

7. ATmega328P Code

Pieter P

main.cpp

```
1 // Configuration and initialization of the analog-to-digital converter:
2 #include "ADC.hpp"
3 // Capacitive touch sensing:
4 #include "Touch.hpp"
5 // PID controller:
6 #include "Controller.hpp"
7 // Configuration of PWM and Timer2/0 for driving the motor:
8 #include "Motor.hpp"
9 // Reference signal for testing the performance of the controller:
10 #include "Reference.hpp"
11 // Helpers for low-level AVR Timer2/0 and ADC registers:
12 #include "Registers.hpp"
13 // Parsing incoming messages over Serial using SLIP packets:
14 #include "SerialSLIP.hpp"
15
16 #include <Arduino.h>           // setup, loop, analogRead
17 #include <Arduino_Helpers.h>   // EMA.hpp
18 #include <Wire.h>              // I²C slave
19
20 #include "SMA.hpp"             // SMA filter
21 #include <AH/Filters/EMA.hpp>   // EMA filter
22
23 // ----- Description ----- //
24
25 // This sketch drives up to four motorized faders using a PID controller. The
26 // motor is disabled when the user touches the knob of the fader.
27 //
28 // Everything is driven by Timer2, which runs (by default) at a rate of
29 // 31.250 kHz. This high rate is used to eliminate audible tones from the PWM
30 // drive for the motor. Timer0 is used for the PWM outputs of faders 3 and 4.
31 // Every 30 periods of Timer2 (960 µs), each analog input is sampled, and
32 // this causes the PID control loop to run in the main loop function.
33 // Capacitive sensing is implemented by measuring the RC time on the touch pin
34 // in the Timer2 interrupt handler. The "touched" status is sticky for >20 ms
35 // to prevent interference from the 50 Hz mains.
36 //
37 // There are options to (1) follow a test reference (with ramps and jumps), (2)
38 // to receive a target position over I²C, or (3) to run experiments based on
39 // commands received over the serial port. The latter is used by a Python script
40 // that performs experiments with different tuning parameters for the
41 // controllers.
42
43 // ----- Hardware ----- //
44
45 // Fader 0:
46 // - A0: wiper of the potentiometer           (ADC0)
47 // - D8: touch pin of the knob                 (PB0)
48 // - D2: input 1A of L293D dual H-bridge 1     (PD2)
49 // - D3: input 2A of L293D dual H-bridge 1     (OC2B)
50 //
51 // Fader 1:
52 // - A1: wiper of the potentiometer           (ADC1)
53 // - D9: touch pin of the knob                 (PB1)
54 // - D7: input 3A of L293D dual H-bridge 1     (PD7)
55 // - D11: input 4A of L293D dual H-bridge 1    (OC2A)
56 //
57 // Fader 2:
58 // - A2: wiper of the potentiometer           (ADC2)
59 // - D10: touch pin of the knob                (PB2)
60 // - D4: input 1A of L293D dual H-bridge 2     (PD4)
61 // - D5: input 2A of L293D dual H-bridge 2     (OC0B)
62 //
63 // Fader 3:
64 // - A3: wiper of the potentiometer           (ADC3)
65 // - D12: touch pin of the knob                (PB4)
66 // - D13: input 3A of L293D dual H-bridge 2    (PB5)
67 // - D6: input 4A of L293D dual H-bridge 2    (OC0A)
68 //
69 // If fader 1 is unused:
70 // - D13: LED or scope as overrun indicator    (PB5)
71 //
72 // For communication:
73 // - D0: UART TX                               (TXD)
74 // - D1: UART RX                               (RXD)
75 // - A4: I²C data                             (SDA)
76 // - A5: I²C clock                           (SCL)
77 //
78 // Connect the outer connections of the potentiometers to ground and Vcc, it's
79 // recommended to add a 100 nF capacitor between each wiper and ground.
80 // Connect the 1,2EN and 3,4EN enable pins of the L293D chips to Vcc.
81 // Connect a 500kΩ pull-up resistor between each touch pin and Vcc.
82 // On an Arduino Nano, you can set an option to use pins A6/A7 instead of A2/A3.
83 // Note that D13 is often pulsed by the bootloader, which might cause the fader
84 // to move when resetting the Arduino. You can either disable this behavior in
85 // the bootloader, or use a different pin (e.g. A2 or A3 on an Arduino Nano).
86 // The overrun indicator is only enabled if the number of faders is less than 4,
87 // because it conflicts with the motor driver pin of Fader 1. You can choose a
88 // different pin instead.
89
90 // ----- Configuration ----- //
91
92 // Enable MIDI input/output.
```

```

93 #define WITH_MIDI 0
94 // Print to the Serial monitor instead of sending actual MIDI messages.
95 #define MIDI_DEBUG 0
96
97 struct Config {
98     // Print the control loop and interrupt frequencies to Serial at startup:
99     static constexpr bool print_frequencies = true;
100     // Print the setpoint, actual position and control signal to Serial.
101     // Note that this slows down the control loop significantly, it probably
102     // won't work if you are using more than one fader without increasing
103     // `interrupt_divisor`:
104     static constexpr bool print_controller_signals = false;
105     static constexpr uint8_t controller_to_print = 0;
106     // Follow the test reference trajectory (true) or receive the target
107     // position over I2C or Serial (false):
108     static constexpr bool test_reference = false;
109     // Increase this divisor to slow down the test reference:
110     static constexpr uint8_t test_reference_speed_div = 4;
111     // Allow control for tuning and starting experiments over Serial:
112     static constexpr bool serial_control = true;
113     // I2C slave address (zero to disable I2C):
114     static constexpr uint8_t i2c_address = 8;
115     // The baud rate to use for the Serial interface (e.g. for MIDI_DEBUG,
116     // print_controller_signals, serial_control, etc.)
117     static constexpr uint32_t serial_baud_rate = 1000000;
118     // The baud rate to use for MIDI over Serial.
119     // Use 31'250 for MIDI over 5-pin DIN, HIDUINO/USBMidiklik.
120     // Hairless MIDI uses 115'200 by default.
121     // The included python/SerialMIDI.py script uses 1'000'000.
122     static constexpr uint32_t midi_baud_rate = serial_baud_rate;
123
124     // Number of faders, must be between 1 and 4:
125     static constexpr size_t num_faders = 1;
126     // Actually drive the motors. If set to false, runs all code as normal, but
127     // doesn't turn on the motors.
128     static constexpr bool enable_controller = true;
129     // Use analog pins (A0, A1, A6, A7) instead of (A0, A1, A2, A3), useful for
130     // saving digital pins on an Arduino Nano:
131     static constexpr bool use_A6_A7 = false;
132     // Use pin A2 instead of D13 as the motor driver pin for the fourth fader.
133     // Allows D13 to be used as overrun indicator, and avoids issues with the
134     // bootloader blinking the LED.
135     // Can only be used if `use_A6_A7` is set to true.
136     static constexpr bool fader_3_A2 = false;
137     // Change the setpoint to the current position when touching the knob.
138     // Useful if your DAW does not send any feedback when manually moving the
139     // fader.
140     static constexpr bool touch_to_current_position = false;
141
142     // Capacitive touch sensing RC time threshold.
143     // Increase this time constant if the capacitive touch sense is too
144     // sensitive or decrease it if it's not sensitive enough:
145     static constexpr float touch_rc_time_threshold = 150e-6; // seconds
146     // Bit masks of the touch pins (must be on port B):
147     static constexpr uint8_t touch_masks[] = {1 << PB0, 1 << PB1, 1 << PB2,
148                                               1 << PB4};
149
150     // Use phase-correct PWM (true) or fast PWM (false), this determines the
151     // timer interrupt frequency, prefer phase-correct PWM with prescaler 1 on
152     // 16 MHz boards, and fast PWM with prescaler 1 on 8 MHz boards, both result
153     // in a PWM and interrupt frequency of 31.250 kHz
154     // (fast PWM is twice as fast):
155     static constexpr bool phase_correct_pwm = true;
156     // The fader position will be sampled once per `interrupt_divisor` timer
157     // interrupts, this determines the sampling frequency of the control loop.
158     // Some examples include 20 → 320 μs, 30 → 480 μs, 60 → 960 μs,
159     // 90 → 1,440 μs, 124 → 2,016 μs, 188 → 3,008 μs, 250 → 4,000 μs.
160     // 60 is the default, because it works with four faders. If you only use
161     // a single fader, you can go as low as 20 because you only need a quarter
162     // of the computations and ADC time:
163     static constexpr uint8_t interrupt_divisor = 60 / (1 + phase_correct_pwm);
164     // The prescaler for the timer, affects PWM and control loop frequencies:
165     static constexpr unsigned prescaler_fac = 1;
166     // The prescaler for the ADC, affects speed of analog readings:
167     static constexpr uint8_t adc_prescaler_fac = 64;
168
169     // Turn off the motor after this many seconds of inactivity:
170     static constexpr float timeout = 2;
171
172     // EMA filter factor for fader position filters:
173     static constexpr uint8_t adc_ema_K = 2;
174     // SMA filter length for setpoint filters, improves tracking of ramps if the
175     // setpoint changes in steps (e.g. when the DAW only updates the reference
176     // every 20 ms). Powers of two are significantly faster (e.g. 32 works well):
177     static constexpr uint8_t setpoint_sma_length = 0;
178
179     // ----- Computed Quantities ----- //
180
181     // Sampling time of control loop:
182     constexpr static float Ts = 1. * prescaler_fac * interrupt_divisor * 256 *
183         (1 + phase_correct_pwm) / F_CPU;
184     // Frequency at which the interrupt fires:
185     constexpr static float interrupt_freq =
186         1. * F_CPU / prescaler_fac / 256 / (1 + phase_correct_pwm);
187     // Clock speed of the ADC:

```

```

188 constexpr static float adc_clock_freq = 1. * F_CPU / adc_prescaler_fac;
189 // Pulse pin D13 if the control loop took too long:
190 constexpr static bool enable_overrun_indicator =
191     num_faders < 4 || fader_3_A2;
192
193 static_assert(0 < num_faders && num_faders <= 4,
194     "At most four faders supported");
195 static_assert(use_A6_A7 || !fader_3_A2,
196     "Cannot use A2 for motor driver "
197     "and analog input at the same time");
198 static_assert(!WITH_MIDI || !serial_control,
199     "Cannot use MIDI and Serial control at the same time");
200 static_assert(!WITH_MIDI || !print_controller_signals,
201     "Cannot use MIDI while printing controller signals");
202 };
203 constexpr uint8_t Config::touch_masks[];
204 constexpr float Ts = Config::Ts;
205
206 // ----- ADC, Capacitive Touch State and Motors ----- //
207
208 ADCManager<Config> adc;
209 TouchSense<Config> touch;
210 Motors<Config> motors;
211
212 // ----- Setpoints and References ----- //
213
214 // Setpoints (target positions) for all faders:
215 Reference<Config> references[Config::num_faders];
216
217 // ----- Controllers ----- //
218
219 // The main PID controllers. Need tuning for your specific setup:
220
221 PID controllers[] {
222     // This is an example of a controller with very little overshoot
223     {
224         6, // Kp: proportional gain
225         2, // Ki: integral gain
226         0.035, // Kd: derivative gain
227         Ts, // Ts: sampling time
228         60, // fc: cutoff frequency of derivative filter (Hz), zero to disable
229     },
230     // This one has more overshoot, but less ramp tracking error
231     {
232         4, // Kp: proportional gain
233         11, // Ki: integral gain
234         0.028, // Kd: derivative gain
235         Ts, // Ts: sampling time
236         40, // fc: cutoff frequency of derivative filter (Hz), zero to disable
237     },
238     // This is a very aggressive controller
239     {
240         8.55, // Kp: proportional gain
241         440, // Ki: integral gain
242         0.043, // Kd: derivative gain
243         Ts, // Ts: sampling time
244         70, // fc: cutoff frequency of derivative filter (Hz), zero to disable
245     },
246     // Fourth controller
247     {
248         6, // Kp: proportional gain
249         2, // Ki: integral gain
250         0.035, // Kd: derivative gain
251         Ts, // Ts: sampling time
252         60, // fc: cutoff frequency of derivative filter (Hz), zero to disable
253     },
254 };
255
256 // ----- MIDI ----- //
257
258 #if WITH_MIDI
259 #include <Control_Surface.h>
260
261 #if MIDI_DEBUG
262 USBDebugMIDI_Interface midi {Config::serial_baud_rate};
263 #else
264 HardwareSerialMIDI_Interface midi {Serial, Config::midi_baud_rate};
265 #endif
266
267 template <uint8_t Idx>
268 void sendMIDIMessages(bool touched) {
269     // Don't send if the UART buffer is (almost) full
270     if (Serial.availableForWrite() < 6) return;
271     // Touch
272     static bool prevTouched = false; // Whether the knob is being touched
273     if (touched != prevTouched) {
274         const MIDIAddress addr = MCU::FADER_TOUCH_1 + Idx;
275         touched ? midi.sendNoteOn(addr, 127) : midi.sendNoteOff(addr, 127);
276         prevTouched = touched;
277     }
278     // Position
279     static Hysteresis<6 - Config::adc_ema_K, uint16_t, uint16_t> hyst;
280     if (prevTouched && hyst.update(adc.readFiltered(Idx))) {
281         auto value = AH::increaseBitDepth<14, 10, uint16_t>(hyst.getValue());
282         midi.sendPitchBend(MCU::VOLUME_1 + Idx, value);

