In the previous chapter we learned how to play and display "raw" binary soundfiles using Audacity. In order to play such a soundfile, we needed to tell the program, not only the sample rate, sample type and channel count; but also the endianness of the samples themselves. The purpose of a dedicated soundfile format such as WAVE and AIFF is to encapsulate this information in a header section prefixed to the audio data itself, which an application can read to determine the format of the file. Such a file format can thus be described as "self-describing." Writing code to read and write such soundfiles is a complicated job, not least because of the many alternative formats a soundfile can have. There is a clear trade-off for the audio programmer. It is easy to write a raw soundfile, as we have seen, but such files are only portable with the greatest difficulty, and more cumbersome to play. The possibility of forgetting what format a given file has is very great. Conversely, writing the code to write (and more particularly, to read) a self-describing file format is a non-trivial task, especially where that format is very complex with many variations and options.

Fortunately, several C libraries have been created that deal with all the arcane details of writing modern soundfile formats, so that reading and writing such files need be no more difficult than reading or writing a text file. The formats define all the above information, and often much more, including optional information that while not essential to play a file, is nevertheless useful.

This chapter demonstrates how to generate simple waveforms, such as a triangle wave and a sine wave, and write them to a soundfile, and also how to modify or combine sounds. In the process we are introduced to further elements of the C language, and to further techniques of code design. As you may suspect, these programs will be longer, and more complex than before, as many more tasks are being performed. This complexity includes the use of multiple source files, and the use of external libraries that need to be linked to our program. This is, however, such a common task that many tools have been developed over the years to streamline the process as much as possible. One such tool is the *makefile*, a text file that defines all the elements and actions needed to build a program, together with any dependencies such as external libraries.
Behind the details of new C keywords, the arithmetic of sine waves, and the definition of
new functions lie deeper and increasingly important issues of robustness, flexibility, extensi-
bility, and, embracing all these, comprehensibility. As our programs develop in sophistica-
tion, so will our ambitions as programmers rise, and it becomes more and more likely that
in writing any one program we will anticipate many further possibilities.
In particular, just as the programs in this section make use of a pre-written code library,
you will yourself be writing many functions that are likely to be required for many different
projects, and which could therefore also form components of your own library. So in reading
the following sections, consider not only the immediate task at hand, but also how general a
block of code or a function is, and whether you might find uses for it in other projects.
This chapter therefore assumes and expects that you will now be contemplating, at least,
writing your own programs, above and beyond the examples given here. You will be creating
building blocks of code; and it almost goes without saying that building blocks can be used
to make many very different edifices. The challenge is in designing those blocks for maxi-
mum re-usability, while ensuring that they are also as easy to use as possible. It remains one
of the most interesting and intriguing exercises for the programmer, to anticipate the future
use of something. Often, new ideas demand that a function be rewritten \textit{a posteriori}. Yet in
writing a program the programmer’s first thought is understandably, to “get it working,”
and worry about other possibilities later. As a result, it can often happen that a program is
written twice. The first is a “proof of concept” version that works; the second version learns
from the first and implements enhancements, alternative approaches to design, increases
modularity and robustness, and identifies blocks of code worth generalizing into a library
for use elsewhere. Once this is done, the program may even be rewritten a third time. This
is normal.

2.1 A Simple Soundfile Library: \textit{portsf}

2.1.1 The \textit{Ports}f Header Files—Public and Private Files

The \textit{portsf} soundfile library, developed especially for this book, was designed to be as simple
as possible to use. It is not intended in any way to be an alternative to “industrial-strength”
libraries such as \textit{libsndfile}. It comprises only two source files, each with an associated header
file:

\begin{verbatim}
ieee80.c
ieee80.h
portsf.c
portsf.h
\end{verbatim}

This relatively small size reflects the fact that only the two most common soundfile formats
are supported, AIFF and WAVE, though this includes 24-bit and 32-bit formats (both integer
and float), and multi-channel files, including the new WAVE_FORMAT_EXTENSIBLE format introduced by Microsoft to support surround sound and high-precision audio (e.g. 24-bit and beyond). Libraries such as libsdfile (demonstrated elsewhere in this book) support a very wide range of formats (including compressed formats such as ADPCM), and are highly efficient, but are also very much larger, using a large number of source files, with code that is very much less accessible to beginning programmers.

The two files ieee80.c and ieee80.h are very small (the header file especially so), and are made distinct as a courtesy to the author, Bill Gardner of MIT and Wave Arts (you can read the details in the header file). Both are “private” files required by portsf.c (specifically, to deal with the sample-rate field in AIFF headers). There is no technical reason why these files could not be concatenated into one; you might like to try this as an exercise. However, in the form shown, they conveniently illustrate an important distinction between private and public header files.

To use the library, the only header file the user needs to #include is portsf.h—this defines all the structures and functions in the library. We will be making frequent reference to this file in the following sections. As this is a public header file, we will use the angle brackets with #include:

```c
#include <portsf.h>
```

This header file defines all we need to know about the portsf library—various custom types and structures, and the specialized functions for handling sound files. The main implementation file portsf.c on the other hand is large and complex, making use of several C language idioms not so far presented. It will nevertheless repay study. You will for example notice immediately the use of conditional compilation (as introduced in the previous chapter) using #ifdef and #ifndef. With long headers such as this, any technique that can be used to avoid duplication of effort is very welcome. In this case, the conditional compilation is used to detect if portsf.h has already been included by another source file. If so, the whole body of the file is skipped. As it happens, this also enables the file to be read by C++ programs, C++ being especially disapproving of duplicated type definitions. If at this stage you are alarmed by portsf.c and feel you do not understand it, the answer is that it doesn’t matter (much). Often, code for a general-purpose library is very complex, not least because it seeks to be general-purpose; but all we need to know about are the various definitions and declarations in the header file.

