Architecture and Design Decisions

Pieter P

This page gives a high-level overview of the architecture of the motor controller code and how the different parts interact.

PWM

Timer2 and Timer0 are used to generate up to four PWM channels at a rate of $31{,}250\,\mathrm{Hz}$. This is a convenient rate, because it is outside of the audible spectrum. Lower frequencies cause the motor to emit annoying noises.

One pin of the H-bridge motor driver is connected to a normal GPIO pin, the other to one of the four PWM outputs of the timers. In order to minimize the number of required IO pins, the motor driver's enable pins are permanently wired to Vcc. The motor is turned off by setting both outputs to the same level.

The direction of the motor is reversed by inverting the GPIO output and inverting the PWM output by changing bit `COM2B0` in the `TCCR2A` register.

ADC

It is important that the control loop runs at a regular interval. To do this, ADC conversions to measure the position of the faders are started in a timer interrupt.

The timer interrupts fire at a high rate, and we have to change the multiplexer channel before starting the next measurement (because we're reading the position of multiple faders), so we just set the “start conversion” bit in the timer interrupt service routine, without using the auto-triggering or free-running mode of the ADC.

Because the interrupt frequency is high ($31{,}250\,\mathrm{Hz}$), we divide this rate further in software, by default, the frequency is reduced by a factor of 30. The ADC measurements are distributed evenly over those 30 interrupts. This is done using a simple counter:

```
constexpr uint8_t num_faders = 3;
constexpr uint8_t interrupt_divisor = 30;
constexpr uint8_t adc_start_count = interrupt_divisor / num_faders;

// Fires at a constant rate of 31,250 Hz:
ISR(TIMER2_OVF_vect) {
    static uint8_t counter = 0;

    for (uint8_t fader = 0; fader < num_faders; ++fader) {
        if (counter == fader * adc_start_count) {
            startADCConversion(fader);
            break;
        }
    }

    ++counter;
    if (counter == interrupt_divisor)
        counter = 0;
}
```

In this example, the Timer2 frequency is divided by 30, so the sampling rate of each of the three ADC channels is $31{,}250\,\mathrm{Hz}/30 \approx 1{,}042\,\mathrm{Hz}$ or $960\mu\mathrm{s}$. The first ADC conversion (measurement) is started when `counter == 0`, the second when `counter == 10`, and the third when `counter == 20`. If the number of faders doesn't divide the interrupt counter divisor evenly, the result of `adc_start_count` is floored. For example, if `num_faders == 4`, the conversions are started when `counter == 0, 7, 14, 21`. This doesn't affect the sampling rates.

The result of the ADC conversion is written to a variable in the ADC conversion ready interrupt.

Control loops

The PID controllers are updated in the main loop. They run whenever a new ADC measurement is available. This means that they are indirectly controlled by the rate of Timer2 as well:

> Timer2 ISR starts ADC conversion → ADC conversion writes measurement to variable → main loop reads ADC measurement → PID controller runs → PWM duty cycle is updated

Capacitive touch sensing

The conductive knob of the fader can be seen as a small capacitive load connected to the Arduino pin. When the knob is touched, the capacitance is higher than when it is not being touched. The capacitance is not measured directly, instead, a large resistor is added between V_{cc} and the knob, and then the RC-time is measured, i.e. the time it takes for the capacitive knob to charge to a voltage of $\left(1 - e^{-1}\right) V_{\mathrm{cc}}$ through the resistor. In practice, we're measuring the time it takes for the voltage to rise to the Arduino's input pin high-voltage, V_{IH}, which is not exactly the RC-time, but the principle is exactly the same.

Let's say that we're using a resistor of $500\,\mathrm{k\Omega}$ and that the capacitance of the untouched fader knob is around $0.1\,\mathrm{nF}$. The RC-time of this circuit is $50\,\mu\mathrm{s}$. We could then define a threshold of, say, $160\,\mu\mathrm{s}$. If the measured RC-time is higher than the threshold, we consider the knob touched. This threshold time corresponds to 5 periods of the Timer2 interrupt. The following figure shows the pin voltage in function of time for two scenarios: the knob being released and the knob being touched. The threshold time and the threshold voltage of $\left(1 - e^{-1}\right) V_{\mathrm{cc}}$ are shown as well.