```

```

283     }
284 }
285
286 void updateMIDISetpoint(ChannelMessage msg) {
287     auto type = msg.getMessageType();
288     auto channel = msg.getChannel().getRaw();
289     if (type == MIDIMessageType::PITCH_BEND && channel < Config::num_faders)
290         references[channel].setMasterSetpoint(msg.getData14bit() >> 4);
291 }
292
293 void initMIDI() { midi.begin(); }
294
295 void updateMIDI() {
296     while (1) {
297         auto evt = midi.read();
298         if (evt == MIDIReadEvent::NO_MESSAGE)
299             break;
300         else if (evt == MIDIReadEvent::CHANNEL_MESSAGE)
301             updateMIDISetpoint(midi.getChannelMessage());
302     }
303 }
304
305 #endif
306
307 // ----- Printing all signals for serial plotter ----- //
308
309 template <uint8_t Idx>
310 void printControllerSignals(int16_t setpoint, int16_t adcval, int16_t control) {
311     // Send (binary) controller signals over Serial to plot in Python
312     if (Config::serial_control && references[Idx].experimentInProgress()) {
313         const int16_t data[3] {setpoint, adcval, control};
314         SLIPSender(Serial).writePacket(reinterpret_cast<const uint8_t*>(data),
315                                         sizeof(data));
316     }
317     // Print signals as text
318     else if (Config::print_controller_signals &&
319             Idx == Config::controller_to_print) {
320         Serial.print(setpoint);
321         Serial.print('\t');
322         Serial.print(adcval);
323         Serial.print('\t');
324         Serial.print((control + 256) * 2);
325         Serial.println();
326     }
327 }
328
329 // ----- Control logic ----- //
330
331 template <uint8_t Idx>
332 void updateController(uint16_t setpoint, int16_t adcval, bool touched) {
333     auto &controller = controllers[Idx];
334
335     // Prevent the motor from being turned off after being touched
336     if (touched) controller.resetActivityCounter();
337
338     // Set the target position
339     if (Config::setpoint_sma_length > 0) {
340         static SMA<Config::setpoint_sma_length, uint16_t, uint32_t> sma;
341         uint16_t filtsetpoint = sma(setpoint);
342         controller.setSetpoint(filtsetpoint);
343     } else {
344         controller.setSetpoint(setpoint);
345     }
346
347     // Update the PID controller to get the control action
348     int16_t control = controller.update(adcval);
349
350     // Apply the control action to the motor
351     if (Config::enable_controller) {
352         if (touched) // Turn off motor if knob is touched
353             motors.setSpeed<Idx>(0);
354         else
355             motors.setSpeed<Idx>(control);
356     }
357
358     // Change the setpoint as we move
359     if (Config::touch_to_current_position && touched)
360         references[Idx].setMasterSetpoint(adcval);
361
362 #if WITH_MIDI
363     sendMIDIMessages<Idx>(touched);
364 #else
365     printControllerSignals<Idx>(controller.getSetpoint(), adcval, control);
366 #endif
367 }
368
369 template <uint8_t Idx>
370 void readAndUpdateController() {
371     // Read the ADC value for the given fader:
372     int16_t adcval = adc.read(Idx);
373     // If the ADC value was updated by the ADC interrupt, run the control loop:
374     if (adcval >= 0) {
375         // Check if the fader knob is touched
376         bool touched = touch.read(Idx);
377         // Read the target position