Unlike portsf.h, the header file ieee80.h is, conversely, a private file used internally by the two C files, and if you inspect these, you will notice that the other form of the #include directive is used there:

```c
#include "ieee80.h"
```

This, you will recall from chapter 1, asks the compiler to search first in the directory that contains the current source file (i.e. the file containing the #include directive).
When portsf is created as a true library module, the files ieee80.h and ieee80.c, together with portsf.c, will in effect disappear; the main header file portsf.h will be your only means of access to the library. Thus the ieee80-related functions are private to the library, not accessible to the programmer, and in some future implementation may be redefined, or maybe even eliminated completely. This is a form of code encapsulation. If the user depended directly on the existence and exact form of these functions they could not be removed or modified without breaking user projects. In the following sections you will discover further mechanisms in C for supporting such encapsulation (you may also encounter the expression “data hiding” in this context); the language C++ develops this much further into a core principle of software design.

2.1.2 The Makefile

For the first time, we require a special (non-system) header file stored not in our project directory but elsewhere. As a result we will need to tell the compiler where to find it, by adding the directory containing required header files (their path) to the list to be searched by the compiler. The procedure for doing this is compiler-specific, but whichever one you use will give you an option to set this “search path” for header files. The recommended approach in this book is to use the UNIX-style development system based on the use of a “makefile.” This includes Linux, OS X and the MinGW environment on Windows. On the DVD, a makefile is provided for all example programs in each chapter.

A makefile defines all the stages of compiling a program—or indeed of several programs; it is read by a special command-line program called make, standard on all UNIX-based systems (including OS X). You will find that a single makefile supports building all the programs in a chapter. Typing make will automatically build all of them (building portsf as well if necessary). For projects involving a number of source files, external libraries, and other special build options (such as whether to make a “debug build,” or a “release build” optimized for speed), such a makefile is essential. The proprietary project files used by compilers such as Visual Studio or Xcode are themselves in essence makefiles, albeit typically somewhat extended and verbose. One important task handled by make, in conjunction with the makefile, is tracking whether source files need to be rebuilt or not—i.e. by noting the creation date of object files and other dependencies. If a project comprises dozens (or hundreds) of source files, this speeds up development considerably by avoiding unnecessary recompilation.2

The supplied makefiles are written specifically for the directory structure as found on the DVD—the paths are relative to the examples directory, for example using the “. . /” notation to indicate the parent directory. Later on, you may feel confident enough to create your own project directory structure, and create or modify makefiles accordingly. You will use make in conjunction with your makefile to ensure that common files and libraries exist in only one place, but can be used in any project by setting their paths in the makefile. You will see that all the makefiles have much in common, with file and program names being the main differ-
ences. Eventually you will become confident enough to write your own makefile, by modifying one of these examples.

Despite the observations above, try to resist the temptation to copy all of the examples from the DVD together, or to build all the programs at once. A key element of learning to program lies in physically writing code—creating new files, maybe new directories, even modifying a makefile to support a new program. In some cases, the text presents a preliminary version of a program that is not provided as such on the DVD. It is especially important to follow the detailed step-by-step descriptions of the process, adding and changing code as you go. It is only through this direct hands-on experience that you will gain the confidence to work independently on your own projects.

2.1.3 Soundfile Formats—enum and typedef

We have already encountered the enum keyword for defining a list of symbols, associating them in order with ascending integer values. This offers a convenient alternative to a possibly long list of individual definitions using the preprocessor directive #define. It is also possible to apply the typedef keyword to the enum statement, so that, in principle, each symbol in the enum list is no longer the default int, but has the new custom type. This results in a more expressive form of self-documentation than if, for example, a plain int were to be used, or a list of #defines. The new type can then be employed both for function arguments and return values. For example, ports.h defines an enum type for a number of soundfile channel properties:

```c
typedef enum { STDWAVE, MC_STD, MC_MONO, MC_STEREO, MC_QUAD,
               MC_LCRS, MC_BFMT, MC_DOLBY_5_1, MC_WAVE_EX }
               psf_channelformat;
```

This defines the possible speaker formats supported by ports. It includes support for the most common WAVEFORMATEXTENSIBLE speaker position definitions (meaningful only for the WAVE format). The symbol STD_WAVE serves as a “default” value, representing any “standard” soundfile with no special speaker positions—any soundfile format can be represented this way. Additionally, the symbol MC_WAVE_EX acts as a category for any other speaker-feed combinations not otherwise supported explicitly.

Similarly, ports.h defines enum types for the available file formats and the all-important sample type:

```c
typedef enum {
    PSP_FMT_UNKNOWN = 0,
    PSP_STDWAVE,
    PSP_WAVE_EX,
    PSP_AIFF,
    PSP_AIFC
} psf_format;
```
typedef enum {
    PSF_SAMP_UNKNOWN = 0,
    PSF_SAMP_8,  /* not yet supported! */
    PSF_SAMP_16,
    PSF_SAMP_24,
    PSF_SAMP_32,
    PSF_SAMP_IEEE_FLOAT
} psf_stype;

The significance of distinguishing PSF_STDWAVE and PSF_WAVE_EX in psf_format is that files in these formats have the same file extension, .wav, so it is not sufficient to deduce the file format solely from the file name. Similarly, it is not uncommon for a file with the extension .aif or .aiff to be in the newer AIFF-C format, despite the fact that Apple advocates a distinct extension—.afc or .aifc.

These custom types are then used in the main structure defined in portsf.h to contain the essential properties of a soundfile, again using typedef:

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

This custom type is used variously as argument and return value in a number of functions in portsf. When a soundfile is opened for reading, a PSF_PROPS structure is automatically filled in; when a file is written, the programmer must likewise ensure that all fields contain valid values. Taken together, the custom types and the portsf functions define an application programming interface (API). An API is intended to hide all the low-level implementation details (e.g. byte-swapping), while providing sufficient flexibility and power to support a range of projects. We will see how all the elements of the portsf API work together in the following sections, which begin with a demonstration program using the library to convert a soundfile from one format to another. For more details, see appendix C.

