ATmega328P Code

Pieter P

`main.cpp`

```
// Configuration and initialization of the analog-to-digital converter:
#include "ADC.hpp"
// Capacitive touch sensing:
#include "Touch.hpp"
// PID controller:
#include "Controller.hpp"
// Configuration of PWM and Timer2/0 for driving the motor:
#include "Motor.hpp"
// Reference signal for testing the performance of the controller:
#include "Reference.hpp"
// Helpers for low-level AVR Timer2/0 and ADC registers:
#include "Registers.hpp"
// Parsing incoming messages over Serial using SLIP packets:
#include "SerialSLIP.hpp"

#include <Arduino.h>         // setup, loop, analogRead
#include <Arduino_Helpers.h> // EMA.hpp
#include <Wire.h>            // I²C slave

#include "SMA.hpp"            // SMA filter
#include <AH/Filters/EMA.hpp> // EMA filter

// ------------------------------ Description ------------------------------- //

// This sketch drives up to four motorized faders using a PID controller. The
// motor is disabled when the user touches the knob of the fader.
//
// Everything is driven by Timer2, which runs (by default) at a rate of
// 31.250 kHz. This high rate is used to eliminate audible tones from the PWM
// drive for the motor. Timer0 is used for the PWM outputs of faders 3 and 4.
// Every 30 periods of Timer2 (960 µs), each analog input is sampled, and
// this causes the PID control loop to run in the main loop function.
// Capacitive sensing is implemented by measuring the RC time on the touch pin
// in the Timer2 interrupt handler. The “touched” status is sticky for >20 ms
// to prevent interference from the 50 Hz mains.
//
// There are options to (1) follow a test reference (with ramps and jumps), (2)
// to receive a target position over I²C, or (3) to run experiments based on
// commands received over the serial port. The latter is used by a Python script
// that performs experiments with different tuning parameters for the
// controllers.

// -------------------------------- Hardware -------------------------------- //

// Fader 0:
//  - A0:  wiper of the potentiometer          (ADC0)
//  - D8:  touch pin of the knob               (PB0)
//  - D2:  input 1A of L293D dual H-bridge 1   (PD2)
//  - D3:  input 2A of L293D dual H-bridge 1   (OC2B)
//
// Fader 1:
//  - A1:  wiper of the potentiometer          (ADC1)
//  - D9:  touch pin of the knob               (PB1)
//  - D13: input 3A of L293D dual H-bridge 1   (PB5)
//  - D11: input 4A of L293D dual H-bridge 1   (OC2A)
//
// Fader 2:
//  - A2:  wiper of the potentiometer          (ADC2)
//  - D10: touch pin of the knob               (PB2)
//  - D4:  input 1A of L293D dual H-bridge 2   (PD4)
//  - D5:  input 2A of L293D dual H-bridge 2   (OC0B)
//
// Fader 3:
//  - A3:  wiper of the potentiometer          (ADC3)
//  - D12: touch pin of the knob               (PB4)
//  - D7:  input 3A of L293D dual H-bridge 2   (PD7)
//  - D6:  input 4A of L293D dual H-bridge 2   (OC0A)
//
// If fader 1 is unused:
//  - D13: LED or scope as overrun indicator   (PB5)
//
// For communication:
//  - D0:  UART TX                             (TXD)
//  - D1:  UART RX                             (RXD)
//  - A4:  I²C data                            (SDA)
//  - A5:  I²C clock                           (SCL)
//
// Connect the outer connections of the potentiometers to ground and Vcc, it's
// recommended to add a 100 nF capacitor between each wiper and ground.
// Connect the 1,2EN and 3,4EN enable pins of the L293D chips to Vcc.
// Connect a 500kΩ pull-up resistor between each touch pin and Vcc.
// On an Arduino Nano, you can set an option to use pins A6/A7 instead of A2/A3.
// Note that D13 is often pulsed by the bootloader, which might cause the fader
// to move when resetting the Arduino. You can either disable this behavior in
// the bootloader, or use a different pin (e.g. A3 or A4 on an Arduino Nano).
// The overrun indicator is only enabled if the number of faders is 1, because
// it conflicts with the motor driver pin of Fader 1. You can choose a different
// pin instead.

// ----------------------------- Configuration ------------------------------ //

struct Config {
    // Print the control loop and interrupt frequencies to Serial at startup:
    static constexpr bool print_frequencies = true;
```

```
    // Print the setpoint, actual position and control signal to Serial.
    // Note that this slows down the control loop significantly, it probably
    // won't work if you are using more than one fader without increasing
    // `interrupt_divisor`:
    static constexpr bool print_controller_signals = false;
    static constexpr uint8_t controller_to_print = 0;
    // Follow the test reference trajectory (true) or receive the target
    // position over I²C or Serial (false):
    static constexpr bool test_reference = false;
    // Increase this divisor to slow down the test reference:
    static constexpr uint8_t test_reference_speed_div = 4;
    // Allow control for tuning and starting experiments over Serial:
    static constexpr bool serial_control = true;
    // I²C slave address (zero to disable I²C):
    static constexpr uint8_t i2c_address = 8;

    // Number of faders, must be between 1 and 4:
    static constexpr size_t num_faders = 1;
    // Actually drive the motors:
    static constexpr bool enable_controller = true;
    // Use analog pins (A0, A1, A6, A7) instead of (A0, A1, A2, A3), useful for
    // saving digital pins on an Arduino Nano:
    static constexpr bool use_A6_A7 = true;
    // Use pin A2 instead of D13 as the motor driver pin for the second fader.
    // Can only be used if `use_A6_A7` is set to true:
    static constexpr bool fader_1_A2 = true;

    // Capacitive touch sensing RC time threshold.
    // Increase this time constant if the capacitive touch sense is too
    // sensitive or decrease it if it's not sensitive enough:
    static constexpr float touch_rc_time_threshold = 150e-6; // seconds
    // Bit masks of the touch pins (must be on port B):
    static constexpr uint8_t touch_masks[] = {1 << PB0, 1 << PB1, 1 << PB2,
                                              1 << PB4};

    // Use phase-correct PWM (true) or fast PWM (false), this determines the
    // timer interrupt frequency, prefer phase-correct PWM with prescaler 1 on
    // 16 MHz boards, and fast PWM with prescaler 1 on 8 MHz boards, both result
    // in a PWM and interrupt frequency of 31.250 kHz
    // (fast PWM is twice as fast):
    static constexpr bool phase_correct_pwm = true;
    // The fader position will be sampled once per `interrupt_divisor` timer
    // interrupts, this determines the sampling frequency of the control loop.
    // Some examples include 20 → 320 μs, 30 → 480 μs, 60 → 960 μs,
    // 90 → 1,440 μs, 124 → 2,016 μs, 188 → 3,008 μs, 250 → 4,000 μs.