```

```

378     uint16_t setpoint = references[Idx].getNextSetpoint();
379     // Run the control loop
380     updateController<Idx>(setpoint, adcval, touched);
381     // Write -1 so the controller doesn't run again until the next value is
382     // available:
383     adc.write(Idx, -1);
384     if (Config::enable_overrun_indicator)
385         cbi(PORTB, 5); // Clear overrun indicator
386 }
387 }
388
389 // ----- Setup & Loop ----- //
390
391 void onRequest();
392 void onReceive(int);
393 void updateSerialIn();
394
395 void setup() {
396     // Initialize some globals
397     for (uint8_t i = 0; i < Config::num_faders; ++i) {
398         // all fader positions for the control loop start of as -1 (no reading)
399         adc.write(i, -1);
400         // reset the filter to the current fader position to prevent transients
401         adc.writeFiltered(i, analogRead(adc.channel_index_to_mux_address(i)));
402         // after how many seconds of not touching the fader and not changing
403         // the reference do we turn off the motor?
404         controllers[i].setActivityTimeout(Config::timeout);
405     }
406
407     // Configure the hardware
408     if (Config::enable_overrun_indicator) sbi(DDRB, 5); // Pin 13 output
409
410     #if WITH_MIDI
411         initMIDI();
412     #else
413         if (Config::print_frequencies || Config::print_controller_signals ||
414             Config::serial_control)
415             Serial.begin(Config::serial_baud_rate);
416     #endif
417
418     adc.begin();
419     touch.begin();
420     motors.begin();
421
422     // Print information to the serial monitor or legends to the serial plotter
423     if (Config::print_frequencies) {
424         Serial.println();
425         Serial.print(F("Interrupt frequency (Hz): "));
426         Serial.println(Config::interrupt_freq);
427         Serial.print(F("Controller sampling time (µs): "));
428         Serial.println(Config::Ts * 1e6);
429         Serial.print(F("ADC clock rate (Hz): "));
430         Serial.println(Config::adc_clock_freq);
431         Serial.print(F("ADC sampling rate (Sps): "));
432         Serial.println(adc.adc_rate);
433     }
434     if (Config::print_controller_signals) {
435         Serial.println();
436         Serial.println(F("Reference\tActual\tControl\t-"));
437         Serial.println(F("0\t0\t0\t0\r\n0\t0\t0\t024"));
438     }
439
440     // Initialize I²C slave and attach callbacks
441     if (Config::i2c_address) {
442         Wire.begin(Config::i2c_address);
443         Wire.onRequest(onRequest);
444         Wire.onReceive(onReceive);
445     }
446
447     // Enable Timer2 overflow interrupt, this starts reading the touch sensitive
448     // knobs and sampling the ADC, which causes the controllers to run in the
449     // main loop
450     sbi(TIMSK2, TOIE2);
451 }
452
453 void loop() {
454     if (Config::num_faders > 0) readAndUpdateController<0>();
455     if (Config::num_faders > 1) readAndUpdateController<1>();
456     if (Config::num_faders > 2) readAndUpdateController<2>();
457     if (Config::num_faders > 3) readAndUpdateController<3>();
458     #if WITH_MIDI
459         updateMIDI();
460     #else
461         if (Config::serial_control) updateSerialIn();
462     #endif
463 }
464
465 // ----- Interrupts ----- //
466
467 // Fires at a constant rate of `interrupt_freq`.
468 ISR(TIMER2_OVF_vect) {
469     // We don't have to take all actions at each interrupt, so keep a counter to
470     // know when to take what actions.
471     static uint8_t counter = 0;
472

```

```

473     adc.update(counter);
474     touch.update(counter);
475
476     ++counter;
477     if (counter == Config::interrupt_divisor) counter = 0;
478 }
479
480 // Fires when the ADC measurement is complete. Stores the reading, both before
481 // and after filtering (for the controller and for user input respectively).
482 ISR(ADC_vect) { adc.complete(); }
483
484 // ----- Wire ----- //
485
486 // Send the touch status and filtered fader positions to the master.
487 void onRequest() {
488     uint8_t touched = 0;
489     for (uint8_t i = 0; i < Config::num_faders; ++i)
490         touched |= touch.touched[i] << i;
491     Wire.write(touched);
492     for (uint8_t i = 0; i < Config::num_faders; ++i) {
493         uint16_t filt_read = adc.readFiltered14ISR(i);
494         Wire.write(reinterpret_cast<const uint8_t *>(&filt_read), 2);
495     }
496 }
497
498 // Change the setpoint of the given fader based on the value in the message
499 // received from the master.
500 void onReceive(int count) {
501     if (count < 2) return;
502     if (Wire.available() < 2) return;
503     uint16_t data = Wire.read();
504     data |= uint16_t(Wire.read()) << 8;
505     uint8_t idx = data >> 12;
506     data &= 0x03FF;
507     if (idx < Config::num_faders) references[idx].setMasterSetpoint(data);
508 }
509
510 // ----- Serial ----- //
511
512 // Read SLIP messages from the serial port that allow dynamically updating the
513 // tuning of the controllers. This is used by the Python tuning script.
514 //
515 // Message format: <command> <fader> <value>
516 // Commands:
517 //   - p: proportional gain Kp
518 //   - i: integral gain Ki
519 //   - d: derivative gain Kd
520 //   - c: derivative filter cutoff frequency f_c (Hz)
521 //   - m: maximum absolute control output
522 //   - s: start an experiment, using getNextExperimentSetpoint
523 // Fader index: up to four faders are addressed using the characters '0' - '3'.
524 // Values: values are sent as 32-bit little Endian floating point numbers.
525 //
526 // For example the message 'c0\x00\x00\x20\x42' sets the derivative filter
527 // cutoff frequency of the first fader to 40.
528
529 void updateSerialIn() {
530     static SLIPParser parser;
531     static char cmd = '\0';
532     static uint8_t fader_idx = 0;
533     static uint8_t buf[4];
534     static_assert(sizeof(buf) == sizeof(float), "");
535     // This function is called if a new byte of the message arrives:
536     auto on_char_receive = [&](char new_byte, size_t index_in_packet) {
537         if (index_in_packet == 0)
538             cmd = new_byte;
539         else if (index_in_packet == 1)
540             fader_idx = new_byte - '0';
541         else if (index_in_packet < 6)
542             buf[index_in_packet - 2] = new_byte;
543     };
544     // Convert the 4-byte buffer to a float:
545     auto as_f32 = [&] {
546         float f;
547         memcpy(&f, buf, sizeof(float));
548         return f;
549     };
550     // Read and parse incoming packets from Serial:
551     while (Serial.available() > 0) {
552         uint8_t c = Serial.read();
553         auto msg_size = parser.parse(c, on_char_receive);
554         // If a complete message of 6 bytes was received, and if it addresses
555         // a valid fader:
556         if (msg_size == 6 && fader_idx < Config::num_faders) {
557             // Execute the command:
558             switch (cmd) {
559                 case 'p': controllers[fader_idx].setKp(as_f32()); break;
560                 case 'i': controllers[fader_idx].setKi(as_f32()); break;
561                 case 'd': controllers[fader_idx].setKd(as_f32()); break;
562                 case 'c': controllers[fader_idx].setEMACutoff(as_f32()); break;
563                 case 'm': controllers[fader_idx].setMaxOutput(as_f32()); break;
564                 case 's':
565                     references[fader_idx].startExperiment(as_f32());
566                     controllers[fader_idx].resetIntegral();
567                     break;