There is a limit to the amount of documentation a developer can supply with a library—some degree of prior knowledge has to be assumed. In the case of portsf, it is assumed the user is familiar with the use of soundfiles, and with the use of WAVE and AIFF soundfiles in particular. The structure PSF_PROPS introduced in the previous section collects this essential knowledge into one object; the fact that it contains only five elements demonstrates that one need not know very much about a soundfile in order to be able to use it. This knowledge can be summarized as follows:
A soundfile contains audio data at a particular sample rate (number of samples per second). Most audio systems support a range of standard sample rates—22,050, 44,100, 48,000, 96,000 and even 192,000 hertz for DVD audio. We can store this parameter in

PSF_PROPS structure element: srate

A soundfile can contain several channels of audio—one for mono, two for stereo, six for Dolby 5.1, and so on. In a multi-channel file, the sample rate defines the frame rate, where one frame contains one sample for each channel. As a soundfile represents audio that has either been recorded through, or is intended to be played back through, a soundcard, these samples are necessarily interleaved, see figure 1.12 in chapter 1. An alternative view of interleaved samples is shown in figure 2.1. This is represented by the following structure element:

PSF_PROPS element: chans

A soundfile can contain audio samples in a range of data formats. By far the most common format is 16-bit, represented by the C short data type. Other formats in widespread use include 24-bit "packed" (stored in three bytes), and 32-bit floats, represented by the C float.
data type. Less common, but possible, is the 32-bit integer format, represented (at least in a
32-bit platform) by the C long (and, usually, int) data type. Other formats are possible: the
older 8-bit sample type is still in use here and there, but is not currently supported in portsf,
though provision has been made for its support at some future date. A 64-bit floating-point
sample is possible in both the AIFF and WAVE formats, but this is currently not supported by
portsf. A more exotic sample format is supported by the WAVEFORMATEXTENSIBLE file for-
mat. This distinguishes between the actual sample size and the "container size." A typical
example of this would be a 24-bit "unpacked" format in which the three bytes of the sample
are contained in a 32-bit word. Such formats (which will be very rare in a soundfile) are not
currently supported in portsf, which records only the overall container size. Note the first
symbol PSF_SAMP_UNKNOWN, which is used to indicate that a soundfile has some unrecog-
nized sample type.

PSF_PROPS element: samptype

- A soundfile can be written in a number of file formats. One reason for this variety is the
different storage formats on different platforms (see note 1). The WAVE and AIFF formats
are the most familiar, but there are many others, only two of which are currently supported
by portsf. The new WAVEFORMATEXTENSIBLE format from Microsoft has already been
mentioned—this is intended to replace WAVE, and includes support for multi-channel audio
with defined speaker positions, and support for high-resolution sample formats including
the concept of a container size, as described above. It also supports user-defined custom for-
mats (new versions of WAVE had to be registered with Microsoft; the new format enables
anyone to define a new custom format without this requirement). You can find full technical
documentation on this format at http://www.microsoft.com/hwdev/tech/audio/multichaud.
as. Additionally, the new AIFF-C format is supported. This was originally designed by
Apple to support a range of compressed sample types, but a recent extension added support
for a 32-bit floating-point format, which the original AIFF format could not support. The
portsf library does not support compressed audio types in any of the file formats. Again, it
is possible that a user may seek to open or create a file in an unsupported format; accord-
ingly, this is provided for in the custom format entry PSF_FMT_UNKNOWN.

- A soundfile can contain audio channels associated with specific speaker positions. This
property is relatively new, but has greatly increased in importance thanks to the popularity
of surround sound. With a stereo signal, it is established largely by convention that channel
1 is front left, and channel 2 is front right. Beyond that, speaker positions were undefined
in the standard WAVE format, and it was primarily for this reason that the WAVEFORMAT-
EXTENSIBLE format was introduced by Microsoft. Apple had, in the initial specification for
the AIFF format, proposed standard speaker positions for several surround formats, but in so
doing highlighted the difficulties in doing so. Two four-channel configurations were defined,
for quad surround and LCRS Surround, but there was no means of indicating which of these
was contained in a given soundfile—a four-channel AIFF file is therefore ambiguous, accord-
ing to its own specification. At the time of writing, the only soundfile formats unambiguously supporting defined speaker positions are WAVEFORMATEXTENSIBLE, and the new CAF file format introduced by Apple (not supported by portsf, but supported by libsndfile). The portsf library supports only the primary subset of many possible channel configurations, as defined for psf_chformat. One symbol in that list, MC_BFMT, requires further explanation. This refers to a custom version of WAVEFORMATEXTENSIBLE designed to hold an Ambisonic B-Format signal. This is an advanced form of surround encoding, representing the specially processed outputs of three coincident figure-eight microphones (oriented to the three spatial axes), which together can record a complete spherical “soundfield”—this includes information about the height of a sound, as well as its distance and position in the horizontal plane. This format therefore contrasts with the other channel formats in that the channels represent not speaker feeds, but a compact encoding of a full three-dimensional soundfield, which has to be decoded before being passed to loudspeakers. It is possible to decode to speaker layouts containing a large number of speakers; a minimum of four speakers would be required to decode the signals for horizontal surround sound, and a minimum of eight speakers in a cube layout, to decode with height.

PSF_PROPS element: chformat

This it is currently only relevant when the file format is PSF_WAVE_EX; it is ignored for all other formats (STDNWave is assumed).

Finally, one important property of all the file formats supported by portsf is that the file header (which stores the properties of the file such as sample rate, sample type and number of channels) can also contain further optional information contained in special “chunks”—collectively, the AIFF and WAVE formats store all information in distinct chunks, each of which has a unique name (both “WAVE” and “AIFF” are examples of chunk names). We will see in the next section how portsf includes support for one very useful optional chunk.

