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

```
$ 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 the program 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 two 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 four 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` (an 8-bit integer, often used to express a character) that 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 will stop.

Next, an array of 1,024 `float` elements (`FRAME_BLOCK_LEN * 2`) is declared and defined:

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

This array is the buffer that will contain the block of 512 frames read from the audio file. It is dimensioned to 1,024, 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 section of code 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 (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.

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 able to return only a single argument, but there is a trick to make it possible to 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. To enable this, 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, which 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 more than two channels, the program prints an error message and exits , 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 subsection. This player will allow us to specify both the starting point and the duration of the file to play. It also provides 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: 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, and `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, clean up the 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 2chans 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 the 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).

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. The flag 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` (which 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 the file *crack.c*, so is compiled together with *player2.c*). Being a library function, `crack()` will be used as a black box—that is, we have only to know what it does, not how it functions internally.

Finally, we have the declarations of three flags: the `flag` variable (used as temporary storage of starting option flags provided by the user when starting the program at the command-line shell), the `timflag` variable (which signals if the `-t` option has been provided), and the `durflag` variable (which signals if the `-d` option has been provided). These two variables are linked to `startime` and `dur`, respectively, which will contain the start time and the 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)
```

We can see that this function has one output argument and four 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, every option flag 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, which in this case must be followed by a numeric datum indicating the starting point (for `-t` flag) and the duration (for `-d` flag) in seconds. In the string provided in the third argument of `crack()`, the flags we define are `-t`, `-d`, `-T`, and `-D` (upper-case letters meaning the same thing as lower-case letters). The | character indicates that other data must follow the corresponding flag letter, without spaces between the flag letter and the datum in the command line.

The last input argument of `crack()` is a boolean flag indicating whether `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 `-T`), then `timflag` is set to `1` (i.e. *true*, meaning 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 arise. 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 displays an error message). This will happen if the user provides a flag other than `-t` or `-d`.

If the user has not provided an 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. If the result is true, we assign the value immediately after the question mark; otherwise we assign the value after the colon. Such expressions are used quite often in C.

In case the user did not provide the `-d` option flag, `endpoint` is assigned the total length of the file and `dur` 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 only supports mono and stereo files. The *portsf* library function call

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

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`, which moves the pointer starting from current position, and `PSF_SEEK_END`, which counts backward 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 real time 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.

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] which 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 the user doesn't have to worry about it. The user need only fill the buffer with the proper data items (in musical cases these items are sample frames, MIDI messages, or control signal frames) and provide the address of the buffer. The `outBlock()` function call doesn't return until the block of sample frames has been sent

these flawed functions? Please implement these and then substitute them in the example program, comparing the results.

Exercise 3.6.2

The audio callback used in *HelloRing.c* includes the line

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

which declares a static variable. The use of static variables is highly discouraged in modern programming, as it can cause problems in large systems. In keeping with a better style of programming, do the following:

(a) Remove the static definition.
(b) Replace the static definition 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.

Exercise 3.6.3

Implement a stereo ring modulator with two oscillators, one for each channel, at different frequencies. Remember that each oscillator will have to keep its phase and its increment separate. In complement to what you have done in exercise 2, remove another simplistic feature of *HelloRing*: the use of a global variable to hold the sampling increment. Substitute for it 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.)

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 chapter 5 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*, a book 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.

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* on the DVD.

11. -, 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, '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* (Last-In First-Out) *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 concur-