Edyta Pawlowska, 123RF

Perl script runs a digital recorder backward

YVRUT YSPOT

Michael Schilli whips up a Perl application that plays arbitrary microphone recordings backward. **BY MICHAEL SCHILLI**

A conspiracy theory says that "Revolution 9" by the Beatles played backward contains a message from the band. Just recently, I was listening to "Work It," by Missy Elliot, on web radio and noticed that, despite hearing the song many times since it was released in 2002, I couldn't understand part of the chorus. Although this happens frequently to non-native speakers like myself, this time, Wikipedia [1] solved the puzzle by revealing that Missie had played her vocals backward!

Play It Again ...

What music aficionados now refer to as backmasking [2] was a popular pastime back in the days of my youth. At the time, there were no ego-shooters and, as I started to get bored with politically correct wooden toys, the only thing left was to mix the chemicals that my doting parents had brought back from the drugstore and see how loud an explosion I could create. I also had a cassette recorder that I had managed to talk into playing tapes backward by hacking the drive. I would speak the magic words "Redro Kertesack!" into the mic, and when you played it back, the loudspeakers would say "Kkkassettrekorrdeeer!" in an accent somewhere between Eastern European and Martian: hours of fun for the whole family!

When you run the script in Listing 1 at the command line, it displays the footer line shown in Figure 1 and prompts you to press the *R* (for Record) key to record a message via the microphone [3]. While you are recording, the menu in Figure 2 tells you that pressing *S* (for Stop) will stop the recording and that *P* (for Play) will play pack the resulting Ogg file backward. During playback, the text in Figure 3 appears; it is immediately replaced by the menu in Figure 1 when it runs out of sound data.

Curses Dancing with POE

The minimalist graphical interface comes courtesy of the Curses::UI::POE module, which combines the graphic routines from the Curses library with the POE multitasking environment. Although the script has to handle lengthy tasks such as recording a sound file, re-

Figure 1: Pressing the R key starts the recording.

Figure 2: During recording: S stops recording and P plays it back ... backward.

Figure 3: The flipit script uses the Sox tool to reverse the recording.

versing it, or playing it back, I want the GUI to respond to user input without delay. Experienced followers of this column will recall that POE, the cooperative multitasking environment, uses an asynchronous approach that often causes newcomers difficulty. But once you've gotten the hang of it, you can quickly build robust applications.

The call to the constructor in line 13 sets the *color_support* option to enable color at the console and defines two states that are typical for POE. POE enters the first of these, *_start*, immediately after starting the so-called POE kernel; for all intents and purposes, this is the mother of all states in POE's state machine. Line 15 saves a typical POE data structure, the session heap, in the global *$HEAP* variable to allow access to session data from parts of the script called

Listing 1: flipit

```
001 #!/usr/local/bin/perl -w
002 use strict;
003 use POE;
004 use POE::Wheel::Run;
005 use Curses::UI::POE;
006 use File::Temp qw(tempfile);
007 use Sysadm::Install qw(:all);
008 use POSIX;
009
010 our $HEAP;
011
012 my $CUI =
013   Curses::UI::POE->new(
014    -color_support => 1,
015    inline_states  => {
016     _start => sub {
017       $HEAP = $_[HEAP];
018     },
019    play_ended =>
020      \&footer_update,
021  }
022 );
023
024 my $WIN =
025   $CUI->add("win_id",
026             "Window");
027
028 my $FOOT = $WIN->add(
029  qw( bottom Label
030     -y -1 -paddingspaces 1
031     -fg white -bg blue)
032 );
033
034 footer_update();
035
036 $CUI->set_binding(
037  sub { exit 0; }, "q");
038 $CUI->set_binding(
039  \&play_flipped, "p");
040 $CUI->set_binding(\&record,
041  "r");
042 $CUI->set_binding(
043  \&record_stop, "s");
044
045 $CUI->mainloop;
046
047 ############################
048 sub record {
049 ############################
050  if ( defined
051      $HEAP->{recorder}->{wheel})
052  {
053   return;   # Still recording
054  }
055
056  my ($fh, $tempfile) =
057    tempfile(
058      SUFFIX => ".ogg",
059      UNLINK => 1
060    );
061
062  my $wheel =
063    POE::Wheel::Run->new(
064      Program     => "rec",
065      ProgramArgs => [$tempfile],
066      StderrEvent => 'ignore',
067    );
068
069  $HEAP->{recorder} = {
070   wheel => $wheel,
071   file  => $tempfile,
072  };
073
074  $FOOT->text("Recording ... "
075     . "([s] to stop, "
076     . "[p] to play)");
077  $FOOT->draw();
078 }
079
080 ############################
081 sub record_stop {
082 ############################
083  my $wheel =
084    $HEAP->{recorder}
085    ->{wheel};
086
087  return if !defined $wheel;
088
089  $wheel->kill(SIGTERM);
090  delete $HEAP->{recorder}
091    ->{wheel};
092  footer_update();
093 }
094
095 ############################
096 sub footer_update {
097 ############################
098  my $text = "[r] to record";
099
100  if ( defined
101   $HEAP->{recorder}->{file}){
102   $text .= ", [p] to play";
103  }
104
105  $text .= ", [q] to quit";
106
107  $FOOT->text($text);
108  $FOOT->draw();
109 }
110
111 ############################
112 sub play_flipped {
113 ############################
114  if ( defined
115   $HEAP->{recorder}->{wheel})
116  {
117     # Recording active? Stop.
118   record_stop($HEAP);
119  }
120
121  $FOOT->text("Playing ...");
122  $FOOT->draw();
123
124  my $recorded =
125    $HEAP->{recorder}->{file};
126
127  return
128    if !defined $recorded;
129
130  my $wheel =
131    POE::Wheel::Run->new(
132     Program     => \&sox_play,
133     ProgramArgs => [$recorded],
134     StderrEvent => 'ignore',
135     CloseEvent => 'play_ended',
136    );
137
138  $HEAP->{players}
139    ->{ $wheel->ID } = $wheel;
140 }
141
142 ############################
143 sub sox_play {
144 ############################
145  my ($recording) = @_;
146
147  my ($fh, $tmpfile) =
148    tempfile(
149     SUFFIX => ".ogg");
150
151  tap "sox", $recording,
152    $tmpfile, "reverse";
153  tap "play", $tmpfile;
154
155  unlink $tmpfile;
156 }
```