2.1.4 Initializing the portsf Library

Somewhat unusually for a soundfile library (and unlike any of the file i/o functions in the C standard library), portsf includes an initialization function, to be called before any other functions in the library are called:

```c
int psf_init(void);
```

Currently this function does very little, but it is expected that future versions of the library will be more dependent on it. Portsf maintains an internal block of information on all active soundfiles, up to an internal fixed limit, currently set at 64. The function psf_init ensures that this block is in a clean state at startup. There is a corresponding function that should be called at the end of a program, as part of final “cleaning up”:

```c
int psf_finish(void);
```
This closes any soundfiles that have been left open and returns the library to the same clean state. In command-line programs, such things do not matter (as the operating system will close any open files anyway when the program terminates); but in GUI-based programs that can in effect restart multiple times (they are "re-entrant"), these functions will ensure that the library is always in a stable state.

Both functions return the value 0 to indicate success. Currently, psf_init() cannot in fact fail, so no harm is done in ignoring the return value. The function psf_finish() will return a non-zero value if there is an error closing a file. In future versions of the library, both these functions may increase in importance.

### 2.1.5 Basic Soundfile Handling in ports—the switch...case Keywords

A soundfile API of any usefulness must support the basic tasks associated with all file handling—opening and creating files, reading and writing data, and closing them again. Error reporting is also essential—for example, in the case that a user tries to open a file that doesn't exist, or tries to write to a file that is not open, and also in the case of a system error such as a disk failure. The API may also offer other useful secondary facilities.

#### Opening a Soundfile for Reading

The function to open an existing soundfile is shown in ports/h as

```
int psf_sndOpen(const char *path, PSF_PROPS *props, int rescale);
```

The comments in the header file indicate how this function is to be used:

```
/* open existing soundfile. Receive format info in props.
   Supports auto rescale from PEAK data, with floats files.
   Only RDONLY access supported.
   Return sf descriptor >= 0, or some PSF_E_+ on error.
*/
```

Though somewhat terse, these comments provide quite a lot of information:

1. "open existing soundfile" tells the programmer that this function cannot be used to create a new file.
2. "only RDONLY access supported" tells the programmer that you can read from the opened file but cannot write to it. "RDONLY" is standard C shorthand for "Read ONLY" (echoing the standard C file i/o API). Thus, ports enforce "non-destructive" processing—you cannot modify an existing soundfile.
3. "Return sf descriptor >= 0" tells the programmer that the int value returned from the function will be a value that identifies the file (for future use with other functions). Any value below zero will signify an error. The ports/h header file defines a long list of possible error values (needless to say, as an enum list). Most programmers will just test the value against zero, and if negative, will simply tell the user "there was an error," but ideally the
specific error value can be used to give the user a more informative message, such as "that is not a soundfile."

The comments also describe the input arguments to the function. The first of these should be fairly obvious—the name of the file. The second is almost as obvious—a pointer to the PSF_PROPS structure described in subsection 2.1.3; the comments indicate that if the function succeeds, the structure will then be filled with information describing the file.

The third argument is, however, very strange-sounding, and is an example of a non-essential but sometimes useful facility. The comments refer to "PEAK data." This is a reference to an optional header chunk that a WAVE or AIFF file can contain, which records the value and position of the loudest sample, for each channel in the file. We encountered a prototype for this facility at the end of chapter 1 (see listing 1.9.4). In portsf this is encapsulated as an automatic facility. It is especially useful with floating-point soundfiles. While the "normal" range of a floating-point sample is ±1.0, it is also possible to have numbers larger than that—"over-range samples." You would expect that playing such a soundfile would result in unpleasant clipping and distortion. However, if the file contains the PEAK chunk, the information about the largest sample can be used to rescale the audio data automatically, to bring it within the normal range. This facility is represented by the rescale argument to psf_sndOpen. If rescale is non-zero (one would usually just use the value 1), rescaling is applied, otherwise samples will be read unaltered. Of course, if the file does not contain a PEAK chunk, the rescale argument is simply ignored. This facility is provided primarily for use in soundfile playback applications, where over-range samples will necessarily be clipped. By reading the PEAK data from the header when the file is opened, portsf can automatically rescale the audio data so that such soundfiles can be rendered without clipping. In most other situations, we want to read in the data unchanged, so that the rescale argument to psf_sndOpen would be set to zero.

To open an existing soundfile, therefore, we need an instance of the PSF_PROPS structure, to receive format information about the file, and an int variable ("soundfile descriptor"), to receive the return value from psf_sndOpen. Given, for example, a 16-bit stereo WAVE file at 44,100 Hz, called "sample.wav," we can open it as follows, and give the user some information about it:

```c
PSF_PROPS props;
int sf;

sf = psf_sndOpen("sample.wav", &props, 0);
if(sf < 0){
    printf("Error: unable to open soundfile\n");
    return 1;
}

printf("Sample rate = %d\n", props.srate);
printf("number of channels = %d\n", props.chans);
```
However, the use of custom types in PSF_PROPS leads to a difficulty. The function printf() cannot know anything about this type—were we to try to display it using the method above, it would merely write whatever number PSF_SAMP_16 happened to have been given by the compiler—not very informative to the user. Instead, we need to write code to test for each possible value, and print an appropriate message:

```c
if(props_srate == PSF_SAMP_8)
    printf("sample type: 8bit\n");
else if(props_srate == PSF_SAMP_16)
    printf("sample type: 16bit\n");
...
```