    // 60 is the default, because it works with four faders. If you only use
    // a single fader, you can go as low as 20 because you only need a quarter
    // of the computations and ADC time:
    static constexpr uint8_t interrupt_divisor = 60 / (1 + phase_correct_pwm);
    // The prescaler for the timer, affects PWM and control loop frequencies:
    static constexpr unsigned prescaler_fac = 1;
    // The prescaler for the ADC, affects speed of analog readings:
    static constexpr uint8_t adc_prescaler_fac = 64;

    // Turn off the motor after this many seconds of inactivity:
    static constexpr float timeout = 2;

    // EMA filter factor for fader position filters:
    static constexpr uint8_t adc_ema_K = 2;
    // SMA filter length for setpoint filters, improves tracking of ramps if the
    // setpoint changes in steps (e.g. when the DAW only updates the reference
    // every 20 ms). Powers of two are significantly faster (e.g. 32 works well):
    static constexpr uint8_t setpoint_sma_length = 0;

    // ------------------------ Computed Quantities ------------------------- //

    // Sampling time of control loop:
    constexpr static float Ts = 1. * prescaler_fac * interrupt_divisor * 256 *
                                (1 + phase_correct_pwm) / F_CPU;
    // Frequency at which the interrupt fires:
    constexpr static float interrupt_freq =
        1. * F_CPU / prescaler_fac / 256 / (1 + phase_correct_pwm);
    // Clock speed of the ADC:
    constexpr static float adc_clock_freq = 1. * F_CPU / adc_prescaler_fac;
    // Pulse pin D13 if the control loop took too long:
    constexpr static bool enable_overrun_indicator =
        num_faders < 2 || fader_1_A2;

    static_assert(use_A6_A7 || !fader_1_A2,
                  "Cannot use A2 for motor driver "
                  "and analog input at the same time");
};
constexpr uint8_t Config::touch_masks[];
constexpr float Ts = Config::Ts;

// ----------------- ADC, Capacitive Touch State and Motors ----------------- //

ADCManager<Config> adc;
TouchSense<Config> touch;
Motors<Config> motors;

// ----------------------- Setpoints and References ----------------------- //

// Setpoints (target positions) for all faders:
Reference<Config> references[Config::num_faders];
```

```

// ----------------------------- Controllers ------------------------------ //

// The main PID controllers. Need tuning for your specific setup:

PID controllers[] {
    // This is an example of a controller with very little overshoot
    {
        6,     // Kp: proportional gain
        2,     // Ki: integral gain
        0.035, // Kd: derivative gain
        Ts,    // Ts: sampling time
        60, // fc: cutoff frequency of derivative filter (Hz), zero to disable
    },
    // This one has more overshoot, but less ramp tracking error
    {
        4,     // Kp: proportional gain
        11,    // Ki: integral gain
        0.028, // Kd: derivative gain
        Ts,    // Ts: sampling time
        40, // fc: cutoff frequency of derivative filter (Hz), zero to disable
    },
    // This is a very aggressive controller
    {
        8.55,  // Kp: proportional gain
        440,   // Ki: integral gain
        0.043, // Kd: derivative gain
        Ts,    // Ts: sampling time
        70, // fc: cutoff frequency of derivative filter (Hz), zero to disable
    },
    // Fourth controller
    {
        6,     // Kp: proportional gain
        2,     // Ki: integral gain
        0.035, // Kd: derivative gain
        Ts,    // Ts: sampling time
        60, // fc: cutoff frequency of derivative filter (Hz), zero to disable
    },
};

template <uint8_t Idx>
void printControllerSignals(int16_t setpoint, int16_t adcval, int16_t control) {
    // Send (binary) controller signals over Serial to plot in Python
    if (Config::serial_control && references[Idx].experimentInProgress()) {
        const int16_t data[3] {setpoint, adcval, control};
        SLIPSender(Serial).writePacket(reinterpret_cast<const uint8_t *>(data),
                                       sizeof(data));
    }
    // Print signals as text
    else if (Config::print_controller_signals &&
             Idx == Config::controller_to_print) {
        Serial.print(setpoint);
        Serial.print('\t');
        Serial.print(adcval);
        Serial.print('\t');
        Serial.print((control + 256) * 2);
        Serial.println();
    }
}

template <uint8_t Idx>
void updateController(uint16_t setpoint, int16_t adcval, bool touched) {
    auto &controller = controllers[Idx];

    // Prevent the motor from being turned off after being touched
    if (touched) controller.resetActivityCounter();

    // Set the target position
    if (Config::setpoint_sma_length > 0) {
        static SMA<Config::setpoint_sma_length, uint16_t, uint32_t> sma;
        uint16_t filtsetpoint = sma(setpoint);
        controller.setSetpoint(filtsetpoint);
    } else {
        controller.setSetpoint(setpoint);
    }

    // Update the PID controller to get the control action
    int16_t control = controller.update(adcval);

    // Apply the control action to the motor
    if (Config::enable_controller) {
        if (touched) // Turn off motor if knob is touched
            motors.setSpeed<Idx>(0);
        else
            motors.setSpeed<Idx>(control);
    }

    printControllerSignals<Idx>(controller.getSetpoint(), adcval, control);
}

template <uint8_t Idx>
void readAndUpdateController() {
    // Read the ADC value for the given fader:
    int16_t adcval = adc.read(Idx);
    // If the ADC value was updated by the ADC interrupt, run the control loop:
```

```
    if (adcval >= 0) {
        // Check if the fader knob is touched
        bool touched = touch.read(Idx);
        // Read the target position
        uint16_t setpoint = references[Idx].getNextSetpoint();
        // Run the control loop
        updateController<Idx>(setpoint, adcval, touched);
        // Write -1 so the controller doesn't run again until the next value is
        // available:
        adc.write(Idx, -1);
        if (Config::enable_overrun_indicator)
            cbi(PORTB, 5); // Clear overrun indicator
    }
}

// ------------------------------ Setup & Loop ------------------------------ //

void onRequest();
void onReceive(int);
void updateSerialIn();

void setup() {
    // Initialize some globals
    for (uint8_t i = 0; i < Config::num_faders; ++i) {
        // all fader positions for the control loop start of as -1 (no reading)
        adc.write(i, -1);
        // reset the filter to the current fader position to prevent transients
        adc.writeFiltered(i, analogRead(adc.channel_index_to_mux_address(i)));
        // after how many seconds of not touching the fader and not changing
        // the reference do we turn off the motor?
        controllers[i].setActivityTimeout(Config::timeout);
    }

    // Configure the hardware
    if (Config::print_frequencies || Config::print_controller_signals ||
        Config::serial_control)
        Serial.begin(1000000);

    if (Config::enable_overrun_indicator) sbi(DDRB, 5); // Pin 13 output

    adc.begin();
    touch.begin();
    motors.begin();

    if (Config::print_frequencies) {
        Serial.println();
        Serial.print(F("Interrupt frequency (Hz): "));
        Serial.println(Config::interrupt_freq);
        Serial.print(F("Controller sampling time (μs): "));
        Serial.println(Config::Ts * 1e6);
        Serial.print(F("ADC clock rate (Hz): "));
        Serial.println(Config::adc_clock_freq);
        Serial.print(F("ADC sampling rate (Sps): "));
        Serial.println(adc.adc_rate);
    }
    if (Config::print_controller_signals) {
        Serial.println();
        Serial.println(F("Reference\tActual\tControl\t-"));
        Serial.println(F("0\t0\t0\t0\r\n0\t0\t0\t1024"));
    }

    // Initalize I²C slave and attach callbacks
    if (Config::i2c_address) {
        Wire.begin(Config::i2c_address);
        Wire.onRequest(onRequest);
        Wire.onReceive(onReceive);
    }

    // Enable Timer2 overflow interrupt, this starts reading the touch sensitive
    // knobs and sampling the ADC, which causes the controllers to run in the
    // main loop
    sbi(TIMSK2, TOIE2);
}

void loop() {
    if (Config::num_faders > 0) readAndUpdateController<0>();
    if (Config::num_faders > 1) readAndUpdateController<1>();
    if (Config::num_faders > 2) readAndUpdateController<2>();
    if (Config::num_faders > 3) readAndUpdateController<3>();
    if (Config::serial_control) updateSerialIn();
}

// ------------------------------- Interrupts ------------------------------- //

// Fires at a constant rate of `interrupt_freq`.