```

```
568         default: break;
569     }
570 }
571 }
572 }
```


ADC.hpp

```

1  #pragma once
2  #include "Registers.hpp"
3  #include <avr/interrupt.h>
4  #include <util/atomic.h>
5
6  #include <Arduino_Helpers.h> // EMA.hpp
7
8  #include <AH/Filters/EMA.hpp> // EMA filter
9  #include <AH/Math/IncreaseBitDepth.hpp> // increaseBitDepth
10
11 template <class Config>
12 struct ADCManager {
13     /// Evenly distribute the analog readings in the control loop period.
14     constexpr static uint8_t adc_start_count =
15         Config::interrupt_divisor / Config::num_faders;
16     /// The rate at which we're sampling using the ADC.
17     constexpr static float adc_rate = Config::interrupt_freq / adc_start_count;
18     /// Check that this doesn't take more time than the 13 ADC clock cycles it
19     /// takes to actually do the conversion. Use 14 instead of 13 just to be safe.
20     static_assert(adc_rate <= Config::adc_clock_freq / 14, "ADC too slow");
21
22     /// Enable the ADC with Vcc reference, with the given prescaler, auto
23     /// trigger disabled, ADC interrupt enabled.
24     /// Called from main program, with interrupts enabled.
25     void begin();
26
27     /// Start an ADC conversion on the given channel.
28     /// Called inside an ISR.
29     void startConversion(uint8_t channel);
30
31     /// Start an ADC conversion at the right intervals.
32     /// @param counter
33     /// Counter that keeps track of how many times the timer interrupt
34     /// fired, between 0 and Config::interrupt_divisor - 1.
35     /// Called inside an ISR.
36     void update(uint8_t counter);
37
38     /// Read the latest ADC result.
39     /// Called inside an ISR.
40     void complete();
41
42     /// Get the latest ADC reading for the given index.
43     /// Called from main program, with interrupts enabled.
44     int16_t read(uint8_t idx);
45     /// Get the latest filtered ADC reading for the given index.
46     /// Called from main program, with interrupts enabled.
47     /// @return (16 - Config::adc_ema_K)-bit filtered ADC value.
48     uint16_t readFiltered(uint8_t idx);
49     /// Get the latest filtered ADC reading for the given index.
50     /// Called from main program, with interrupts enabled.
51     /// @return 14-bit filtered ADC value.
52     uint16_t readFiltered14(uint8_t idx);
53     /// Get the latest filtered ADC reading for the given index.
54     /// Called inside an ISR.
55     /// @return 14-bit filtered ADC value.
56     uint16_t readFiltered14ISR(uint8_t idx);
57
58     /// Write the ADC reading for the given index.
59     /// Called from main program, with interrupts enabled.
60     void write(uint8_t idx, int16_t val);
61     /// Write the filtered ADC reading for the given index.
62     /// Called only before ADC interrupts are enabled.
63     /// @param val 10-bit ADC value.
64     void writeFiltered(uint8_t idx, uint16_t val);
65
66     /// Convert a 10-bit ADC reading to the largest possible range for the given
67     /// value of Config::adc_ema_K.
68     uint16_t shiftValue10(uint16_t val);
69     /// Convert the given shifted filtered value to a 14-bit range.
70     uint16_t unShiftValue14(uint16_t val);
71
72     /// Convert the channel index between 0 and Config::num_faders - 1 to the
73     /// actual ADC multiplexer address.
74     constexpr static inline uint8_t
75     channel_index_to_mux_address(uint8_t adc_mux_idx) {
76         return Config::use_A6_A7
77             ? (adc_mux_idx < 2 ? adc_mux_idx : adc_mux_idx + 4)
78             : adc_mux_idx;
79     }
80
81     /// Index of the ADC channel currently being read.
82     uint8_t channel_index = Config::num_faders;
83     /// Latest 10-bit ADC reading of each fader (updated in ADC ISR). Used for
84     /// the control loop.
85     volatile int16_t readings[Config::num_faders];
86     /// Filters for ADC readings.
87     EMA<Config::adc_ema_K, uint16_t> filters[Config::num_faders];
88     /// Filtered (shifted) ADC readings. Used to output over MIDI etc. but not
89     /// for the control loop.
90     volatile uint16_t filtered_readings[Config::num_faders];
91 };
92

