SOME NOTES ON PWM ON THE RASPBERRY PI

PUBLISHED TUE, SEP 26, 2017 BY LARS KELLOGG-STEDMAN

I was recently working on a project in which I wanted to drive a simple piezo buzzer attached to a GPIO pin on a Raspberry Pi. I was already using the RPi.GPIO module in my project so that seemed like a logical place to start, but I ran into a few issues.

You drive a piezo buzzer by generating a PWM signal with the appropriate frequency. The `RPi.GPIO` module implements PWM via software, which is tricky on a non-realtime system. It's difficult to get the timing completely accurate, which results in sounds that are a little wobbly at best. Since I'm simply generating tones with a buzzer (rather than, say, controlling a servo) this is mostly just an annoyance.

The second more significant problem is that the `RPi.GPIO` seems to be buggy. After driving the buzzer a few times, my application would invariable crash with a segmentation fault:

```
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x764cbc54 in output_gpio () from /usr/lib/python3/dist-packages/RPi/_GPI
(gdb) bt
#0  0x764dac54 in output_gpio () from /usr/lib/python3/dist-packages/RPi/_GPI
#1  0x764dc9bc in pwm_thread () from /usr/lib/python3/dist-packages/RPi/_GPIO
```

```
#2  0x00001000 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb)
```

At this point, I started looking for alternatives. One option would be to implement my own software PWM solution in my Python code, but that would suffer from the same timing issues as the `RPi.GPIO` implementation. I knew that the Raspberry Pi has support for hardware PWM, so I went looking for information on how to make use of that feature.

I found this article which describes how to enable kernel support for hardware PWM. You can read the article for details, but if you have a Raspberry Pi 3 running kernel 4.9 or later, the answer boils down to:

- Edit `/boot/config.txt`.
- Add the line `dtoverlay=pwm-2chan`
- Save the file.
- Reboot.

After rebooting your Pi and you will have access to hardware PWM on (BCM) pins 18 and 19. You will find a new `sysfs` directory `/sys/class/pwm/pwmchip0`, which operates much like the sysfs support for gpio: there is a special file `export` that you use to gain access to PWN pins. To access pin 18:

```
echo 0 > export
```

To access pin 19:

```
echo 1 > export
```

Running the above will result in two new directories appearing, `/sys/class/pwm/pwmchip0/pwm0` and `/sys/class/pwm/pwmchip0/pwm1`. Each

of these directories contains special files for controlling the PWM output. Of interest in this case are:

- `duty_cycle` - set the duty cycle of the PWM signal.
- `enable` - enable (write a `1`) or disable (write a `0`) PWM output.
- `period` - set the period of the PWM signal.

Both `duty_cycle` and `period` expect values in nanoseconds. So, for example, to emit a 440Hz tone, you first need to calculate the period for that frequency:

```
period = 1 / frequency = 1 / 440 = (approx) .00227272 seconds
```

Then convert that into nanoseconds:

```
period = .00227272 * 1e+9 = 2272720
```

For a 50% duty cycle, just dive that number by 2:

```
duty_cycle = 2272720 / 2 = 1136360
```

Now, echo those values to the appropriate `sysfs` files:

```
echo $period > /sys/class/pwm/pwmchip0/pwm1/period
echo $duty_cycle > /sys/class/pwm/pwmchip0/pwm1/duty_cycle
```

You'll want to set `period` first. The value of `duty_cycle` must always be lower than `period`, so if you try setting `duty_cycle` first it's possible you will get an error.

To actually generate the tone, you need to enable the output:

```
echo 1 > /sys/class/pwm/pwmchip0/pwm1/enable
```

This all works great, but there is one problem: you need to be `root` to perform any of the above operations. This matches the default behavior of the GPIO subsystem, but in that case there are standard udev rules that take care of granting permission to members of the `gpio` group. I was hoping to use the same solution for PWM. There is a set of udev rules proposed at https://github.com/raspberrypi/linux/issues/1983, but due to a kernel issue, no udev events are sent when exporting pins so the rules have no impact on permissions in the `pwm0` and `pwm1` directories.

Until the necessary patch has merged, I've worked around this issue by creating a systemd unit that takes care of exporting the pins and setting permissions correctly. The unit is very simple:

```
[Unit]
Description=Configure PWM permissions
Before=myapp.service

[Service]
Type=oneshot
ExecStart=/usr/bin/rpi-configure-pwm

[Install]
WantedBy=multi-user.target
```

And the corresponding script is:

```
#!/bin/sh

PWM=/sys/class/pwm/pwmchip0

echo 0 > ${PWM}/export
echo 1 > ${PWM}/export

chown -R root:gpio $PWM/*
chmod -R g+rwX $PWM/*
```

The `Before=myapp.service` in the unit file ensures that this unit will run before my application starts up. To use the above, drop the unit file into `/etc/systemd/system/rpi-configure-pwm.service`, and drop the script into

`/usr/bin/rpi-configure-pwm`. Don't forget to `systemctl enable rpi-configure-pwm`.

Tagged: raspberrypi, pwm