ISR(TIMER2_OVF_vect) {
    // We don't have to take all actions at each interupt, so keep a counter to
    // know when to take what actions.
    static uint8_t counter = 0;

    adc.update(counter);
    touch.update(counter);

    ++counter;
    if (counter == Config::interrupt_divisor) counter = 0;
```

```
}

// Fires when the ADC measurement is complete. Stores the reading, both before
// and after filtering (for the controller and for user input respectively).
ISR(ADC_vect) { adc.complete(); }

// --------------------------------- Wire --------------------------------- //

// Send the touch status and filtered fader positions to the master.
void onRequest() {
    uint8_t touched = 0;
    for (uint8_t i = 0; i < Config::num_faders; ++i)
        touched |= touch.touched[i] << i;
    Wire.write(touched);
    for (uint8_t i = 0; i < Config::num_faders; ++i) {
        uint16_t filt_read = adc.filtered_readings[i];
        Wire.write(reinterpret_cast<const uint8_t *>(&filt_read), 2);
    }
}

// Change the setpoint of the given fader based on the value in the message
// received from the master.
void onReceive(int count) {
    if (count < 2) return;
    if (Wire.available() < 2) return;
    uint16_t data = Wire.read();
    data |= uint16_t(Wire.read()) << 8;
    uint8_t idx = data >> 12;
    data &= 0x03FF;
    if (idx < Config::num_faders) references[idx].setMasterSetpoint(data);
}

// --------------------------------- Serial ------------------------------- //

// Read SLIP messages from the serial port that allow dynamically updating the
// tuning of the controllers. This is used by the Python tuning script.
//
// 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.

void updateSerialIn() {
    static SLIPParser parser;
    static char cmd = '\0';
    static uint8_t fader_idx = 0;
    static uint8_t buf[4];
    static_assert(sizeof(buf) == sizeof(float), "");
    // This function is called if a new byte of the message arrives:
    auto on_char_receive = [&](char new_byte, size_t index_in_packet) {
        if (index_in_packet == 0) {
            cmd = new_byte;
        } else if (index_in_packet == 1) {
            fader_idx = new_byte - '0';
        } else if (index_in_packet < 6) {
            buf[index_in_packet - 2] = new_byte;
        }
    };
    // Convert the 4-byte buffer to a float:
    auto as_f32 = [&] {
        float f;
        memcpy(&f, buf, sizeof(float));
        return f;
    };
    // Read and parse incoming packets from Serial:
    while (Serial.available() > 0) {
        uint8_t c = Serial.read();
        auto msg_size = parser.parse(c, on_char_receive);
        // If a complete message of 6 bytes was received, and if it addresses
        // a valid fader:
        if (msg_size == 6 && fader_idx < Config::num_faders) {
            // Execute the command:
            switch (cmd) {
                case 'p': controllers[fader_idx].setKp(as_f32()); break;
                case 'i': controllers[fader_idx].setKi(as_f32()); break;
                case 'd': controllers[fader_idx].setKd(as_f32()); break;
                case 'c': controllers[fader_idx].setEMACutoff(as_f32()); break;
                case 'm': controllers[fader_idx].setMaxOutput(as_f32()); break;
                case 's':
                    references[fader_idx].startExperiment(as_f32());
                    controllers[fader_idx].resetIntegral();
                    break;
                default: break;
            }
        }
```

```
    }
}
```

ADC.hpp

```cpp
#pragma once
#include "Registers.hpp"
#include <avr/interrupt.h>
#include <util/atomic.h>

#include <Arduino_Helpers.h> // EMA.hpp

#include <AH/Filters/EMA.hpp> // EMA filter

template <class Config>
struct ADCManager {
    /// Evenly distribute the analog readings in the control loop period.
    constexpr static uint8_t adc_start_count =
        Config::interrupt_divisor / Config::num_faders;
    /// The rate at which we're sampling using the ADC.
    constexpr static float adc_rate = Config::interrupt_freq / adc_start_count;
    // Check that this doesn't take more time than the 13 ADC clock cycles it
    // takes to actually do the conversion. Use 14 instead of 13 just to be safe.
    static_assert(adc_rate <= Config::adc_clock_freq / 14, "ADC too slow");

    /// Enable the ADC with Vcc reference, with the given prescaler, auto
    /// trigger disabled, ADC interrupt enabled.
    /// Called from main program, with interrupts enabled.
    void begin();

    /// Start an ADC conversion on the given channel.
    /// Called inside an ISR.
    void startConversion(uint8_t channel);

    /// Start an ADC conversion at the right intervals.
    /// @param  counter
    ///         Counter that keeps track of how many times the timer interrupt
    ///         fired, between 0 and Config::interrupt_divisor - 1.
    /// Called inside an ISR.
    void update(uint8_t counter);

    /// Read the latest ADC result.
    /// Called inside an ISR.
    void complete();

    /// Get the latest ADC reading for the given index.
    /// Called from main program, with interrupts enabled.
    int16_t read(uint8_t idx);
    /// Get the latest filtered ADC reading for the given index.
    /// Called from main program, with interrupts enabled.
    int16_t readFiltered(uint8_t idx);

    /// Write the ADC reading for the given index.
    /// Called from main program, with interrupts enabled.
    void write(uint8_t idx, int16_t val);
    /// Write the filtered ADC reading for the given index.
    /// Called only before ADC interrupts are enabled.
    void writeFiltered(uint8_t idx, int16_t val);

    /// Convert the channel index between 0 and Config::num_faders - 1 to the
    /// actual ADC multiplexer address.
    constexpr static inline uint8_t
    channel_index_to_mux_address(uint8_t adc_mux_idx) {
        return Config::use_A6_A7
                   ? (adc_mux_idx < 2 ? adc_mux_idx : adc_mux_idx + 4)
                   : adc_mux_idx;
    }

    /// Index of the ADC channel currently being read.
    uint8_t channel_index = Config::num_faders;
    /// Latest ADC reading of each fader (updated in ADC ISR). Used for the
    /// control loop.
    volatile int16_t readings[Config::num_faders];
    /// Filters for ADC readings.