We encountered this potentially long sequence of if...else tests in chapter 1. To handle such situations a little more elegantly, C provides the keywords switch and case, which together provide a flexible system for selecting among alternatives, where the variable being tested is of an integral type, such as char, short or int (recall that C treats all enum types as being equivalent to int). It is demonstrated in listing 2.1.2 in the form of a function

```c
int check_sampletype(psf_srate srate);
```

that will display the sample type of the file and will return a value indicating whether we can accept it or not. The function can be used to replace the code suggested above:

```c
if(!check_sampletype(props_srate))
    printf("file has unsupported sample type\n")1
    return 1;
}
```

**Listing 2.1.2: Selecting Among Alternatives—the switch...case Keywords**

```c
1  int check_sampletype(psf_srate srate)
2  {
3      int accept = 1;
4
5      printf("sample type: ");
6      switch(srate){
7          case(PSF_SAMP_8):
8              printf("8 bit\n");
9              accept = 0; /* No 8bit files! */
10         break;
11          case(PSF_SAMP_16):
12              printf("16 bit\n");
13         break;
14          case(PSF_SAMP_24):
15              printf("24 bit\n");
```
break;
17 case(PSF_SAMP_32):
18 printf("32bit (integer)\n");
19 break;
20 case(PSF_SAMP_IEEE_FLOAT):
21 printf("32bit floating point\n");
22 break;
23 default:
24 printf("unknown\n");
25 accept = 0;
26 }
27 return accept;
28 }
29 }

There are several things to note about the use of switch and case:

(1) The keyword switch takes a single argument of any integral type (enclosed in curved brackets), followed by a code block enclosed by braces (lines 1, 2, 29).
(2) The code block can contain one or more "selection statements," using either the case (e.g. line 7) or default (line 23) keywords. Only one default statement can be used, but any number of case statements can be used.
(3) The case keyword takes a single argument, signifying a specific value of the variable used as the argument to switch, followed by a colon, and then by any number of statements to be executed (e.g. lines 7-10). Execution then passes to the following selection statement, or reaches the end of the switch block. The default keyword (also followed by a colon) takes no argument—it applies to all values not matched by the provided case statements.
(4) To stop execution "falling through" from one selection statement to the next, the break keyword can be used to jump out of the switch block (e.g. line 10).
(5) Selection statements can be in any order. For example, the default statement can be placed first.
(6) As the code belonging to each selection statement is deemed to be just one level below the switch expression, the switch and case expressions are usually given the same indentation. The programmer is free to change this, however.
(7) The case keyword can only be used inside a switch statement block.

Opening a Soundfile for Writing
To create a new soundfile, we would expect a similarly named function, and in portsf this function is somewhat more elaborate than psf_sndOpen:

```c
int psf_sndCreate(const char *path, const PSF_PROPS *props,
int clip_floats, int minheader, int mode);
```
(1) The first two arguments are the same as those for `psf_sndOpen`, except that the second is also declared as const, indicating that the `PSF_PROPS` data is now an input, and will not be modified. We will have to fill this structure with the required format data for the file, before passing it to `psf_sndCreate`.

(2) The argument `int clip_floats` is used to set the way in which floating-point data is written to the file. As was noted above, the unique aspect of floating-point soundfiles is that the samples can contain over-range values. Depending on the application, you may or many not want these to be clipped to the normal maxima, -1.0 and +1.0. Use of this facility depends on whether you have requested that the soundfile include the PEAK chunk, which records the maximum values in the file. As not all applications will even know about the PEAK chunk (and will simply ignore it when reading), the safe approach is to set this argument to 1; but for experimental purposes you may want, for example, to offer the user the choice. Needless to say, this parameter is ignored for all other sample formats.

(3) `int minheader`: it is an unfortunate fact of life that many applications fail to deal with WAVE formats that contain optional chunks before the audio data—many older UNIX-originated programs suffer from this. By setting `minheader` to 1, the soundfile is created with a "minimum header" containing just the required format and data chunks—this therefore means that no PEAK data will be written to the file. Ideally, of course, `minheader` should be set to 0 always, and will be in all the examples presented here.

(4) `int mode`: this final argument is provided to offer some control over read-write access to the file. The possible modes are defined by the custom enum type `psf_create_mode` defined in `portfs.h`:

```c
/* second two are speculative at present! */
typedef enum {PSF_CREATE_RDWR, PSF_CREATE_TEMPORARY,
              PSF_CREATE_RDONLY } psf_create_mode;
```

The comment indicates that in fact only the first option is supported, to create a 'normal' file with public read/write access. The other options seek to create 'private' soundfiles, in one form or another, and may be implemented in future versions of `portsf`.

To create a soundfile, we need, as for `psf_sndOpen()`, to define a `PSF_PROPS` variable, with the essential difference that this time we have to fill in the format properties explicitly—if we have not already obtained format information from another file:

```c
int ofd;
PSF_PROPS props;
/* define a hi-res 5.1 surround WAVE-EX file, with PEAK chunk support */
props.srate = 96000;
props.chans = 6;
props.samptype = PSF_SAMP_24;
props.format = PSF_WAVE_EX;
props.chformat = MC_DOLBY_5_1;
```
ofd = psf_sndCreate("soundtrack.wav", &props, 1, 0, PSF_CREATE_RDWR);
if(ofd < 0){
    printf("Error: unable to create output file\n");
    return 1;
}

Setting a File Format from the Name
A common requirement among users who have to move soundfiles between platforms (e.g. from a Macintosh to a PC) is to convert from one format to another—for example, from WAVE to AIFF. But how does the user indicate this? In command-line applications (and, arguably, in general), the simplest and best method is for the user to specify the format by using a filename with the appropriate file extension; .wav, .aiff, .aifc, etc. With this approach, there is no danger of a user specifying, for example, the WAVE format but giving a filename with the .aiff extension.\textsuperscript{3} The \texttt{portf} API supports this system by means of a function that returns the file format from the given filename:

\begin{verbatim}
psf_format format;
format = psf_getFormatExt("soundtrack.wav");
\end{verbatim}