```

```

93 template <class Config>
94 inline void ADCManager<Config>::begin() {
95     constexpr auto prescaler = factorToADCPrescaler(Config::adc_prescaler_fac);
96     static_assert(prescaler != ADCPrescaler::Invalid, "Invalid prescaler");
97
98     ATOMIC_BLOCK(ATOMIC_FORCEON) {
99         cbi(ADCSRA, ADEN); // Disable ADC
100
101         cbi(ADMUX, REFS1); // Vcc reference
102         sbi(ADMUX, REFS0); // Vcc reference
103
104         cbi(ADMUX, ADLAR); // 8 least significant bits in ADCL
105
106         setADCPrescaler(prescaler);
107
108         cbi(ADCSRA, ADSC); // Auto trigger disable
109         sbi(ADCSRA, ADIFSC); // ADC Interrupt Enable
110         sbi(ADCSRA, ADIFR); // Enable ADC
111     }
112 }
113
114 template <class Config>
115 inline void ADCManager<Config>::update(uint8_t counter) {
116     if (Config::num_faders > 0 && counter == 0 * adc_start_count)
117         startConversion(0);
118     else if (Config::num_faders > 1 && counter == 1 * adc_start_count)
119         startConversion(1);
120     else if (Config::num_faders > 2 && counter == 2 * adc_start_count)
121         startConversion(2);
122     else if (Config::num_faders > 3 && counter == 3 * adc_start_count)
123         startConversion(3);
124 }
125
126 template <class Config>
127 inline void ADCManager<Config>::startConversion(uint8_t channel) {
128     channel_index = channel;
129     ADMUX &= 0xF0;
130     ADMUX |= channel_index_to_mux_address(channel);
131     sbi(ADCSRA, ADSC); // ADC Start Conversion
132 }
133
134 template <class Config>
135 inline void ADCManager<Config>::complete() {
136     if (Config::enable_overrun_indicator && readings[channel_index] >= 0)
137         sbi(PORTB, 5); // Set overrun indicator
138     uint16_t value = ADC; // Store ADC reading
139     readings[channel_index] = value;
140     // Filter the reading
141     auto &filter = filters[channel_index];
142     filtered_readings[channel_index] = filter(shiftValue10(value));
143 }
144
145 template <class Config>
146 inline int16_t ADCManager<Config>::read(uint8_t idx) {
147     int16_t val;
148     ATOMIC_BLOCK(ATOMIC_FORCEON) { val = readings[idx]; }
149     return val;
150 }
151
152 template <class Config>
153 inline void ADCManager<Config>::write(uint8_t idx, int16_t val) {
154     ATOMIC_BLOCK(ATOMIC_FORCEON) { readings[idx] = val; }
155 }
156
157 template <class Config>
158 inline uint16_t ADCManager<Config>::shiftValue10(uint16_t val) {
159     return AH::increaseBitDepth<16 - Config::adc_ema_K, 10, uint16_t>(val);
160 }
161
162 template <class Config>
163 inline uint16_t ADCManager<Config>::unShiftValue14(uint16_t val) {
164     const int shift = 6 - Config::adc_ema_K - 4;
165     return shift >= 0 ? val >> shift : val << -shift;
166 }
167
168 template <class Config>
169 inline uint16_t ADCManager<Config>::readFiltered14ISR(uint8_t idx) {
170     return unShiftValue14(filtered_readings[idx]);
171 }
172
173 template <class Config>
174 inline uint16_t ADCManager<Config>::readFiltered(uint8_t idx) {
175     uint16_t val;
176     ATOMIC_BLOCK(ATOMIC_FORCEON) { val = filtered_readings[idx]; }
177     return val;
178 }
179
180 template <class Config>
181 inline uint16_t ADCManager<Config>::readFiltered14(uint8_t idx) {
182     return unShiftValue14(readFiltered(idx));
183 }
184
185 template <class Config>
186 inline void ADCManager<Config>::writeFiltered(uint8_t idx, uint16_t val) {
187     filters[idx].reset(shiftValue10(val));

```

```
188     filtered_readings[idx] = shiftValue10(val);  
189 }
```

Touch.hpp

```
1 #pragma once
2 #include <avr/interrupt.h>
3 #include <avr/io.h>
4 #include <util/atomic.h>
5
6 template <class Config>
7 struct TouchSense {
8
9     /// The actual threshold as a number of interrupts instead of seconds:
10     static constexpr uint8_t touch_sense_thres =
11         Config::interrupt_freq * Config::touch_rc_time_threshold;
12     /// Ignore mains noise by making the "touched" status stick for longer than
13     /// the mains period:
14     static constexpr float period_50Hz = 1. / 50;
15     /// Keep the "touched" status active for this many periods (see below):
16     static constexpr uint8_t touch_sense_stickiness =
17         Config::interrupt_freq * period_50Hz * 4 / Config::interrupt_divisor;
18     /// Check that the threshold is smaller than the control loop period:
19     static_assert(touch_sense_thres < Config::interrupt_divisor,
20         "Touch sense threshold too high");
21
22     /// The combined bit mask for all touch GPIO pins.
23     static constexpr uint8_t gpio_mask =
24         (Config::num_faders > 0 ? Config::touch_masks[0] : 0) |
25         (Config::num_faders > 1 ? Config::touch_masks[1] : 0) |
26         (Config::num_faders > 2 ? Config::touch_masks[2] : 0) |
27         (Config::num_faders > 3 ? Config::touch_masks[3] : 0);
28
29     /// Initialize the GPIO pins for capacitive sensing.
30     /// Called from main program, with interrupts enabled.
31     void begin();
32
33     /// Check which touch sensing knobs are being touched.
34     /// @param counter
35     ///     Counter that keeps track of how many times the timer interrupt
36     ///     fired, between 0 and Config::interrupt_divisor - 1.
37     /// Called inside an ISR.
38     void update(uint8_t counter);
39
40     /// Get the touch status for the given index.
41     /// Called from main program, with interrupts enabled.
42     bool read(uint8_t idx);
43
44     /// Timers to take into account the stickiness.
45     uint8_t touch_timers[Config::num_faders] {};
46     /// Whether the knobs are being touched.
47     volatile bool touched[Config::num_faders] {};
48 };
49
50 template <class Config>
51 void TouchSense<Config>::begin() {
52     ATOMIC_BLOCK(ATOMIC_FORCEON) {
53         PORTB &= ~gpio_mask; // low
54         DDRB |= gpio_mask;    // output mode
55     }
56 }
57
58 // 0. The pin mode is "output", the value is "low".
59 // 1. Set the pin mode to "input", touch_timer = 0.
60 // 2. The pin will start charging through the external pull-up resistor.
61 // 3. After a fixed amount of time, check whether the pin became "high":
62 //    if this is the case, the RC-time of the knob/pull-up resistor circuit
63 //    was smaller than the given threshold. Since R is fixed, this can be used
64 //    to infer C, the capacitance of the knob: if the capacitance is lower than
65 //    the threshold (i.e. RC-time is lower), this means the knob was not touched.
66 // 5. Set the pin mode to "output", to start discharging the pin to 0V again.
67 // 6. Some time later, the pin has discharged, so switch to "input" mode and
68 //    start charging again for the next RC-time measurement.
69 //
70 // The "touched" status is sticky: it will remain set for at least
71 // touch_sense_stickiness ticks. If the pin never resulted in another "touched"
72 // measurement during that period, the "touched" status for that pin is cleared.
73
74 template <class Config>
75 void TouchSense<Config>::update(uint8_t counter) {
76     if (gpio_mask == 0)
77         return;
78     if (counter == 0) {
79         DDRB &= ~gpio_mask; // input mode, start charging
80     } else if (counter == touch_sense_thres) {
81         uint8_t touched_bits = PINB;
82         DDRB |= gpio_mask; // output mode, start discharging
83         for (uint8_t i = 0; i < Config::num_faders; ++i) {
84             if (Config::touch_masks[i] == 0)
85                 continue;
86             bool touch_i = (touched_bits & Config::touch_masks[i]) == 0;
87             if (touch_i) {
88                 touch_timers[i] = touch_sense_stickiness;
89                 touched[i] = true;
90             } else if (touch_timers[i] > 0) {
91                 --touch_timers[i];
92                 if (touch_timers[i] == 0) touched[i] = false;
93             }
94         }
95     }
96 }
```

```
93     }
94   }
95 }
96
97
98 template <class Config>
99 bool TouchSense<Config>::read(uint8_t idx) {
100     bool t;
101     ATOMIC_BLOCK(ATOMIC_FORCEON) { t = touched[idx]; }
102     return t;
103 }
```