    EMA<Config::adc_ema_K, uint16_t> filters[Config::num_faders];
    /// Filtered ADC readings. Used to output over MIDI.
    volatile uint16_t filtered_readings[Config::num_faders];
};

template <class Config>
inline void ADCManager<Config>::begin() {
    constexpr auto prescaler = factorToADCPrescaler(Config::adc_prescaler_fac);
    static_assert(prescaler != ADCPrescaler::Invalid, "Invalid prescaler");

    ATOMIC_BLOCK(ATOMIC_FORCEON) {
        cbi(ADCSRA, ADEN); // Disable ADC

        cbi(ADMUX, REFS1); // Vcc reference
        sbi(ADMUX, REFS0); // Vcc reference

        cbi(ADMUX, ADLAR); // 8 least significant bits in ADCL

        setADCPrescaler(prescaler);

        cbi(ADCSRA, ADATE); // Auto trigger disable
        sbi(ADCSRA, ADIE);  // ADC Interrupt Enable
        sbi(ADCSRA, ADEN);  // Enable ADC
    }
}
```

```cpp

template <class Config>
inline void ADCManager<Config>::update(uint8_t counter) {
    if (Config::num_faders > 0 && counter == 0 * adc_start_count)
        startConversion(0);
    else if (Config::num_faders > 1 && counter == 1 * adc_start_count)
        startConversion(1);
    else if (Config::num_faders > 2 && counter == 2 * adc_start_count)
        startConversion(2);
    else if (Config::num_faders > 3 && counter == 3 * adc_start_count)
        startConversion(3);
}

template <class Config>
inline void ADCManager<Config>::startConversion(uint8_t channel) {
    channel_index = channel;
    ADMUX &= 0xF0;
    ADMUX |= channel_index_to_mux_address(channel);
    sbi(ADCSRA, ADSC); // ADC Start Conversion
}

template <class Config>
inline void ADCManager<Config>::complete() {
    if (Config::enable_overrun_indicator && readings[channel_index] >= 0)
        sbi(PORTB, 5);    // Set overrun indicator
    uint16_t value = ADC; // Store ADC reading
    readings[channel_index] = value;
    // Filter the reading
    auto &filter = filters[channel_index];
    filtered_readings[channel_index] = filter(value << (6 - Config::adc_ema_K));
}

template <class Config>
inline int16_t ADCManager<Config>::read(uint8_t idx) {
    int16_t v;
    ATOMIC_BLOCK(ATOMIC_FORCEON) { v = readings[idx]; }
    return v;
}

template <class Config>
inline void ADCManager<Config>::write(uint8_t idx, int16_t val) {
    ATOMIC_BLOCK(ATOMIC_FORCEON) { readings[idx] = val; }
}

template <class Config>
inline int16_t ADCManager<Config>::readFiltered(uint8_t idx) {
    int16_t v;
    ATOMIC_BLOCK(ATOMIC_FORCEON) { v = filtered_readings[idx]; }
    return v;
}

template <class Config>
inline void ADCManager<Config>::writeFiltered(uint8_t idx, int16_t val) {
    filters[idx].reset(val);
    filtered_readings[idx] = val;
}
```

Touch.hpp

```cpp
#pragma once
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>

template <class Config>
struct TouchSense {

    /// The actual threshold as a number of interrupts instead of seconds:
    static constexpr uint8_t touch_sense_thres =
        Config::interrupt_freq * Config::touch_rc_time_threshold;
    /// Ignore mains noise by making the “touched” status stick for longer than
    /// the mains period:
    static constexpr float period_50Hz = 1. / 50;
    /// Keep the “touched” status active for this many periods (see below):
    static constexpr uint8_t touch_sense_stickiness =
        Config::interrupt_freq * period_50Hz * 4 / Config::interrupt_divisor;
    /// Check that the threshold is smaller than the control loop period:
    static_assert(touch_sense_thres < Config::interrupt_divisor,
                  "Touch sense threshold too high");

    /// The combined bit mask for all touch GPIO pins.
    static constexpr uint8_t gpio_mask =
        (Config::num_faders > 0 ? Config::touch_masks[0] : 0) |
        (Config::num_faders > 1 ? Config::touch_masks[1] : 0) |
        (Config::num_faders > 2 ? Config::touch_masks[2] : 0) |
        (Config::num_faders > 3 ? Config::touch_masks[3] : 0);

    /// Initialize the GPIO pins for capacitive sensing.
    /// Called from main program, with interrupts enabled.
    void begin();

    /// Check which touch sensing knobs are being touched.
    /// @param  counter
    ///         Counter that keeps track of how many times the timer interrupt
    ///         fired, between 0 and Config::interrupt_divisor - 1.
    /// Called inside an ISR.
    void update(uint8_t counter);

    /// Get the touch status for the given index.
    /// Called from main program, with interrupts enabled.
    bool read(uint8_t idx);

    // Timers to take into account the stickiness.
    uint8_t touch_timers[Config::num_faders] {};
    // Whether the knobs are being touched.
    volatile bool touched[Config::num_faders];
};

template <class Config>
void TouchSense<Config>::begin() {
    ATOMIC_BLOCK(ATOMIC_FORCEON) {
        PORTB &= ~gpio_mask; // low
        DDRB |= gpio_mask;   // output mode
    }
}

// 0. The pin mode is “output”, the value is “low”.
// 1. Set the pin mode to “input”, touch_timer = 0.
// 2. The pin will start charging through the external pull-up resistor.
// 3. After a fixed amount of time, check whether the pin became “high”:
//    if this is the case, the RC-time of the knob/pull-up resistor circuit
//    was smaller than the given threshold. Since R is fixed, this can be used
//    to infer C, the capacitance of the knob: if the capacitance is lower than
//    the threshold (i.e. RC-time is lower), this means the knob was not touched.
// 5. Set the pin mode to “output”, to start discharging the pin to 0V again.
// 6. Some time later, the pin has discharged, so switch to “input” mode and
//    start charging again for the next RC-time measurement.
//
// The “touched” status is sticky: it will remain set for at least
// touch_sense_stickiness ticks. If the pin never resulted in another “touched”
// measurement during that period, the “touched” status for that pin is cleared.

template <class Config>
void TouchSense<Config>::update(uint8_t counter) {
    if (counter == 0) {
        DDRB &= ~gpio_mask; // input mode, start charging
    } else if (counter == touch_sense_thres) {
        uint8_t touched_bits = PINB;
        DDRB |= gpio_mask; // output mode, start discharging
        for (uint8_t i = 0; i < Config::num_faders; ++i) {
            bool touch_i = (touched_bits & Config::touch_masks[i]) == 0;
            if (touch_i) {
                touch_timers[i] = touch_sense_stickiness;
                touched[i] = true;
            } else if (touch_timers[i] > 0) {
                --touch_timers[i];
                if (touch_timers[i] == 0) touched[i] = false;
            }
        }
    }
}

template <class Config>
```

```
bool TouchSense<Config>::read(uint8_t idx) {
    bool t;
    ATOMIC_BLOCK(ATOMIC_FORCEON) { t = touched[idx]; }
    return t;
}
```

`Controller.hpp`

```
#pragma once

#include <stddef.h>
#include <stdint.h>

/// @see @ref horner(float,float,const float(&)[N])
constexpr inline float horner_impl(float xa, const float *p, size_t count,
                                   float t) {
    return count == 0 ? p[count] + xa * t
                      : horner_impl(xa, p, count - 1, p[count] + xa * t);
}

/// Evaluate a polynomial using
/// [Horner's method](https://en.wikipedia.org/wiki/Horner%27s_method).
template <size_t N>
constexpr inline float horner(float x, float a, const float (&p)[N]) {
    return horner_impl(x - a, p, N - 2, p[N - 1]);
}

/// Compute the weight factor of a exponential moving average filter
/// with the given cutoff frequency.