by event handlers of the graphical interface. POE enters the second state, *play_ended*, after successfully playing a recorded sound backward. For this case, line 20 defines the *footer_update()* handler, which modifies the text showing the recording status in the footer bar.

Widgets on Screen!

The GUI comprises a main window, *$WIN*, and a footer line, *$FOOT*. The main window is brought to life by the Curses::UI::POE module's *add()* method in line 25. The former simply passes things on to Curses::UI or, to be more precise, Curses::UI::Container. The first parameter is the ID for the window (set to "*win_id*" in the script), and the second parameter, "*Window*", defines the class of the widget to create.

The second widget, the footer with the instructions concerning permitted user actions, is created in line 28. In a typically GUI style, the parent widget, *$WIN*, calls the *add()* method to create the footer widget it includes. The *-fg white* and *-bg blue* values define white foreground type against a blue background. The first parameter, the string *bottom*, is the ID of the new window we created; the second, *Label*, is the widget class. The value of *-1* for the *-y* parameter puts the label right down at the bottom of the window. The *-paddingspaces* option extends the label to the horizontal margin of the surrounding main window.

The label has a *text()* method that deletes and sets the text in the footer bar. The *footer_update()* function called in line 34 refreshes the newly defined, and temporarily empty, footer with the details of the user commands permitted after program started.

Ready to Rumble

Lines 36-43 define what happens if the user presses the *Q* (Quit), *P* (Play), *R* (Record), or *S* (Stop Recording) buttons. If *Q* is pressed, flipit calls the *exit()* function, which quits the program. The GUI cleans itself up and neatly collapses.

The handler functions assigned here, *play_flipped()* (play the audio file backwards), *record()* (start recording), and *record_stop()* (stop recording) are defined lower down in the script. To keep things simple, all of these functions access the global *$HEAP* and global widget variables, although this occurs indirectly via the *footer_update()* function.

In typical GUI style, the program then enters the main event loop in line 45. It stays in the loop and keeps processing user input until somebody presses *Q*. If the user presses *R*, the GUI jumps to the *record()* function in line 48. It first checks that a recording is in progress, and if so, it simply cancels by calling *return*, thus ignoring the keypress.

Sox Audio Tool

If no recording is in progress, the *tempfile()* function from the File::Temp module creates a temporary file with an *.ogg* suffix. Perl's automatic wrecker's ball will automatically destroy this when the program terminates, thanks to the *UNLINK* option that was set.

The *.ogg* suffix is important because the *sox* tool that will use it determines the encoding method for the audio file.

POE uses a POE::Wheel::Run object to launch external programs, so the GUI can continue without interruption and even trigger actions if the spawned child program terminates. The wheel in line 62 simply ignores any events that occur as *Stderr* output, (*StderrEvent*). Only the user can stop the recorder.

The program called here, *rec*, is included with the *sox* package (as is the *play* utility I will use later), and I only need to pass it the name of the audio file to create it. The *rec* program will use an internal laptop microphone or an external mic plugged into the sound card for recording. Note that POE::Wheel::Run expects two separate parameters for the program to be called and the parameter list for the program: *Program* and *ProgramArgs*, respectively.

The code as of line 63 will not delay the GUI at all, by the way; any required actions happen in the background. To avoid the wheel losing its last reference when the *record* function exits, thus falling victim to Perl's garbage collector, line 69 stores a reference to it as *recorder* in the POE-specific *$HEAP*. It also stores the name of the temporary file to allow the play function to access it later on. Finally, *record()* updates the footer to tell the user that they can press *S* to stop and *P* to play.

