Guide to BOOK Chapter 2 Examples: *Richard Dobson*

Audio Programming in C

Building the Examples

In this chapter, not only are the programs more complex, but also the process of building them is a little more elaborate. In all cases, a number of source files is involved, including the files that form the soundfile library – *portsf*. As we shall see, there are at least three ways to build things. One way involves issuing a series of commands directly at the command-line. A second way involves incorporating these same commands into a *makefile,* and invoking the program *make* from the terminal.

We begin however with the third way, issuing one long command from the command-line. As you begin, in your Terminal session, ensure that your current directory is the Book Chapter 2 *examples* directory as copied from the DVD. This directory contains not only the program source files themselves, but also an "*include*" sub-directory containing header files, most importantly *portsf.h*. To check this in the Terminal, type:

```
$ ls include
```

There is also a sub-directory *portsf* containing the two source files for the library, *portsf.c* and *ieee80.c*. There may also be a file called *Makefile* (ignore this for the moment):

```
$ ls portsf
```

The program *envx* comprises the main source file *envx.c*, and the auxiliary file *breakpoints.c*. It also uses *portsf*. Thus building the program involves a total of four source files, plus two header files in the include directory. This is our single long command line, which is already too long to show in this book without a line-break:

```
$ gcc -o envx -Dunix breakpoints.c envx.c portsf/portsf.c
              portsf/ieee80.c -I include
```

After you press **Enter**, there will be a brief moment of apparent inactivity, until the shell prompt is displayed. You have successfully built *envx*, and you can confirm this in the usual way:

```
$ ./envx
ENVX: extract amplitude envelope from mono soundfile
insufficient arguments.
usage: envx [-wN] insndfile outfile.brk
        -wN:   set extraction window size to N msecs
              (default: 15)
$
```

This `gcc` command has three primary elements:

1. Specify the name for the executable: `-o envx`
2. Give the list of source files: `breakpoints.c envx.c portsf/portsf.c portsf/ieee80.c`
3. Give the path to search for header files: `-I`*`path`*

Two primary `gcc` flag commands are used: `-o` for the output name, and `-I` for the include directory. Being flag options, they can appear in any order on the command line (the only strictly essential arguments to *gcc* are the names of source files). However, the above arrangement is the most usual. Note that paths can be absolute, or relative (to the working directory). Absolute paths are used for system libraries, or third-party libraries installed at system or user level (e.g. `/usr/local/include`). Here we need to use relative paths, in this case to the *portsf* subdirectory.

The `-D` flag is used to supply macro definitions (as created in source files with `#define`) at the compiler level. *Portsf* is written to be cross-platform, and has a few platform-specific elements in the code. To compile on Windows, the symbol WIN32 would need to be defined. For Linux and OS X, we need to define the symbol `unix` (look for this at the top of *portsf.c*). This is the standard mechanism by which to modify source code before compilation, according to the needs of different platforms, or of different configurations such as "debug" or "release" builds of a program.

It is very apparent that this way to compile such a program is at best cumbersome. Even with the benefit of a shell with command history (i.e. you can repeat, and possibly modify, the command by clicking the up-arrow key, which displays the previously used command), it clearly will not scale well to programs that involve even more source files. Also, though we at no point have any need or interest in modifying the *portsf* source files, this command blindly recompiles them every time. In this case the files are relatively small, so that compilation takes but a few seconds, but we would not want to build the whole of *libsndfile* (supposing we actually could) this way!

Portsf is called a library; and our second way to build things involves building it into a true C library file. Again, we will first try the long-winded approach. You will need firstly to enter the portsf directory:

```
$ cd portsf
```

Building *portsf* as a library involves two basic steps:

1. compile each source file into an *object file*:

```
$ gcc -o ieee80.o -c -Dunix -I../include -c ieee80.c
$ gcc -o portsf.o -c -Dunix -I../include -c portsf.c
```

The `-c` flag prevents *gcc* from performing its default action which is to attempt to make an executable program (with a `main()` function). Here we simply want to compile each source file into an object file. An object file (with the `.o` extension) is the raw compiled form of a source file. A program is created by *linking* together a set of these object files. A library is (as its name indicates) little more than an "archive" of one or more object files. To make the library itself (which will be called *libportsf.a*, we use the standard Unix program *ar*:

```
$ ar -r libportsf.a portsf.o ieee80.o
ar: creating archive libportsf.a
$
```

The flag `-r` is the primary command to `ar`, to add files to the archive (creating it if it does not already exist); the library name is followed by each required *object* file. As shown, it prints a simple confirmatory message to the screen. It does not really tell us anything useful, so the `-r` flag is usually combined with the `-c` option, to suppress messages:

```
$ ar -rc libportsf.a portsf.o ieee80.o
```

Finally, we copy *libportsf.a* to our *lib* directory. By convention this is called *installing* the library:

```
$ cp libportsf.a  ../lib
```

Returning up to our program directory, we can now revise the commands to *gcc* to link with the new library:

```
$ cd ..
$ gcc -o envx breakpoints.c envx.c -I include -Llib -lportsf -lm
```

This is our second way to build programs. The new command line now has a rather more comfortable length. It shows two new elements, indicated by upper case "`L`" and lower case "`l`":

`-L`*path* – give the path to a directory to search for libraries.
`-l`*file* – give the name of a library to search for functions.