/// @see https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Exponential%20Moving%20Average/Exponential-Moving-Average.html#cutoff-frequency
///      for the formula.
inline float calcAlphaEMA(float f_n) {
    // Taylor coefficients of
    // α(fₙ) = cos(2πfₙ) - 1 + √( cos(2πfₙ)² - 4 cos(2πfₙ) + 3 )
    // at fₙ = 0.25
    constexpr static float coeff[] {
        +7.3205080756887730e-01, +9.7201214975728490e-01,
        -3.7988125051760377e+00, +9.5168450173968860e+00,
        -2.0829320344443730e+01, +3.0074306603814595e+01,
        -1.6446172139457754e+01, -8.0756002564633450e+01,
        +3.2420501524111750e+02, -6.5601870948443250e+02,
    };
    return horner(f_n, 0.25, coeff);
}

/// Standard PID (proportional, integral, derivative) controller. Derivative
/// component is filtered using an exponential moving average filter.
class PID {
  public:
    PID() = default;
    /// @param  kp
    ///         Proportional gain
    /// @param  ki
    ///         Integral gain
    /// @param  kd
    ///         Derivative gain
    /// @param  Ts
    ///         Sampling time (seconds)
    /// @param  fc
    ///         Cutoff frequency of derivative EMA filter (Hertz),
    ///         zero to disable the filter entirely
    PID(float kp, float ki, float kd, float Ts, float f_c = 0,
        float maxOutput = 255)
        : Ts(Ts), maxOutput(maxOutput) {
        setKp(kp);
        setKi(ki);
        setKd(kd);
        setEMACutoff(f_c);
    }

    /// Update the controller: given the current position, compute the control
    /// action.
    float update(uint16_t input) {
        // The error is the difference between the reference (setpoint) and the
        // actual position (input)
        int16_t error = setpoint - input;
        // The integral or sum of current and previous errors
        int32_t newIntegral = integral + error;
        // Compute the difference between the current and the previous input,
        // but compute a weighted average using a factor α ∈ (0,1]
        float diff = emaAlpha * (prevInput - input);
        // Update the average
        prevInput -= diff;

        // Check if we can turn off the motor
        if (activityCount >= activityThres && activityThres) {
            float filtError = setpoint - prevInput;
            if (filtError >= -errThres && filtError <= errThres) {
                errThres = 2; // hysteresis
                integral = newIntegral;
                return 0;
            } else {
                errThres = 1;
            }
        } else {
            ++activityCount;
            errThres = 1;
        }

        bool backward = false;
        int32_t calcIntegral = backward ? newIntegral : integral;
```

```cpp

        // Standard PID rule
        float output = kp * error + ki_Ts * calcIntegral + kd_Ts * diff;

        // Clamp and anti-windup
        if (output > maxOutput)
            output = maxOutput;
        else if (output < -maxOutput)
            output = -maxOutput;
        else
            integral = newIntegral;

        return output;
    }

    void setKp(float kp) { this->kp = kp; }               ///< Proportional gain
    void setKi(float ki) { this->ki_Ts = ki * this->Ts; } ///< Integral gain
    void setKd(float kd) { this->kd_Ts = kd / this->Ts; } ///< Derivative gain

    float getKp() const { return kp; }          ///< Proportional gain
    float getKi() const { return ki_Ts / Ts; } ///< Integral gain
    float getKd() const { return kd_Ts * Ts; } ///< Derivative gain

    /// Set the cutoff frequency (-3 dB point) of the exponential moving average
    /// filter that is applied to the input before taking the difference for
    /// computing the derivative term.
    void setEMACutoff(float f_c) {
        float f_n = f_c * Ts; // normalized sampling frequency
        this->emaAlpha = f_c == 0 ? 1 : calcAlphaEMA(f_n);
    }

    /// Set the reference/target/setpoint of the controller.
    void setSetpoint(uint16_t setpoint) {
        if (this->setpoint != setpoint) this->activityCount = 0;
        this->setpoint = setpoint;
    }
    /// @see @ref setSetpoint(int16_t)
    uint16_t getSetpoint() const { return setpoint; }

    /// Set the maximum control output magnitude. Default is 255, which clamps
    /// the control output in [-255, +255].
    void setMaxOutput(float maxOutput) { this->maxOutput = maxOutput; }
    /// @see @ref setMaxOutput(float)
    float getMaxOutput() const { return maxOutput; }

    /// Reset the activity counter to prevent the motor from turning off.
    void resetActivityCounter() { this->activityCount = 0; }
    /// Set the number of seconds after which the motor is turned off, zero to
    /// keep it on indefinitely.
    void setActivityTimeout(float s) {
        if (s == 0)
            activityThres = 0;
        else
            activityThres = uint16_t(s / Ts) == 0 ? 1 : s / Ts;
    }

    /// Reset the sum of the previous errors to zero.
    void resetIntegral() { integral = 0; }

  private:
    float Ts = 1;               ///< Sampling time (seconds)
    float maxOutput = 255;      ///< Maximum control output magnitude
    float kp = 1;               ///< Proportional gain
    float ki_Ts = 0;            ///< Integral gain times Ts
    float kd_Ts = 0;            ///< Derivative gain divided by Ts
    float emaAlpha = 1;         ///< Weight factor of derivative EMA filter.
    float prevInput = 0;        ///< (Filtered) previous input for derivative.
    uint16_t activityCount = 0; ///< How many ticks since last setpoint change.
    uint16_t activityThres = 0; ///< Threshold for turning off the output.
    uint8_t errThres = 1;       ///< Threshold with hysteresis.
    int32_t integral = 0;       ///< Sum of previous errors for integral.
    uint16_t setpoint = 0;      ///< Position reference.
};
```

Motor.hpp

```cpp
#pragma once
#include "Registers.hpp"
#include <avr/interrupt.h>
#include <util/atomic.h>

/// Configure Timer0 in either phase correct or fast PWM mode with the given
/// prescaler, enable output compare B.
inline void setupMotorTimer0(bool phase_correct, Timer0Prescaler prescaler) {
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
        setTimer0WGMode(phase_correct ? Timer0WGMode::PWM
                                      : Timer0WGMode::FastPWM);
        setTimer0Prescaler(prescaler);
        sbi(TCCR0A, COM0B1); // Table 14-6, 14-7 Compare Output Mode
        sbi(TCCR0A, COM0A1); // Table 14-6, 14-7 Compare Output Mode
    }
}

/// Configure Timer2 in either phase correct or fast PWM mode with the given
/// prescaler, enable output compare B.
inline void setupMotorTimer2(bool phase_correct, Timer2Prescaler prescaler) {
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
        setTimer2WGMode(phase_correct ? Timer2WGMode::PWM
                                      : Timer2WGMode::FastPWM);
        setTimer2Prescaler(prescaler);
        sbi(TCCR2A, COM2B1); // Table 14-6, 14-7 Compare Output Mode
        sbi(TCCR2A, COM2A1); // Table 14-6, 14-7 Compare Output Mode
    }
}

/// Configure the timers for the PWM outputs.