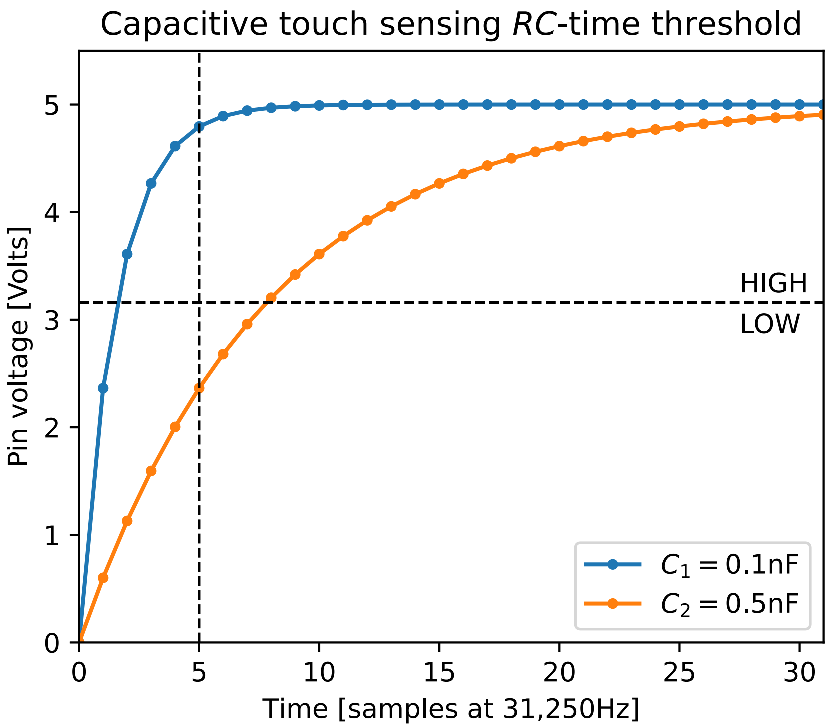

Image source code

To determine whether the knob is being touched, we can just look at the state of the pin after the threshold time: if it's high, the RC-time is less than the threshold time, and the knob is not touched, if it's low, the RC-time is higher than the threshold time, and the knob is touched.

In practice, we just continuously charge and discharge the pin in the Timer2 interrupt. We simply start charging every 30 interrupts, then we count the number of interrupts to the threshold time (5 in this case), and we read the digital state of the pins. Then we switch the pin to output mode to discharge it for a couple of cycles, and then we start charging again.

```cpp
// GPIO pin with the fader knob and pull-up resistor:
constexpr uint8_t touch_pin = 8;
// Frequency at which the Timer2 interrupt fires:
constexpr float interrupt_freq = 31'250;
// Interrupts per control loop period:
constexpr uint8_t interrupt_divisor = 30;
// Minimum RC-time to consider fader knob touched:
constexpr float touch_rc_time_threshold = 160e-6; // seconds
// Same threshold, but as a number of interrupts rather than seconds:
constexpr uint8_t touch_sense_thres = interrupt_freq * touch_rc_time_threshold;
static_assert(touch_sense_thres < interrupt_divisor, "RC-time too long");

volatile bool touched = false; // Whether the knob is touched or not

// Fires at a constant rate of 31,250 Hz:
ISR(TIMER2_OVF_vect) {
    static uint8_t counter = 0;

    if (counter == 0) {
        pinMode(touch_pin, INPUT); // start charging
    } else if (counter == touch_sense_thres) {
        touched = digitalRead(touch_pin) == LOW;
        pinMode(touch_pin, OUTPUT); // start discharging
    }

    ++counter;
    if (counter == interrupt_divisor)
        counter = 0;
}
```

The only reason to wait 30 interrupt cycles before charging again is to synchronize with the ADC and the control loops. This is just for convenience, because in the actual implementation, both touch sensing and starting ADC conversions are handled in the same interrupt service routine. To minimize the overhead, all touch pins are on the same GPIO port, so touch sensing can be done very efficiently using direct port manipulation.

Communication

I^{2}C (Wire)

The motor controller acts as an I^{2}C slave. The master can read the fader positions and whether they are being touched or not, and the master can write the position setpoints.

The response contains the fader positions and touch status as follows (represented in binary):

```
0000 tttt
aaaa aaaa   00aa aaaa
bbbb bbbb   00bb bbbb
cccc cccc   00cc cccc
dddd dddd   00dd dddd
```

`tttt` contains the touch status of up to four faders, the least significant of the four bits is the first fader.
The length of the message depends on the `Config::num_faders` constant. `aaaa aaaa 00aa aaaa` encodes the position of the first fader as a 16-bit Little-Endian integer, `bbbb bbbb 00bb bbbb` for the second fader, and so on. By default, these positions are 14-bit numbers (obtained by oversampling and averaging the 10-bit ADC readings). The number of bits of the position values is `16 - Config::adc_ema_K`, so if the `Config::adc_ema_K` constant changes, the scale of the values changes as well.

To set the reference position, the master sends a message in the following format:

```
rrrr rrrr   00ff 00rr
```

`rrrr rrrr 0000 00rr` is the 10-bit reference position, encoded as a Little-Endian integer, and `ff` is the index of the fader to address (0 to 3).

UART (Serial)

Tuning parameters can be updated at runtime by sending them over the serial port, and it is possible to start experiments, logging the reference, actual position, and control signal. The SLIP protocol (RFC 1055) is used to handle packet framing.

The input format is explained here:

```
// Message format: <command> <fader> <value>
// Commands:
//   - p: proportional gain Kp
//   - i: integral gain Ki
//   - d: derivative gain Kd
//   - c: derivative filter cutoff frequency f_c (Hz)
//   - m: maximum absolute control output
//   - s: start an experiment, using getNextExperimentSetpoint
// Fader index: up to four faders are addressed using the characters '0' - '3'.
// Values: values are sent as 32-bit little Endian floating point numbers.
//
// For example the message 'c0\x00\x00\x20\x42' sets the derivative filter
// cutoff frequency of the first fader to 40.
```

The outgoing messages are just SLIP packets containing the reference, the measured position and the control signal as three signed 16-bit Little-Endian integers.