provide output parameters */
    &outputParameters, /* provide input parameters */
    SAMPLING_RATE,     /* set sampling rate */
    FRAME_BLOCK_LEN,   /* set frames per buffer */
    paClipOff,         /* set no clip */
    audio_callback,    /* provide the callback function address */
    NULL );            /* provide no data for the callback */

    Pa_StartStream(audioStream); /* start the callback mechanism */
      }
```

This function has the task of creating (opening) an audio stream object, capable of starting the callback process. Notice that the address of the `audioStream` pointer is passed as an argument. After a call to this function, it will point to the audio stream object, which is now open.

The `paClipOff` item is a bitwise flag that can be used together with other flags[38] to describe the behavior of the callback process. Normally, Portaudio clips[39] out-of-range samples. Since we are dealing with floating-point samples, `paClipOff` disables default clipping of the samples. To set more than one flag, it is possible to use a bitwise OR operator '|' between flag identifiers. For example, to set the `paClipOff` and the `paDitherOff` flags at the same time, one would provide the following expression as an argument:

```
paClipOff  |  paDitherOff
```

We will also provide the `audio_callback` function address[40] as an argument so that the `Pa_OpenStream()` function can register it with Portaudio. The last argument is a pointer

to an optional user-defined data pointer that is passed at each call to the audio callback function. In *helloRing.c* we don't make use of this feature, so the pointer is set to `NULL`.

At the end of the `init_stuff()` function, the callback mechanism is started by a call to the `Pa_StartStream()` API function. It will automatically call the `audio_callback()` function every time the input and output buffers need to be filled, in a completely transparent way[41]. The callback mechanism stops when the `terminate_stuff()` function is called (in response to a user keypress, as discussed above):

```
void terminate_stuff()
{
Pa_StopStream( audioStream );     /* stop the callback mechanism */
Pa_CloseStream( audioStream );    /* destroy the audio stream object */
Pa_Terminate();                   /* terminate portaudio */
}
```

Both the `Pa_StopStream()` and `Pa_CloseStream()` get the pointer to the audio stream object as an argument. The first stops audio streaming, and the second close the stream object (invalidating corresponding pointer). The `Pa_Terminate()` function cleans up Portaudio.

3.5 Conclusion

In this chapter, we have explored the concept of audio streams, which is central to audio processing and programming. We have looked at several examples first producing streams as text, then as binary data, to soundfiles and finally realtime audio. This text should have provided the basis for further exploration of the manipulation of audio signals, as discussed in other chapters of this book and DVD-ROM.

3.6 Exercises

Ex.3.6.1 The waveform-drawing functions `fill_square(),fill_saw()` and `fill_triangle()` in HelloTable are quite basic and have a major flaw: they are not bandlimited. This means that the sound synthesized with these always contain aliased components, which contribute to the low quality of the sound. How would you go about generating band-limited versions of these flawed functions? Please implement these and then substitute them in the example program, comparing the result.

Ex.3.6.2 The audio callback used in *HelloRing* includes the line:

```
static double phase = 0;
```

that declares a static variable. The use of static variables is highly discouraged in modern programming, as it can cause problems in large systems. So, in keeping with a better style of programming, please

(a) remove the static definition

(b) substitute it with some externally-provided memory (hint: the userData argument to the callback is never used. Perhaps you could try giving it something to do!).

Rebuild and test your program. You should now be happier that you have converted a nasty-looking lazy feature into something much more clever…

Ex.3.6.3 Implement a stereo ring-modulator, which will have two oscillators, one for each channel, at different frequencies. Remember that each oscillator will have to keep its phase and increment separate. In complement to what you have done in exercise 3, remove another lazy feature of *HelloRing*: the use of a global variable to hold the sampling increment. Substitute it for two local variables (to `main`), which will keep the separate increments and use some means of passing their value to the callback. (Hint: you can group the two increments and two phases in a data structure).

3.7 Notes

[1] 'Signal' is referred to here with its technical meaning, such as a function describing variations of voltage in an analog device, or the flow of related numbers describing the course of some physical quantity, such as fluctuating air pressure, plotted and displayed as an exponential curve on a computer screen (see the *Introduction to Digital Signals* chapter for more background).

[2] In my chapters, the term 'sound', will not always be used with its appropriate meaning, but rather, it will often serve as a synonym for 'audio signal'.

[3] The '***.c***' file extension of "*helloworld.c*" and "*hellosine.c*" indicates that they are C language source files. Other extensions we will deal with in this book are: '***.h***' (include files of both C and C++ languages), '***.cpp***' (C++ language source files) and '***.hpp***' (C++ header files).

[4] The "Hello world!" program is famous because it appeared as the first example in "The C Programming Language" by B.W.Kernighan and D.M.Ritchie, the creators of the C Programming Language. Their book is the 'official' C programming tutorial and the most renowned book about the C programming language ever written.

[5] A sample is the numerical representation of the instantaneous value of a signal, in this case of the air pressure, given that our signal is a sound signal. It is a single numeric value and a sequence of samples makes up a digital audio signal.

[6] We will see what 'standard output' is later in this chapter.

[7] In the C language, a ***table*** is normally implemented as an *array*. Later in this book you will learn what arrays are and their purpose in detail; for now, it is sufficient to know that an *array* is nothing more than a sequence of computer memory locations containing items of the same kind. In the case of audio *tables*, these items are numbers that represent the samples of a sampled sound or of a synthesized wave.

[8] Some non-standard libraries are needed: the first, the ***portsf.lib*** (also known as ***Dobson Audio Lib***) provides several useful routines that deal with reading/writing audio files from/to disk; the second, the ***Tiny Audio Lib*** has already been used for *hellotable.c*, and serves to provide an easy-to-use API for DAC/soundcard real-time audio in/out.

[9] *stdio.h* for the *printf()* function, *portsf.h* for special audio file functions such as *psf_init(), psf_sndOpen(), psf_sndReadFloatFrames(), psf_sndClose(), psf_finish(),*and *tinyAudioLib.h* for *outBlockInterleaved()* and *outBlockMono()*.

[10] You can find the sources of *crack.c* in the provided DVD-ROM.

[11] Actually, in the previous programs *player.c, player2.c, playerGUIsimple.cpp* and *playerGUI.cpp*, used the function:

```
long psf_sndReadFloatFrames(int sfd, float *buf, DWORD nFrames);
```

from the ***portsf*** library, instead of our hypothetical `get_new_data()` function. The function `psf_sndReadFloatFrames( )` reads a block of samples from a wave file and copies them into a previously-allocated memory block. In this case, the user must provide the address of the block in the *buf* pointer before the function call.

[12] This is not to be confused with the term as it is used when referring to analog recording tape, in which a *drop-out* is a sudden loss of amplitude. In digital audio, a *drop-out* means a total absence of signal due to the temporary lack of available samples to convert. This is much more noticeable and disturbing than in the analog case, because loud clicks can be heard.

[13] A block of items of the same type (for example a block of samples) is often called a 'buffer'. So, in our case, the term 'buffers of samples' refers to the sample block. Consequently, using a multiple buffering technique, means we can deal with something like 'buffering of buffers'. For example, assuming that we deal with blocks of 512 samples, in the case of double buffering, we have a buffer made up of two data items, each one made up of a block (or buffer) of 512 samples. In the case of triple buffering we deal with a buffer of three elements (i.e. three blocks of samples) and so on.

[14] There is another buffer type, these are called *LIFO* (**L**ast-**I**n **F**irst-**O**ut) *buffers* or ***stacks***. They are used in other cases, but not in streaming data items to an output.

[15] In most cases, the processing speed of a computer routine is not constant: in certain moments it can be extremely fast, whereas in others it can remain blocked for a limited time or flow extremely slowly. There are several reasons for this, for example: in an operating system there are normally many concurrent processes completely independent from each others, sometimes a process needs more CPU cycles than the other ones and vice-versa in other moments; the cache memory of the processor can sometimes help to increase the speed of a loop, and other times might not be available, because it is busy with other processing routines, and so on. For these reasons some kind of buffering scheme is a necessary prerequisite for realtime streaming.

[16] The unit of measure for buffer size is the number of data items that can be contained by the buffer itself. For example, a buffer of 512 elements is obviously bigger than a buffer of 256 elements.

[17] Such latency calculation is theoretical because, in practice, there is another time lag you have to add to obtain the real latency delay, this lag is determined by the hardware device driver internal mechanism, that is completely hidden from the programmer. Also, sound has a limited propagation velocity in the air medium, so, depending on speaker distance from the listener, the real latency can be proportionally higher. Therefore, real latency delay is always bigger than the theoretical one, obtained by counting the FIFO buffer's data items.

[18] Remember that the Tiny Audio Library wraps the blocking part of the Portaudio library, in order to make its use simpler for beginners. A more efficient and versatile (but even more difficult) way can be obtained by using the Portaudio library directly, as will be the case in the following applications. Portaudio not only supports a blocking mechanism, but also a much more efficient callback mechanism, as we will see later.

[19] Copyright (c) 1999-2000 Ross Bencina and Phil Burk.

[20] The *blocking* API is not as efficient as the *callback-based* API. But it is mainly provided to simplify the programming and development of simple audio utility programs. It consist of a single thread of execution, whose flow is blocked each time an in/out function is called, i.e. such function doesn't return until the corresponding buffers has been flushed. The *Tiny Audio Library* we have used so far is based on the blocking API of Portaudio. For real-world tasks, the callback-based API is highly recommended. In this chapter we deal with the callback-based API of Portaudio only.

[21] The *ring modulation* technique is perhaps the simplest way to modify and process an input signal. Actually, this technique consists of a single multiplication of two signals, the *carrier* and the *modulator*. In a multiplication, the order of the operands is not significant, this means that modulator and carrier are symmetric, i.e. they are swappable without affecting the resulting output. The resulting spectrum of the output signal is made up of the sums and the differences of all the partial frequencies of the two starting signals. Ring modulation is very similar to the *amplitude modulation* technique, the only difference is that, in the latter, a DC offset is added to the modulator, in order to make it positive only. The spectrum of the amplitude modulation contains not only the sums and differences of the frequencies of all carrier and modulator partials (as in the case of ring modulation), but also the original frequencies.

[22] In cases like this, the term *'buffer'* is often exchanged with the term *'block'*, in which case they are synonymous.

[23] Latency doesn't depend solely on the block size, but also on how many blocks are used. The exact number of blocks used at runtime is transparent to the user of the Portaudio API, so that you do not have to worry about it. In any case, the user of this API does have some control over this number, by setting an approximate suggestion of the latency time by means of specific parameters (we will see such parameters later in this chapter).

[24] A `PaStream` object has nothing to do with C++ classes and objects. It is actually a C structure containing pointers, pointers to functions and data that apply in all inner mechanisms of Portaudio. Its type is initially a null pointer.

[25] Here is the type definition of an audio stream callback function in Portaudio v19:

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

[26] The API functions are, in this case, the library functions belonging to the Portaudio library. The `audio_callback()` function is not an API function, because it is written by the Portaudio user, and doesn't belong to the Portaudio library.

[27] In this case, 'Blocks' and 'buffers' are synonymous.

[28] Current Portaudio implementation (v.19) supports blocks of frames having any number of channels, and the following sample formats, expressed by symbolic macro constants:

`paFloat32`	→ 32-bit floating-point samples (floats)
`paInt32`	→ 32-bit signed integers (longs)
`paInt24`	→ Packed 24-bit format
`paInt16`	→ 16-bit signed integers (shorts)

`paInt8`	→ 8-bit signed integers (chars)
`UInt8`	→ 8-bit unsigned integers (unsigned chars)

Even if in *helloRing.c,* we chose a 32-bit floating-point sample format, such a format may not be natively supported by you audio interface (and almost surely it isn't, because most audio interfaces are based on 16-bit or 24-bit integer sample format). However, this fact won't prevent *helloRing.c* from working with your interface because there is a Portaudio internal setting that automatically converts the sample format into a format compatible with your interface, so you needn't worry about explicitly using a sample format natively supported by your interface.

[29] Normally Portaudio uses intereleaved blocks, made up of frames, in which each frame contains a sample of all channels. However, since v19, Portaudio now optionally supports non-interleaved blocks. By using the `paNonInterleaved` macro constant, one sets a bitwise flag which informs the engine that the user doesn't want a single default frame block for all channels, but rather, multiple separated non-interleaved sample blocks, one for each channel. When non-interleaved blocks are chose, the callback arguments `void *inputBuffer` and `void *outputBuffer` don't point to a single sample-frame block, but rather, to an array of sample blocks. And consequently, the void pointers must be cast in the following way inside the audio callback:

```
float *left = ((float **) inputBuffer)[0];
float *right = ((float **) inputBuffer)[1];
```

[30] Remember that local static variables are created at the start of the program and keep their content unchanged between function calls. This is the reason we used a static variable in this context. So, the variable `phase` will keep the last value assumed in each subsequent function call, and you have to be aware that it is set to zero only at the program start, not each time the audio_callback function is called (different from non-static local variables).

[31] Current compilers can generate machine code with the same efficiency, even when using the array style instead of incrementing the pointers. In this case the loop would assume the following style:

```
for( i=0; i < framesPerBuffer; i+=2 ) {
    float sine = sin(phase);
    out[i]   = in[i]   * sine; /* left */
    out[i+1] = in[i+1] * sine; /* right */
    phase  += si;
}
```

Another reason of convenience in using the pointer-increment style is how easy it is to modify the code for multi-channel audio. For example, if we had to support quadraphonic output, the loop would simply be as follows:

```
for( i=0; i < framesPerBuffer; i++ ) {
      float sine =  sin(phase);
      *out++ = *in++ * sine; /* front left */
      *out++ = *in++ * sine; /* front right */
      *out++ = *in++ * sine; /* rear left */
      *out++ = *in++ * sine; /* rear right */
      phase  += si;
}
```

[32] The `paContinue` API enumerator constant (belonging to the `PaStreamCallbackResult` enumeration type) is equal to zero in Portaudio v19. This value informs the Portaudio mechanism that the callback process has to

be continued. There are two other possible return values of the callback function: `paComplete` (equal to one), which informs that the process has completed, so there is no necessity for further calls to the callback functions; and `paAbort` (equal to two), which informs that the callback process must be stopped.

[33] See the ***portaudio.h*** header file, provided with the Portaudio distribution, for more information and documentation about this structure.

[34] As above.

[35] As above.

[36] See note 31.

[37] This is the prototype of the `Pa_OpenStream()` API function:

```
PaError   Pa_OpenStream( PaStream** stream,
const PaStreamParameters *inputParameters,
const PaStreamParameters *outputParameters,
double sampleRate, unsigned long framesPerBuffer,
PaStreamFlags streamFlags, PaStreamCallback *streamCallback,
void *userData );
```

Notice that this function returns an error value, not used in *helloRing.c*.

[38] The possible flags are:

`paNoFlag` → no flag is set
`paClipOff` → disable clipping
`paDitherOff` → disable dithering
`paNeverDropInput` → will not discard overflowed input samples without calling the stream callback
`paPrimeOutputBuffersUsingStreamCallback` → call the stream callback to fill initial output buffers, rather than the default behavior of priming the buffers with zeros (silence).
`paPlatformSpecificFlags` → platform specific flags

By default, Portaudio clips and dithers the samples.

[39] Clipping means making all samples above and below certain thresholds equal to the corresponding threshold values. For example if the allowed range is -1 to +1, and a sample has a value of .8, it passes unmodified by the clipping stage, but if a sample has a value of 1.2 it will be clipped to 1, and if a sample has a value of -2 it will be clipped to -1. By default, the Portaudio mechanism clips samples. Upper and lower clipping limits can vary, depending on the sample formats:

`paFloat32` → -1 to 1
`paInt32` → the limits of a *long* variable (–2,147,483,648 to 2,147,483,647)

`paInt24` → (depends on the implementation, normally the limits are the same of *paInt32*, but the real values are approximate, because the lower 8 bits are truncated)

`paInt16` → the limits of a *short* variable (–32,768 to 32,767)

`paInt8` → the limits of a *char* variable (–128 to 127)

`paUInt8` → the limits of an *unsigned char* variable (0 to 255)

[40] To get the address of a function, it is sufficient to set the function name without round braces, and so, to get the address of the `audio_callback()` function, we can simply provide the `audio_callback` identifier. This identifier returns the address of the target function in both C and C++ languages.

[41] Once the `Pa_StartStream()` function has been successfully called, the callback function is repeatedly called without us needing to worry about doing this explicitly. In the case of *helloRing.c,* there is no need to control anything more, because the modulator frequency is constant. If we had to control it in realtime, for example via MIDI, some control mechanism would have to be activated and performed during the callback process. Part of this control mechanism can reside in the audio callback function, while other part could reside elsewhere (for example, in an eventual MIDI callback function). But often all control gears reside in the audio callback too.