template <class Config>
inline void setupMotorTimers() {
    constexpr auto prescaler0 = factorToTimer0Prescaler(Config::prescaler_fac);
    static_assert(prescaler0 != Timer0Prescaler::Invalid, "Invalid prescaler");
    constexpr auto prescaler2 = factorToTimer2Prescaler(Config::prescaler_fac);
    static_assert(prescaler2 != Timer2Prescaler::Invalid, "Invalid prescaler");

    if (Config::num_faders > 0)
        setupMotorTimer2(Config::phase_correct_pwm, prescaler2);
    if (Config::num_faders > 2)
        setupMotorTimer0(Config::phase_correct_pwm, prescaler0);
}

/// Class for driving up to 4 DC motors using PWM.
template <class Config>
struct Motors {
    void begin();
    template <uint8_t Idx>
    void setSpeed(int16_t speed);

    template <uint8_t Idx>
    void setupGPIO();
    template <uint8_t Idx>
    void forward(uint8_t speed);
    template <uint8_t Idx>
    void backward(uint8_t speed);
};

template <class Config>
inline void Motors<Config>::begin() {
    setupMotorTimers<Config>();

    if (Config::num_faders > 0) {
        sbi(DDRD, 2);
        sbi(DDRD, 3);
    }
    if (Config::num_faders > 1) {
        if (Config::fader_1_A2)
            sbi(DDRC, 2);
        else
            sbi(DDRB, 5);
        sbi(DDRB, 3);
    }
    if (Config::num_faders > 2) {
        sbi(DDRD, 4);
        sbi(DDRD, 5);
    }
    if (Config::num_faders > 3) {
        sbi(DDRD, 7);
        sbi(DDRD, 6);
    }
}

// Fast PWM (Table 14-6):
//   Clear OC0B on Compare Match, set OC0B at BOTTOM (non-inverting mode).
// Phase Correct PWM (Table 14-7):
//   Clear OC0B on compare match when up-counting. Set OC0B on compare match
//   when down-counting.
template <class Config>
template <uint8_t Idx>
inline void Motors<Config>::forward(uint8_t speed) {
    if (Idx >= Config::num_faders)
        return;
    else if (Idx == 0)
```

```cpp
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            cbi(TCCR2A, COM2B0);
            cbi(PORTD, 2);
            OCR2B = speed;
        }
    else if (Idx == 1)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            cbi(TCCR2A, COM2A0);
            if (Config::fader_1_A2)
                cbi(PORTC, 2);
            else
                cbi(PORTB, 5);
            OCR2A = speed;
        }
    else if (Idx == 2)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            cbi(TCCR0A, COM0B0);
            cbi(PORTD, 4);
            OCR0B = speed;
        }
    else if (Idx == 3)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            cbi(TCCR0A, COM0A0);
            cbi(PORTD, 7);
            OCR0A = speed;
        }
}

// Fast PWM (Table 14-6):
//   Set OC0B on Compare Match, clear OC0B at BOTTOM (inverting mode).
// Phase Correct PWM (Table 14-7):
//   Set OC0B on compare match when up-counting. Clear OC0B on compare match
//   when down-counting.
template <class Config>
template <uint8_t Idx>
inline void Motors<Config>::backward(uint8_t speed) {
    if (Idx >= Config::num_faders)
        return;
    else if (Idx == 0)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            sbi(TCCR2A, COM2B0);
            sbi(PORTD, 2);
            OCR2B = speed;
        }
    else if (Idx == 1)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            sbi(TCCR2A, COM2A0);
            if (Config::fader_1_A2)
                sbi(PORTC, 2);
            else
                sbi(PORTB, 5);
            OCR2A = speed;
        }
    else if (Idx == 2)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            sbi(TCCR0A, COM0B0);
            sbi(PORTD, 4);
            OCR0B = speed;
        }
    else if (Idx == 3)
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
            sbi(TCCR0A, COM0A0);
            sbi(PORTD, 7);
            OCR0A = speed;
        }
}

template <class Config>
template <uint8_t Idx>
inline void Motors<Config>::setSpeed(int16_t speed) {
    if (speed >= 0)
        forward<Idx>(speed);
    else
        backward<Idx>(-speed);
}
```

Reference.hpp

```cpp
#include <avr/pgmspace.h>
#include <stddef.h>
#include <stdint.h>
#include <util/atomic.h>

/// Reference signal for testing the controller.
const uint8_t reference_signal[] PROGMEM = {
    // Ramp up
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
    21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
    59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,
    78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
    97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
    113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127,
    128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
    143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157,
    158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172,
    173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187,
    188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202,
    203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217,
    218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232,
    233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247,
    248, 249, 250, 251, 252, 253, 254, 255,
    // Max
    255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255,
    // Ramp down
    255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241,
    240, 239, 238, 237, 236, 235, 234, 233, 232, 231, 230, 229, 228, 227, 226,
    225, 224, 223, 222, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211,
    210, 209, 208, 207, 206, 205, 204, 203, 202, 201, 200, 199, 198, 197, 196,
    195, 194, 193, 192, 191, 190, 189, 188, 187, 186, 185, 184, 183, 182, 181,
    180, 179, 178, 177, 176, 175, 174, 173, 172, 171, 170, 169, 168, 167, 166,
    165, 164, 163, 162, 161, 160, 159, 158, 157, 156, 155, 154, 153, 152, 151,
    150, 149, 148, 147, 146, 145, 144, 143, 142, 141, 140, 139, 138, 137, 136,
    135, 134, 133, 132, 131, 130, 129, 128, 127,
    // Middle
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127,
    // Jump low
    10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
    10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
    10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
    10, 10, 10, 10, 10, 10, 10,
    // Jump middle
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127,
    // Jump high
    245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245,
    245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245,
    245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245,
    245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245, 245,
    245, 245, 245, 245,
    // Jump middle
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
    127, 127, 127, 127, 127, 127, 127, 127,

    // Ramp down
    127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113,
    112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97,
    96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78,
    77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59,
    58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40,
    39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
    20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
    // Low
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0};

/// Get the number of elements in the given array.