Controller.hpp

```
1  #pragma once
2
3  #include <stdint.h>
4  #include <stdint.h>
5
6  /// @see @ref horner(float,float,const float(&)[N])
7  constexpr inline float horner_impl(float xa, const float *p, size_t count,
8                                     float t) {
9      return count == 0 ? p[count] + xa * t
10         : horner_impl(xa, p, count - 1, p[count] + xa * t);
11 }
12
13 /// Evaluate a polynomial using
14 /// [Horner's method](https://en.wikipedia.org/wiki/Horner%27s_method).
15 template <size_t N>
16 constexpr inline float horner(float x, float a, const float (&p)[N]) {
17     return horner_impl(x - a, p, N - 2, p[N - 1]);
18 }
19
20 /// Compute the weight factor of a exponential moving average filter
21 /// with the given cutoff frequency.
22 /// @see https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-
23 filters/Exponential%20Moving%20Average/Exponential-Moving-Average.html#cutoff-frequency
24 /// for the formula.
25 inline float calcAlphaEMA(float f_n) {
26     // Taylor coefficients of
27     //  $\alpha(f_n) = \cos(2\pi f_n) - 1 + \sqrt{\cos(2\pi f_n)^2 - 4 \cos(2\pi f_n) + 3}$ 
28     // at  $f_n = 0.25$ 
29     constexpr static float coeff[] {
30         +7.3205080756887730e-01, +9.7201214975728490e-01,
31         -3.7988125051760377e+00, +9.5168450173968860e+00,
32         -2.0829320344443730e+01, +3.0074306603814595e+01,
33         -1.6446172139457754e+01, -8.0756002564633450e+01,
34         +3.2420501524111750e+02, -6.5601870948443250e+02,
35     };
36     return horner(f_n, 0.25, coeff);
37 }
38
39 /// Standard PID (proportional, integral, derivative) controller. Derivative
40 /// component is filtered using an exponential moving average filter.
41 class PID {
42 public:
43     PID() = default;
44     /// @param kp
45     /// Proportional gain
46     /// @param ki
47     /// Integral gain
48     /// @param kd
49     /// Derivative gain
50     /// @param Ts
51     /// Sampling time (seconds)
52     /// @param fc
53     /// Cutoff frequency of derivative EMA filter (Hertz),
54     /// zero to disable the filter entirely
55     PID(float kp, float ki, float kd, float Ts, float f_c = 0,
56         float maxOutput = 255)
57         : Ts(Ts), maxOutput(maxOutput) {
58         setKp(kp);
59         setKi(ki);
60         setKd(kd);
61         setEMACutoff(f_c);
62     }
63
64     /// Update the controller: given the current position, compute the control
65     /// action.
66     float update(uint16_t input) {
67         // The error is the difference between the reference (setpoint) and the
68         // actual position (input)
69         int16_t error = setpoint - input;
70         // The integral or sum of current and previous errors
71         int32_t newIntegral = integral + error;
72         // Compute the difference between the current and the previous input,
73         // but compute a weighted average using a factor  $\alpha \in (0,1]$ 
74         float diff = emaAlpha * (prevInput - input);
75         // Update the average
76         prevInput -= diff;
77
78         // Check if we can turn off the motor
79         if (activityCount >= activityThres && activityThres) {
80             float filtError = setpoint - prevInput;
81             if (filtError >= -errThres && filtError <= errThres) {
82                 errThres = 2; // hysteresis
83                 return 0;
84             } else {
85                 errThres = 1;
86             }
87         } else {
88             ++activityCount;
89             errThres = 1;
90         }
91
92         bool backward = false;
```

```

92     int32_t calcIntegral = backward ? newIntegral : integral;
93
94     // Standard PID rule
95     float output = kp * error + ki_Ts * calcIntegral + kd_Ts * diff;
96
97     // Clamp and anti-windup
98     if (output > maxOutput)
99         output = maxOutput;
100     else if (output < -maxOutput)
101         output = -maxOutput;
102     else
103         integral = newIntegral;
104
105     return output;
106 }
107
108 void setKp(float kp) { this->kp = kp; }          ///< Proportional gain
109 void setKi(float ki) { this->ki_Ts = ki * this->Ts; } ///< Integral gain
110 void setKd(float kd) { this->kd_Ts = kd / this->Ts; } ///< Derivative gain
111
112 float getKp() const { return kp; }              ///< Proportional gain
113 float getKi() const { return ki_Ts / Ts; }       ///< Integral gain
114 float getKd() const { return kd_Ts * Ts; }       ///< Derivative gain
115
116 // Set the cutoff frequency (-3 dB point) of the exponential moving average
117 // filter that is applied to the input before taking the difference for
118 // computing the derivative term.
119 void setEMACutoff(float f_c) {
120     float f_n = f_c * Ts; // normalized sampling frequency
121     this->emaAlpha = f_c == 0 ? 1 : calcAlphaEMA(f_n);
122 }
123
124 // Set the reference/target/setpoint of the controller.
125 void setSetpoint(uint16_t setpoint) {
126     if (this->setpoint != setpoint) this->activityCount = 0;
127     this->setpoint = setpoint;
128 }
129 // @see @ref setSetpoint(int16_t)
130 uint16_t getSetpoint() const { return setpoint; }
131
132 // Set the maximum control output magnitude. Default is 255, which clamps
133 // the control output in [-255, +255].
134 void setMaxOutput(float maxOutput) { this->maxOutput = maxOutput; }
135 // @see @ref setMaxOutput(float)
136 float getMaxOutput() const { return maxOutput; }
137
138 // Reset the activity counter to prevent the motor from turning off.
139 void resetActivityCounter() { this->activityCount = 0; }
140 // Set the number of seconds after which the motor is turned off, zero to
141 // keep it on indefinitely.
142 void setActivityTimeout(float s) {
143     if (s == 0)
144         activityThres = 0;
145     else
146         activityThres = uint16_t(s / Ts) == 0 ? 1 : s / Ts;
147 }
148
149 // Reset the sum of the previous errors to zero.
150 void resetIntegral() { integral = 0; }
151
152 private:
153     float Ts = 1;          ///< Sampling time (seconds)
154     float maxOutput = 255; ///< Maximum control output magnitude
155     float kp = 1;          ///< Proportional gain
156     float ki_Ts = 0;       ///< Integral gain times Ts
157     float kd_Ts = 0;       ///< Derivative gain divided by Ts
158     float emaAlpha = 1;    ///< Weight factor of derivative EMA filter.
159     float prevInput = 0;   ///< (Filtered) previous input for derivative.
160     uint16_t activityCount = 0; ///< How many ticks since last setpoint change.
161     uint16_t activityThres = 0; ///< Threshold for turning off the output.
162     uint8_t errThres = 1;   ///< Threshold with hysteresis.
163     int32_t integral = 0;   ///< Sum of previous errors for integral.
164     uint16_t setpoint = 0;  ///< Position reference.
165 };