Internally, these new "link" commands are passed to the *gcc* linker – ***ld***. In a complex program, there may be a long list of link commands. In this case, we include one system library, *libm.a* that supplies the functions defined in `<math.h>`.

Note for OS X users: *libm.a* does not exist on this platform, so this command can be omitted; the maths functions are available through a shared ("dynamic") library mechanism instead. However, it should be included in any Makefile wherever compatibility with Linux or other Unix platforms is required.

Note that while the library is called such as *libportsf.a*, the `-l` command is given an abbreviated name. This is a long-standing Unix convention – both the file extension `.a` and the `lib` part of the name are dropped. If you look in your system directories */usr/lib*, or */usr/local/lib*, you will see a very long list of library files, whose names all begin with "`lib`".

The Third Way – The Makefile

For the programs in Book Chapter 1, all we had to do was type:

```
$ make wonder
```

or even:

```
$ make
```

which builds all the examples very quickly and simply. We can do this for all the examples in this chapter as well. In both cases, a *makefile* is used (called *Makefile*), which is read by the program ***make***. It is now time to look at the makefiles in the *examples* and *portsf* directories.

The makefile encapsulates all the commands shown previously, into a collection of *targets*, *rules, actions* and *dependencies*. It also supports a mechanism for text substitution not very different from the `#define` preprocessor directive:

```
      INCLUDES    = -I./include
      LIBS        = -L./lib -lportsf -lm
      CC = gcc
```

A *target* may be a whole program such as *envx*, a single object file, or some action such as "install". Any argument supplied to *make* invokes the rules associated with that target:

```
      envx: envx.c ./lib/libportsf.a
            $(CC)  -o envx envx.c $(INCLUDES) $(LIBS)
```

The first line defines the target, followed by a list of dependencies – here the source-file *envx.c* and the *portsf* library. The second line defines the rule for this target. A target name must start at the beginning of a line, and be followed immediately by a colon and a space or tab, followed by the list of any dependencies. This is followed by zero or more action lines, each of which must start with a TAB indent (a quirk of *make* – it *must* be a tab, not spaces).

The subtle aspect of this syntax is that a dependency may itself be a target. We see this in the case of *portsf*, which has its own rule:

```
      ./lib/libportsf.a:
            cd portsf; make; make install
```

This rule has no dependencies, only an action, which here concatenates the shell command `cd` and two invocations of *make* itself.

In reading the rule for *envx*, *make* checks the status of any dependencies. If *libportsf.a* does not exist or is not "up to date", it invokes its rule, before resuming the *envx* rule. Since (as far as this book is concerned!) *portsf* is read-only and only needs to be compiled once, we see that this mechanism is extremely efficient, avoiding all unnecessary recompilation. Indeed (looking at the makefile in the examples directory), once all the programs have been compiled, invoking *make* on any of the targets will do nothing, *unless* any source file or other dependency has been changed since the last action. It does this by inspecting the creation date and time for both the target and its dependencies. Where a dependency has a time later than the target, the latter is "out of date" and must be recompiled.

The makefile for *portsf* has a few new elements. Recall that to make a library, we invoke the archive tool ***ar*** with a number of object files. An object file is as much a dependency as any other file. In the majority of cases, the rule for all object files is the same – compile it from the corresponding source file. This is expressed in the very arcane-looking rule:

```
.c.o:
	$(CC) $(CFLAGS) -c $<
```

This is a special notation dealing solely with file extensions – it is a rule to convert *any* `.c` file into a `.o` file. The esoteric notation `$<` is an instance of an "automatic variable"; this one acts as a placeholder for the `.c` file that has been invoked for this rule. Any other rule listing a `.o` file as a dependency will cause *make* to locate a matching `.c` file and apply the specified action. Putting it all together we have:

```
CC=gcc
CFLAGS=-O2 -Wall -Dunix  -I../include
PSFOBJS=	portsf.o ieee80.o

.c.o:
	$(CC) $(CFLAGS) -c $<

libportsf.a:	$(PSFOBJS)
	ar -rc libportsf.a $(PSFOBJS)

install:	libportsf.a
	cp libportsf.a ../lib/libportsf.a
```

When we invoke `make install`, the chain of dependencies is followed all the way back to the first, which compiles the sources into object files, checking at each stage whether any or all dependencies are up to date. Finally, the `install` rule simply copies the library into our *lib* directory.