template <class T, size_t N>
constexpr size_t len(const T (&)[N]) {
```

```
    return N;
}

/// Class that handles the three main references/setpoints:
///   1. Sequence programmed in PROGMEM
///   2. Test sequence for tuning experiments (activated over Serial)
///   3. Setpoint set by the I²C master (for real-world use)
template <class Config>
class Reference {
  public:
    /// Called from ISR
    void setMasterSetpoint(uint16_t setpoint) {
        this->master_setpoint = setpoint;
    }

    void startExperiment(float speed_div) {
        this->experiment_speed_div = speed_div;
        this->index = 0;
        this->seq_idx = 0;
    }

    bool experimentInProgress() const { return experiment_speed_div > 0; }

    uint16_t getNextProgmemSetpoint() {
        uint16_t setpoint = pgm_read_byte(reference_signal + index) * 4;
        ++seq_idx;
        if (seq_idx >= Config::test_reference_speed_div) {
            seq_idx = 0;
            ++index;
            if (index == len(reference_signal)) index = 0;
        }
        return setpoint;
    }

    uint16_t getNextExperimentSetpoint() {
        constexpr uint16_t RAMPUP = 0xFFFF;
        auto rampup = [](uint16_t idx, uint16_t duration) {
            return uint32_t(1024) * idx / duration;
        };
        constexpr uint16_t RAMPDOWN = 0xFFFE;
        auto rampdown = [&](uint16_t idx, uint16_t duration) {
            return 1023 - rampup(idx, duration);
        };
        struct TestSeq {
            uint16_t setpoint;
            uint16_t duration;
        };
        // This array defines the test sequence
        constexpr static TestSeq seqs[] {
            {0, 256},  {RAMPUP, 128}, {1023, 32}, {0, 64},    {333, 32},
            {666, 32}, {333, 32},     {0, 32},    {512, 256},
        };

        static uint8_t seq_index = 0;
        static uint16_t index = 0;
        uint16_t duration = seqs[seq_index].duration * experiment_speed_div;
        uint16_t seq_setpoint = seqs[seq_index].setpoint;
        uint16_t setpoint;
        switch (seq_setpoint) {
            case RAMPUP: setpoint = rampup(index, duration); break;
            case RAMPDOWN: setpoint = rampdown(index, duration); break;
            default: setpoint = seq_setpoint;
        }
        ++index;
        if (index == duration) {
            index = 0;
            ++seq_index;
            if (seq_index == len(seqs)) {
                seq_index = 0;
                experiment_speed_div = 0;
            }
        }
        return setpoint;
    }

    /// Called from main program with interrupts enabled
    uint16_t getNextSetpoint() {
        uint16_t setpoint;
        if (Config::serial_control && experiment_speed_div > 0)
            // from the tuning experiment reference
            setpoint = getNextExperimentSetpoint();
        else if (Config::test_reference)
            // from the test reference
            setpoint = getNextProgmemSetpoint();
        else
            // from the I²C master
            ATOMIC_BLOCK(ATOMIC_FORCEON) { setpoint = master_setpoint; }
        return setpoint;
    }

  private:
    uint16_t index = 0;
    uint8_t seq_idx = 0;
    float experiment_speed_div = 0;
```

```
    volatile uint16_t master_setpoint = 0;
};
```

Registers.hpp

```cpp
#pragma once

#include <avr/io.h>
#include <avr/sfr_defs.h>
#include <util/delay.h> // F_CPU

// ------------------------------- Utils -------------------------------- //

#ifndef ARDUINO // Ensures that my IDE sees the correct frequency
#undef F_CPU
#define F_CPU 16000000UL
#endif

#ifndef sbi
/// Set bit in register.
template <class R>
inline void sbi(R &reg, uint8_t bit) {
    reg |= (1u << bit);
}
#define sbi sbi
#endif
#ifndef cbi
/// Clear bit in register.
template <class R>
inline void cbi(R &reg, uint8_t bit) {
    reg &= ~(1u << bit);
}
#define cbi cbi
#endif
/// Write bit in register.
template <class R>
inline void wbi(R &reg, uint8_t bit, bool value) {
    value ? sbi(reg, bit) : cbi(reg, bit);
}

// ------------------------------- Timer0 ------------------------------- //

/// Timer 0 clock select (Table 14-9).
enum class Timer0Prescaler : uint8_t {
    None = 0b000,
    S1 = 0b001,
    S8 = 0b010,
    S64 = 0b011,
    S256 = 0b100,
    S1024 = 0b101,
    ExtFall = 0b110,
    ExtRise = 0b111,
    Invalid = 0xFF,
};

/// Timer 0 waveform generation mode (Table 14-8).
enum class Timer0WGMode : uint8_t {
    Normal = 0b000,
    PWM = 0b001,
    CTC = 0b010,
    FastPWM = 0b011,
    PWM_OCRA = 0b101,
    FastPWM_OCRA = 0b111,
};

// Convert the prescaler factor to the correct bit pattern to write to the
// TCCR0B register (Table 14-9).
constexpr inline Timer0Prescaler factorToTimer0Prescaler(uint16_t factor) {
    return factor == 1      ? Timer0Prescaler::S1
           : factor == 8    ? Timer0Prescaler::S8
           : factor == 64   ? Timer0Prescaler::S64
           : factor == 256  ? Timer0Prescaler::S256
           : factor == 1024 ? Timer0Prescaler::S1024
                            : Timer0Prescaler::Invalid;
}

/// Set the clock source/prescaler of Timer0 (Table 14-9).
inline void setTimer0Prescaler(Timer0Prescaler ps) {
    if (ps == Timer0Prescaler::Invalid)
        return;
    wbi(TCCR0B, CS02, static_cast<uint8_t>(ps) & (1u << 2));
    wbi(TCCR0B, CS01, static_cast<uint8_t>(ps) & (1u << 1));
    wbi(TCCR0B, CS00, static_cast<uint8_t>(ps) & (1u << 0));
}

/// Set the wavefrom generation mode of Timer0 (Table 14-8).
inline void setTimer0WGMode(Timer0WGMode mode) {
    wbi(TCCR0B, WGM02, static_cast<uint8_t>(mode) & (1u << 2));
    wbi(TCCR0A, WGM01, static_cast<uint8_t>(mode) & (1u << 1));
    wbi(TCCR0A, WGM00, static_cast<uint8_t>(mode) & (1u << 0));
}

// ------------------------------- Timer2 ------------------------------- //

/// Timer 0 clock select (Table 17-9).
enum class Timer2Prescaler : uint8_t {
    None = 0b000,
    S1 = 0b001,
    S8 = 0b010,
```

```
    S32 = 0b011,
    S64 = 0b100,
    S128 = 0b101,
    S256 = 0b110,
    S1024 = 0b111,
    Invalid = 0xFF,
};

/// Timer 0 waveform generation mode (Table 17-8).
enum class Timer2WGMode : uint8_t {
    Normal = 0b000,
    PWM = 0b001,
    CTC = 0b010,
    FastPWM = 0b011,
    PWM_OCRA = 0b101,
    FastPWM_OCRA = 0b111,
};

// Convert the prescaler factor to the correct bit pattern to write to the
// TCCR0B register (Table 17-9).
constexpr inline Timer2Prescaler factorToTimer2Prescaler(uint16_t factor) {
    return factor == 1      ? Timer2Prescaler::S1
           : factor == 8    ? Timer2Prescaler::S8
           : factor == 32   ? Timer2Prescaler::S32
           : factor == 64   ? Timer2Prescaler::S64
           : factor == 128  ? Timer2Prescaler::S128
           : factor == 256  ? Timer2Prescaler::S256
           : factor == 1024 ? Timer2Prescaler::S1024
                            : Timer2Prescaler::Invalid;
}

/// Set the clock source/prescaler of Timer2 (Table 17-9).
inline void setTimer2Prescaler(Timer2Prescaler ps) {
    if (ps == Timer2Prescaler::Invalid)
        return;
    wbi(TCCR2B, CS22, static_cast<uint8_t>(ps) & (1u << 2));
    wbi(TCCR2B, CS21, static_cast<uint8_t>(ps) & (1u << 1));
    wbi(TCCR2B, CS20, static_cast<uint8_t>(ps) & (1u << 0));
}

/// Set the wavefrom generation mode of Timer2 (Table 17-8).