This returns a value of type \texttt{psf\_format}, which can therefore be assigned directly to the appropriate element of the \texttt{PSF\_PROPS} structure:

\begin{verbatim}
props.format = format;
\end{verbatim}

Note that this function will return the value \texttt{PSF\_FMT\_UNKNOWN} if an unsupported file extension (or no extension at all) is used.\textsuperscript{4}

Closing Soundfiles (and Recording Peak Sample Values with the PEAK Chunk)
Having opened a soundfile, we will have to close it again at some point. We would expect to find a function to do this in the library; one that takes as an argument the descriptor obtained from \texttt{psf\_sndOpen}. Indeed, on examining \texttt{portf.h} we find just such a function:

\begin{verbatim}
int psf_sndClose(int sfd);
\end{verbatim}

The return value is used to indicate any error arising from closing the file. Occasionally, it is important to tell the user that the function has failed, and why, as it may indicate a problem with the system. In a command-line program, this is generally not considered so important, especially right at the end of a program, so the return value is ignored. One interesting aspect of this function is indicated by the following comment:

\begin{verbatim}/* . . . Updates PEAK data if used. */\end{verbatim}

As sample frames are written to disk, the maximum value per channel is automatically tracked, so that when the file is closed, the PEAK data can be written to the header. For this to happen, of course, \texttt{minHeader} in \texttt{psf\_sndCreate} must be set to 0, as shown above. To
support access to this information, *portsf.h* includes a very simple structure (defined again as a custom type using *typedef*) to hold the peak data for one channel. The structure contains the sample value itself, and the position (in sample frames) in the file of the first such sample:

```c
typedef struct psf_chpeak {
    float val;
    unsigned long pos;
} PSF_CHPEAK;
```

The library necessarily includes a function to read the current PEAK data from the file:

```c
long psf_sndReadPeaks(int sfid, PSF_CHPEAK peakdata[], long *peaktime);
```

This function takes a soundfile descriptor (as returned from *psf_sndCreate*), and a pointer to an array of *PSF_CHPEAK* structures. It is the responsibility of the user to ensure that the array is at least large enough to contain the data for each channel in the file (e.g. as given in *props.chans*):

```c
PSF_CHPEAK peaks[6];
long peaks_valid, peaktime;
peaks_valid = psf_sndReadPeaks(ofd, peaks, &peaktime);
```

In most cases, the number of channels in a file will have been chosen by the user, so that you will have to allocate space for the peaks array using *malloc*:

```c
PSF_CHPEAK* peaks;
peaks = (PSF_CHPEAK*) malloc(props.chans * sizeof(PSF_CHPEAK));
```

The *peaktime* value is rarely needed and can generally be ignored. It corresponds to a field in the PEAK chunk itself, which indicates the time when the PEAK data was written to the file. This time can be checked (a task that is often system dependent, and which is beyond the scope of the present chapter), and if it is clearly different from the modification time of the file as a whole (i.e. substantially older), it indicates that an attempt to modify the audio data in place may have been made (i.e. using “destructive editing”), and therefore that the PEAK data may be unreliable. If you do not need the peaktime information, it is safe to supply *NULL* as the third argument to *psf_sndReadPeaks*:

```c
peaks_valid = psf_sndReadPeaks(ofd, peaks, NULL);
```

### 2.1.6 Reading and Writing—The Sample Frame

The working unit in *portsf* is the multi-channel sample frame. This means that it is not possible, for example, to read just the first (left-channel) sample of a stereo file. This would in all but the most abnormal circumstances be an error, and *portsf* prevents the user from making such errors.
More importantly, the library automatically performs all the low-level calculations and conversions required to read sample frames (of any supported type) from a soundfile into the user's working sample buffer. For this, the recommended sample format is the 32-bit float, into which 24-bit samples can be converted without loss of precision:

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

It is the user's responsibility to supply a buffer of at least $nFrames * props.chans * sizeof(float)$ bytes.

The functions both return the value $-1$ for an error. The function `psf_sndWriteFloatFrames` will not return a positive value less than $nFrames$. In the case of `psf_sndReadFloatFrames`, a return value less than $nFrames$ will indicate that the end of the file has been reached. For the same reason, a return value of $0$ does not signify an error. Programs should expect this and always use the return value in future calculations. Note that the $nFrames$ argument is defined as a custom `DWORD` type. This name is borrowed from Windows usage, and signifies (on a 32-bit platform) an unsigned `long`. Clearly, a negative value for $nFrames$ has no meaning—it is best to prevent such usage altogether.

We can now implement a simple copy procedure with very few lines of code:

```c
float frame* = (float*) malloc(props.chans * sizeof(float));
/* copy file one (multi-channel) frame at a time */
framesread = psf_sndReadFloatFrames(ifd, frame, l);
while (framesread == 1) {
    /* <---- do any processing here! ------> */
    psf_sndWriteFloatFrames(ofd, frame, l);
    framesread = psf_sndReadFloatFrames(ifd, frame, l);
}
```

This will copy a soundfile having any number of channels, and in any of the supported sample formats, while also tracking the PEAK value and position for each channel. As the comment indicates, in a program that performs any processing, this compact loop probably will expand to accommodate code to apply filtering or other signal processing. To change the file and sample format from that of the input file, all that is required is to modify the `PSF_PROPS` structure as desired; `portsf` will take care of all the details.

**2.1.7 Streamlining Error Handling—The goto Keyword**

The code shown above excludes necessary checks for read and write errors, which a practical program must always contain. Even a basic soundfile copy program requests at least three resources from the system—two files and some memory via a call to `malloc`. Any or all of these requests can fail, and to be fully responsible we must release all acquired resources
when we encounter such an error before quitting the program. We have already seen some examples of this in previous programs, where a call to `fclose` or to `free` may appear several times, in code that detects different errors.