Windows Users

As noted in Book Chapter 1, it is strongly recommended that readers install the **MinGW** package, which provides a full Unix-style development environment including *gcc*, *make* and the *gdb* debugger. It is possible to build programs from the command-line using Microsoft's VC++ tool *nmake*, but note that it is not possible to debug programs from the command-line, so this is a very limited approach. For those who nevertheless do prefer to use an IDE, **CodeBlocks** is recommended (it installs MinGW internally). In general terms, it offers a very similar environment, to Microsoft's free *Visual Studio Express* IDE. *Portsf* should be created using a "static library" template – use a menu command "Add Files" to copy the source files to the project. Tell the compiler where to look for header files (e.g. "Search Directories"). For the programs themselves, create a "console application" for each, adding the required source files, and (following the principles demonstrated above) tell the compiler, and (in addition to setting the path to the include directory) tell it to link with *portsf* (e.g. "Linker Settings").

The Book Chapter 2 Example Programs

In all the examples here, the input file (hopefully created from *mystery.raw* – see Book Chapter 1 section 1.9) is specified as *mystery.wav* (use *mystery.aiff* if appropriate); but any suitable soundfile will do. Some programs require the creation of a breakpoint file.

sf2float.c: convert a soundfile to 32bit float format.

Usage:

```
     sf2float infile outfile
```

Example:

```
$ ./sf2float mystery.wav fmystery.wav
```

sfgain.c: change the amplitude of the infile.

Usage:

```
     sfgain infile outfile ampfac

            where ampfac > 0.0
```

Example: reduce level by 6dB.

```
$ ./sfgain mystery.wav mystery05.wav 0.5
```

sfnorm.c: normalize soundfile to a given dB level.

Usage:

```
     sfnorm infile outfile dBval

            where dBval <= 0.0
```

Example: normalize *mystery.wav* to nominal 0dB = -12dBFS.

```
$ ./sfnorm mystery.wav mystery12.wav -12
```

sfpan.c: pan mono soundfile over stereo field.

Usage:

```
     sfpan infile outfile posfile.brk

           posfile.brk is breakpoint file with values in range
           -1.0 <= pos <= 1.0 where -1.0 = full Left,
           0 = Centre, +1.0 = full Right
```

Example:

First, in a word processor, create the following breakpoint file and name it – *pan6.brk*:

```
          0.0  -1.0
          2.0   1.0
          4.0  -1.0
          6.0   0.0
```

```
$ ./sfpan am_traffic.aif trafficpan.wav pan6.brk
```

envx.c: extract envelope from a mono soundfile.

Usage:

```
     envx [-wN] insndfile outfile.brk

           -wN: set extraction window size to N msecs
               (default: 15)
```

Example: get low-res envelope of *almostjungle1.wav* written to *jungle.brk.*

```
$ ./envx -w50 almostjungle1.wav jungle.brk
```

sfenv.c: apply breakpoint envelope to mono soundfile.

Usage:

```
     sfenv infile outfile envelope.brk
```

Example: use *jungle.brk* above, on another file.

```
$ ./sfenv am_traffic.aif trafjungle.aiff jungle.brk
```

siggen.c: generate simple waveforms with breakpoint control.

Usage:

```
     siggen outfile wavetype dur srate amp freq
            where wavetype =:

                    0 = sine
                    1 = triangle
                    2 = square
                    3 = sawtooth up
                    4 = sawtooth down

     dur   = duration of outfile (seconds)
     srate = required sample rate of outfile
     amp   = amplitude value or breakpoint file (0 < amp <= 1.0)
     freq  = frequency value (freq > 0) or breakpoint file.
```

Example: generate triangle linear sweep.

First, in a text editor, create the following breakpoint file and name it – *fsweep.brk*:

```
     0.0 100
     5.0 2000

$ ./siggen  trisweep.wav 1 5 44100 0.5 fsweep.brk
```

oscgen.c: additive synthesis of simple waveforms.

Usage:

```
     oscgen[-sN]outfile dur srate nchans amp freq wavetype noscs

     wavetype:  0 = square
                1 = triangle
                2 = saw up
                3 = saw down
```

Example: as above, stereo, 30 partials.

```
$ ./oscgen trioscswp.wav 5 44100 2 0.5 fsweep.brk 1 30
```

tabgen.c: additive synthesis using table lookup.

Usage:

```
tabgen[-sN][-wN][-t]outfile dur srate nchans amp freq type noscs

     -s  = set sample type to N
          (N = 1 to 5, for 8i, 16i, 24i, 32i, 32f)
     -wN = table length (default = 1024)
     -t  = use truncating lookup (else linear interpolation)

         type:  0 = square
                1 = triangle
                2 = saw up
                3 = saw down
```

Example: triangle sweep as above, truncating lookup, output in floats.

```
$ ./tabgen -s5 -t tritruncf.wav 5 44100 1 0.5 fsweep.brk 1 30
```