inline void setTimer2WGMode(Timer2WGMode mode) {
    wbi(TCCR2B, WGM22, static_cast<uint8_t>(mode) & (1u << 2));
    wbi(TCCR2A, WGM21, static_cast<uint8_t>(mode) & (1u << 1));
    wbi(TCCR2A, WGM20, static_cast<uint8_t>(mode) & (1u << 0));
}

// ---------------------------------- ADC ----------------------------------- //

/// ADC prescaler select (Table 23-5).
enum class ADCPrescaler : uint8_t {
    S2 = 0b000,
    S2_2 = 0b001,
    S4 = 0b010,
    S8 = 0b011,
    S16 = 0b100,
    S32 = 0b101,
    S64 = 0b110,
    S128 = 0b111,
    Invalid = 0xFF,
};

// Convert the prescaler factor to the correct bit pattern to write to the
// ADCSRA register (Table 23-5).
constexpr inline ADCPrescaler factorToADCPrescaler(uint8_t factor) {
    return factor == 2     ? ADCPrescaler::S2_2
           : factor == 4   ? ADCPrescaler::S4
           : factor == 8   ? ADCPrescaler::S8
           : factor == 16  ? ADCPrescaler::S16
           : factor == 32  ? ADCPrescaler::S32
           : factor == 64  ? ADCPrescaler::S64
           : factor == 128 ? ADCPrescaler::S128
                           : ADCPrescaler::Invalid;
}

/// Set the prescaler of the ADC (Table 23-5).
inline void setADCPrescaler(ADCPrescaler ps) {
    if (ps == ADCPrescaler::Invalid)
        return;
    wbi(ADCSRA, ADPS2, static_cast<uint8_t>(ps) & (1u << 2));
    wbi(ADCSRA, ADPS1, static_cast<uint8_t>(ps) & (1u << 1));
    wbi(ADCSRA, ADPS0, static_cast<uint8_t>(ps) & (1u << 0));
}
```

SerialSLIP.hpp

```cpp
#include <Arduino.h>

namespace SLIP_Constants {
const static uint8_t END = 0300;
const static uint8_t ESC = 0333;
const static uint8_t ESC_END = 0334;
const static uint8_t ESC_ESC = 0335;
} // namespace SLIP_Constants

/// Parses SLIP packets: https://datatracker.ietf.org/doc/html/rfc1055
class SLIPParser {
  public:
    template <class Callback>
    size_t parse(uint8_t c, Callback callback);

    void reset() {
        size = 0;
        escape = false;
    }

  private:
    size_t size = 0;
    bool escape = false;
};

template <class Callback>
size_t SLIPParser::parse(uint8_t c, Callback callback) {
    using namespace SLIP_Constants;
    /*
     * handle bytestuffing if necessary
     */
    switch (c) {
        /*
         * if it's an END character then we're done with
         * the packet
         */
        case END: {
            /*
             * a minor optimization: if there is no
             * data in the packet, ignore it. This is
             * meant to avoid bothering IP with all
             * the empty packets generated by the
             * duplicate END characters which are in
             * turn sent to try to detect line noise.
             */
            auto packetLen = size;
            reset();
            if (packetLen) return packetLen;
        } break;

        /*
         * if it's the same code as an ESC character, wait
         * and get another character and then figure out
         * what to store in the packet based on that.
         */
        case ESC: {
            escape = true;
        } break;

        /*
         * here we fall into the default handler and let
         * it store the character for us
         */
        default: {
            if (escape) {
                /*
                 * if "c" is not one of these two, then we
                 * have a protocol violation.  The best bet
                 * seems to be to leave the byte alone and
                 * just stuff it into the packet
                 */
                switch (c) {
                    case ESC_END: c = END; break;
                    case ESC_ESC: c = ESC; break;
                    default: break; // LCOV_EXCL_LINE (protocol violation)
                }
                escape = false;
            }
            callback(c, size);
            ++size;
        }
    }
    return 0;
}

/// Sends SLIP packets: https://datatracker.ietf.org/doc/html/rfc1055
class SLIPSender {
  public:
    SLIPSender(Stream &stream) : stream(stream) {}

    size_t beginPacket() { return stream.write(SLIP_Constants::END); }
    size_t endPacket() { return stream.write(SLIP_Constants::END); }

    size_t write(const uint8_t *data, size_t len);
```

```
    size_t writePacket(const uint8_t *data, size_t len) {
        size_t sent = 0;
        sent += beginPacket();
        sent += write(data, len);
        sent += endPacket();
        return sent;
    }

  private:
    Stream &stream;
};

inline size_t SLIPSender::write(const uint8_t *data, size_t len) {
    // https://datatracker.ietf.org/doc/html/rfc1055
    using namespace SLIP_Constants;
    size_t sent = 0;
    /*
     * for each byte in the packet, send the appropriate character
     * sequence
     */
    while (len--) {
        switch (*data) {
            /*
             * if it's the same code as an END character, we send a
             * special two character code so as not to make the
             * receiver think we sent an END
             */
            case END:
                sent += stream.write(ESC);
                sent += stream.write(ESC_END);
                break;

            /*
             * if it's the same code as an ESC character,
             * we send a special two character code so as not
             * to make the receiver think we sent an ESC
             */
            case ESC:
                sent += stream.write(ESC);
                sent += stream.write(ESC_ESC);
                break;

            /*
             * otherwise, we just send the character
             */
            default: sent += stream.write(*data);
        }
        data++;
    }
    return sent;
}
```

SMA.hpp

```cpp
#pragma once

#include <Arduino_Helpers.h>

#include <AH/Math/Divide.hpp>
#include <AH/STL/algorithm> // std::fill
#include <AH/STL/cstdint>

/**
 * @brief   Simple Moving Average filter.
 *
 * Returns the average of the N most recent input values.
 *
 * @f[
 * y[n] = \frac{1}{N} \sum_{i=0}^{N-1}x[n-i]
 * @f]
 *
 * @see     https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Simple%20Moving%20Average/Simple-Moving-Average.html
 *
 * @tparam  N
 *          The number of samples to average.
 * @tparam  input_t
 *          The type of the input (and output) of the filter.
 * @tparam  sum_t
 *          The type to use for the accumulator, must be large enough to fit
 *          N times the maximum input value.
 */
template <uint8_t N, class input_t = uint16_t, class sum_t = uint32_t>
class SMA {
  public:
    /// Default constructor (initial state is initialized to all zeros).
    SMA() = default;

    /// Constructor (initial state is initialized to given value).
    ///
    /// @param  initialValue
    ///         Determines the initial state of the filter:
    ///         @f$ x[-N] =\ \ldots\ = x[-2] = x[-1] = \text{initialValue} @f$
    ///
    SMA(input_t initialValue) : sum(N * (sum_t)initialValue) {
        std::fill(std::begin(previousInputs), std::end(previousInputs),
                  initialValue);
    }

    /// Update the internal state with the new input @f$ x[n] @f$ and return the
    /// new output @f$ y[n] @f$.
    ///
    /// @param  input
    ///         The new input @f$ x[n] @f$.
    /// @return The new output @f$ y[n] @f$.
    input_t operator()(input_t input) {
        sum -= previousInputs[index];
        sum += input;
        previousInputs[index] = input;
        if (++index == N) index = 0;
        return AH::round_div<N>(sum);
    }

  private:
    uint8_t index = 0;
    input_t previousInputs[N] {};
    sum_t sum = 0;
};
```