While being responsible, programmers also want to be economical, and such duplicated code seems wasteful. It can also become somewhat complicated—depending on the error, different combinations of resources need to be released. Ideally, we want to place all error recovery at one point (e.g. at the end of the program), and simply jump to that point on detection of an error, at which point we release all resources still active. The keyword `goto` does just this, jumping unconditionally to a statement marked by a label:

```c
if(sndWriteFloatFrames(ofd,frame,1) != 1)
    goto cleanup;

/* lots of code. ... */
cleanup:
    if(frame)
        free(frame);
```

The label is also referred to as a “jump target,” and takes the form of a unique name followed by a colon. It can be placed anywhere, but by convention, and not least to promote readability, it is always placed at the start of a line.

The one important restriction with `goto` is that the jump target must be in the same function as the `goto` itself—typically this will be the main function. Thus you cannot jump out of one function into another—which is probably just as well.5

2.1.8 Using portsf for Soundfile Conversion with PEAK Support

The conversion program shown in listing 2.1.3 requires the `portsf` library, either by compiling with all the `portsf` source files or by accessing it as a pre-built library. The latter approach is implemented in the examples folder on the DVD. This build procedure is outlined below, which should be read in conjunction with the 02BOOKexamples-guide.doc programmer's guide found in the chapter 2 “examples” folder on the DVD.

Listing 2.1.3 can be regarded as a simple template or skeleton program for any application that applies a process to a sound, non-destructively. As was noted in the comments (line 88), such processing would be placed inside the main loop copying sample frames. It is worth taking careful note not only of the details of the C language idioms employed, but also of the overall structure. We are already moving to the situation where we are quite comfortable with the C language itself, and pay more and more attention to the strategic and design aspects of programming.

As we have come to expect, the bulk of the code deals with error checking, and messages to the user—`goto` is used to direct all error situations to the cleanup code at the end of the file (lines 111–119). The secret to this approach is to ensure that all relevant variables
(soundfile descriptors, pointers to allocated memory) are initialized to default states (lines 12-16); the cleanup code can then easily discover which resources are active and need to be released. Note that not all errors result in a call to goto: this is only used once a resource such as the first open file has been acquired (line 51). One small detail is the variable error that is incremented when an error is detected. This is only used to ensure that the main function returns the value 0 for success, or 1 for an error.

The code demonstrates a useful aspect of the printf function—it is possible to write several strings in sequence (e.g. lines 21-23), inside one printf call. Note that the strings are not separated by commas or other characters, but only by (optional) white space.

In line 55, the function psf_sndCreate is called with the clip_floats argument set to zero. Coupled with the fact that psf_sndOpen is called with rescale also set to zero, any floating-point input file will be copied to the outfile with any over-range samples copied unaltered. It is of course simple to change this behavior (see exercises, below). The program reports the PEAK values at the end (lines 99-109).6

Listing 2.1.3: sf2float.c

```c
1  /* sf2float.c : convert soundfile to floats */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <portsf.h>
5  
6  enum {ARG_PROGNAME, ARG_INFILE, ARG_OUTFILE, ARG_NARGS};
7  
8  int main(int argc, char* argv[])
9  {
10     PSF_PROPS props;
11     long framesread,totalread;
12     /* init all resource vars to default states */
13     int ifd = -1, ofd = -1;
14     int error = 0;
15     psf_format outformat = PSF_FMT_UNKNOWN;
16     PSF_CHPEAK* peaks = NULL;
17     float* frame = NULL;
18     
19     printf("SF2FLOAT: convert soundfile to floats format\n");
20     
21     if(argc < ARG_NARGS) {
22         printf("insufficient arguments.\n"
23             "usage:\n\tsf2float infile outfile\n");
24         return 1;
25     }
```

/* be good, and startup portsf */
if(psf_init()){
    printf("unable to start portsf\n");
    return 1;
}

ifd = psf_sndOpen(argv[ARG_INFILE],&props,0);
if(ifd < 0){
    printf("Error: unable to open infile %s\n", argv[ARG_INFILE]);
    return 1;
}

/* we now have a resource, so we use goto hereafter
on hitting any error */
/* tell user if source file is already floats */
if(props.samptype == PSF_SAMP_IEEE_FLOAT){
    printf("Info: infile is already in floats format.\n");
}
props.samptype = PSF_SAMP_IEEE_FLOAT;
/* check outfile extension is one we know about */
outformat = psf_getFormatExt(argv[ARG_OUTFILE]);
if(outformat == PSF_FMT_UNKNOWN){
    printf("outfile name %s has unknown format.\n" "Use any of .wav, .aiff, .aif, .afc, .aifc\n", argv[ARG_OUTFILE]);
    error++;
    goto exit;
}
props.format = outformat;

ofd = psf_sndCreate(argv[2],&props,0,0,PSF_CREATE_RDWR);
if(ofd < 0){
    printf("Error: unable to create outfile %s\n", argv[ARG_OUTFILE]);
    error++;
    goto exit;
}

/* allocate space for one sample frame */
frame = (float*) malloc(props.chans * sizeof(float));
if(frame==NULL){
    puts("No memory!\n");
}
```c
error++;
goto exit;
}

/* and allocate space for PEAK info */
peaks = (PSF_PEAK *) malloc(props.chans * sizeof(PSF_PEAK));
if(peaks == NULL){
    puts("No memory\n");
    error++;
goto exit;
}
printf("copying....\n");

/* single-frame loop to do copy, report any errors */
framesread = psf_sndReadFloatFrames(ifd,frame,1);
totalread = 0; /* running count of sample frames */
while (framesread == 1){
totalread++;
    if(psf_sndWriteFloatFrames(ofd,frame,1) != 1){
        printf("Error writing to outfile\n");
        error++;
    break;
}
    /* <---- do any processing here! -------> */
    framesread = psf_sndReadFloatFrames(ifd,frame,1);
}
if(framesread < 0) {
    printf("Error reading infile. Outfile is incomplete.\n");
    error++;
}
else
    printf("Done. %d sample frames copied to %s\n", 
totalread, argv[ARG_OUTFILE]);
/* report PEAK values to user */
if(psf_sndReadPeaks(ofd,peaks,NULL) > 0){
    long l;
    double peaktime;
    printf("PEAK information:\n");
    for(i=0;i < props.chans;i++){
        peaktime = (double) peaks[i].pos / props.srate;
        printf("CH %d: @%.4f at %.4f secs\n", 
i+1, peaks[i].val, peaktime);
    }
```
112  }
113  /* do all cleanup */
114  exit:
115  if(ifd >= 0)
116     psf_sndClose(ifd);
117  if(ofd >= 0)
118     psf_sndClose(ofd);
119  if(frame)
120     free(frame);
121  if(peaks)
122     free(peaks);
123  psf_finish();
124  return error;
125 