Stop

When the *S* key is pressed, it's the *record_stop()* function's turn; it first checks that a wheel is running. If not, somebody must have pressed *S* without a recording actually being in progress. Otherwise, line 89 shoots down the recording program that's running by sending a *SIGTERM* signal gleaned from the POSIX module. Line 90 then removes the reference to the abruptly terminated wheel from the *$HEAP*.

The *footer_update()* function uses the *text()* method to update the footer line and then calls *draw()* to redraw the widget onscreen.

The *play_flipped* play handler first stops any recordings that are in progress and, in line 131, calls the play wheel. The wheel defines a *CloseEvent*, which enters the *play_ended* POE state defined in line 19 when the *sox_play()* function (lines 143-156) called by the wheel returns. POE doesn't waste time here either, but runs *sox_play()* asynchronously and communicates with its output, error, and end events.

To avoid this wheel collapsing when the script exits the scope of the *play_flipped* function (which happens before the wheel starts the external function because of the asynchronous call), line 138 stores a reference to it in the *$HEAP*. Each wheel has a unique ID within a POE session, and because *play_flipped()* stores the reference under this ID, users could launch multiple playbacks in parallel. If you want to try this out, you can press the *P* three or four times in succession.

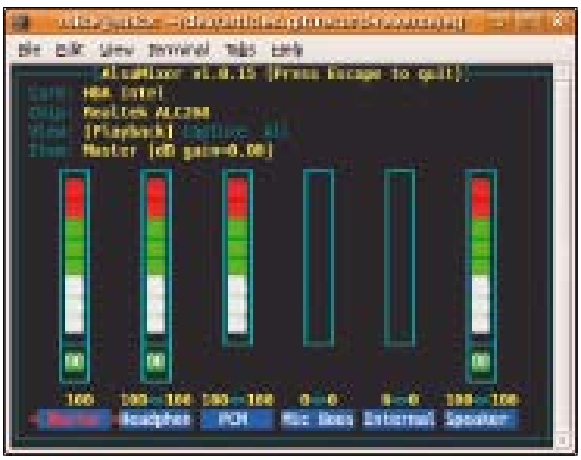

Figure 4: Playback configuration mode after starting Alsamixer.

Flipped

The *sox_play* function creates another temporary, and initially empty, *.ogg* file and passes it to the *sox* utility by calling the CPAN Sysadm::Install module's *tap()* function:

```
sox input.ogg ➥
   output.ogg reverse
```

The *reverse()* option tells *sox* not simply to copy the output into another file but to reverse it while doing so. The sox *play* utility gets the results in line 153 and outputs the file via the sound card. Line 155 then deletes the flipped file because calling *sox_play* again will create a new file.

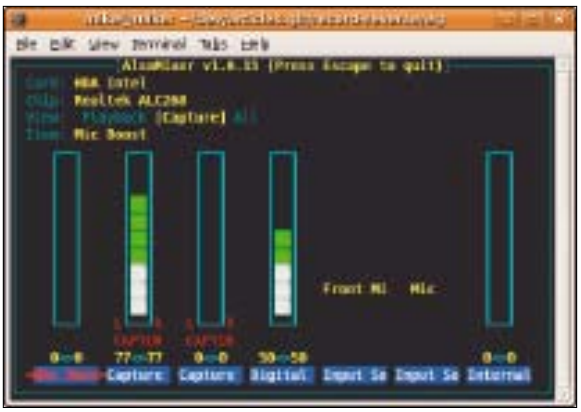

Figure 5: F4 switches to Capture mode. The left "Input So" must read "Front Mic" if you will be using the laptop's built-in microphone. The "Digital" slider sets the record volume.

Activating the Mic

For *rec* to enable the correct microphone, I called the *alsamixer* audio utility for my machine. The start page shows the *Playback* parameters, which relate to data output (Figure 4). F4 switches to Capture mode to adjust microphone settings (Figure 5). For an external, pluggable microphone, set the entry in *Input So* to *Mic*. If your laptop mic is good enough, you can use the arrow keys to set *Front Mic* in Alsamixer. The *Digital* slider sets the sensitivity. Pressing Esc quits Alsamixer and keeps your changes.

Installation

The POE, POE::Wheel::Run, Curses::UI::POE, and Sysadm::Install modules and their dependencies are included with some recent Linux distributions, or you can run a CPAN shell to install them. As a special service, you can watch my training video [4] to learn how to say *Linux Magazine* backward and enjoy flipit's rendering. ■

INFO

[1] "Work It" by Missy Elliott: *http://en.wikipedia.org/wiki/Work_It_(Missy_Elliott_song)*

[2] Backmasking: *http://en.wikipedia.org/wiki/Backmasking*

[3] Listings for this article: *ftp://www.linux-magazin.de/pub/listings/magazin/2010/03/Perl*

[4] Michael Schilli's guide to recording with the *flipit* script: *http://www.youtube.com/watch?v=LdSTla2Tx4o*