2.1.9 Building Programs with *portsf*

There are two approaches readers can take in building *sf2float* and later programs using *portsf*. Both are command-line based. The first, very simple approach, is to copy the chapter 2 examples directory (complete with its sub-directories) from the DVD to any convenient location, cd to the chapter 2 examples directory and run make on the selected program:

    $ make sf2float

This will first build *portsf* (if not already built) and install the library file *libportsf.a* in the directory *lib*. It will then build *sf2float*. This is covered in more detail in the accompanying guide document. Note that, as required for all such development, you will have at least three basic directories—*include* for your header files, *lib* for your libraries, and one or more working directories for your projects. As described above, relative paths are used—so long as the whole examples directory is moved as a unit, the programs will build as described.

The alternative approach is more demanding, but offers a much deeper insight into the build process, including a first step at editing a makefile (see exercise 2.1.7). The goal is to create a standard working environment supporting all your programming projects, whether based on *portsf* or otherwise.7

Test the program first by running it without arguments, to confirm that the usage message is displayed:

    ./sf2float

Then test using any suitable soundfile. This would primarily be a 16-bit WAVE or AIFF file. Use Audacity to confirm that the new file is correct. However it is as important to test the error handling, e.g. by supplying a soundfile in an unsupported format such as a raw binary file, or using an *outfile* name with an unsupported extension.
2.1.10 Exercises

Many of these exercises invite you to discover things on your own. If you find that you are stuck, consider the possibility that other programs presented in this book may well demonstrate solutions to these exercises.

Exercise 2.1.1
In the program sf2float, the main loop to copy samples does so one frame at a time. This is not very efficient. Modify the code to use a larger multi-frame buffer, so that the loop cycles per block rather than per frame. Remember to pay attention to the return value from psf_sndReadFloatFrames.

Exercise 2.1.2
The program sf2float does not report the progress of the copy to the screen. For a long file, the program may appear to have frozen. Remedy this by adding a variable to count the number of samples copied, and display a progress message at some regular interval. Hint: Use the format specifier "\r" (carriage) within printf to overwrite the same line with the updating message string.

Exercise 2.1.3
Study the description of the function psf_sndSize in ports.h. Modify the program to copy up to some limit (which must be less than or equal to the length of the file) defined by the user on the command line.

Exercise 2.1.4
(a) Based on the examples shown in this section, add a function to give a complete display of the infile properties at the start of the program. Include the size of the file in this information. Use the switch...case keywords to print information on the file format and sample type.
(b) Adapt this into a new program sfprop that just reports the format information of a soundfile. Report the PEAK information if present. You may find this useful as a utility to verify output soundfiles are as intended.

Exercise 2.1.5
Sound level is often better expressed in decibels (dB), than as simple amplitude. Full-scale (1.0) is equivalent to 0 dB, and half-amplitude (0.5) is equal to approximately –6 dB. The calculation to convert a normalized amplitude (within ± 1.0) to dB is

\[
\text{loudness (dB)} = 20.0 \times \log_{10}(\text{amp});
\]
Refer to your compiler documentation regarding the math library function (defined in `<math.h>`):

```c
double log10(double x);
```

Use this function to modify the display of the PEAK data at the end of `sf2float.c` (lines 106 and 107) to display the PEAK value for each channel in dB, as well as raw amplitude. If you have built `sffprop` (Ex 4(b) above), modify that in the same way.

**Exercise 2.1.6 (difficult)**

Study the description of the function `psf_sndSeek` in `portsf.h`. Modify `sf2float` to copy the infile N times (N specified by the user), without a break, to the outfile, i.e., to perform simple looping. Use `psf_sndSeek` to ‘rewind’ the infile back to the beginning each time. (Hint: Study the compiler documentation for the C library function `fseek`. This is used internally by `psf_sndSeek`.)

**Exercise 2.1.7**

This exercise involves a makefile. In the “example” directories on the DVD for each chapter, the `portsf` directory is replicated for each chapter. Move the whole `portsf` library to a separate location on your system, and modify the makefile to put `libportsf.a` in a personal `lib` directory that you create. Do the same for the `include` directory. This is to have just one copy of both on your system. Then modify the supplied `makefiles` to use the new locations, and confirm that all the programs build without errors.

### 2.2 Processing Audio

The key to this and the following sections is line 92 of `sf2float.c`:

```c
/* <---- do any processing here! -----> */
```

The focus of most of the work in this chapter will be on inventing code to replace this comment. The simplest possible replacement is proposed in this section. There may be a temptation to go directly to the “examples” directory and build the program as supplied. You already know how to build programs; the core of the process is to practice writing and modifying code. In the next section no final program listing is given; instead, full details are given for how to modify `sf2float.c`. Dive in.

#### 2.2.1 The First Audio Process: Changing Level

The program shown in listing 2.1.3 was described as a template that can be used as the basis for a number of simple audio utilities working on a soundfile. There are however a few lines that will need to be removed, to make this a truly neutral template program: delete lines 39—