```

Motor.hpp

```

1  #pragma once
2  #include "Registers.hpp"
3  #include <avr/interrupt.h>
4  #include <util/atomic.h>
5
6  /// Configure Timer0 in either phase correct or fast PWM mode with the given
7  /// prescaler, enable output compare B.
8  inline void setupMotorTimer0(bool phase_correct, Timer0Prescaler prescaler) {
9      ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
10         setTimer0WGMode(phase_correct ? Timer0WGMode::PWM
11                               : Timer0WGMode::FastPWM);
12         setTimer0Prescaler(prescaler);
13         sbi(TCCR0A, COM0B1); // Table 14-6, 14-7 Compare Output Mode
14         sbi(TCCR0A, COM0A1); // Table 14-6, 14-7 Compare Output Mode
15     }
16 }
17
18 /// Configure Timer2 in either phase correct or fast PWM mode with the given
19 /// prescaler, enable output compare B.
20 inline void setupMotorTimer2(bool phase_correct, Timer2Prescaler prescaler) {
21     ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
22         setTimer2WGMode(phase_correct ? Timer2WGMode::PWM
23                               : Timer2WGMode::FastPWM);
24         setTimer2Prescaler(prescaler);
25         sbi(TCCR2A, COM2B1); // Table 14-6, 14-7 Compare Output Mode
26         sbi(TCCR2A, COM2A1); // Table 14-6, 14-7 Compare Output Mode
27     }
28 }
29
30 /// Configure the timers for the PWM outputs.
31 template <class Config>
32 inline void setupMotorTimers() {
33     constexpr auto prescaler0 = factorToTimer0Prescaler(Config::prescaler_fac);
34     static_assert(prescaler0 != Timer0Prescaler::Invalid, "Invalid prescaler");
35     constexpr auto prescaler2 = factorToTimer2Prescaler(Config::prescaler_fac);
36     static_assert(prescaler2 != Timer2Prescaler::Invalid, "Invalid prescaler");
37
38     if (Config::num_faders > 0)
39         setupMotorTimer2(Config::phase_correct_pwm, prescaler2);
40     if (Config::num_faders > 2)
41         setupMotorTimer0(Config::phase_correct_pwm, prescaler0);
42 }
43
44 /// Class for driving up to 4 DC motors using PWM.
45 template <class Config>
46 struct Motors {
47     void begin();
48     template <uint8_t Idx>
49     void setSpeed(int16_t speed);
50
51     template <uint8_t Idx>
52     void setupGPIO();
53     template <uint8_t Idx>
54     void forward(uint8_t speed);
55     template <uint8_t Idx>
56     void backward(uint8_t speed);
57 };
58
59 template <class Config>
60 inline void Motors<Config>::begin() {
61     setupMotorTimers<Config>();
62
63     if (Config::num_faders > 0) {
64         sbi(DDRD, 2);
65         sbi(DDRD, 3);
66     }
67     if (Config::num_faders > 1) {
68         sbi(DDRD, 7);
69         sbi(DDRB, 3);
70     }
71     if (Config::num_faders > 2) {
72         sbi(DDRD, 4);
73         sbi(DDRD, 5);
74     }
75     if (Config::num_faders > 3) {
76         if (Config::fader_3_A2)
77             sbi(DDRC, 2);
78         else
79             sbi(DDRB, 5);
80
81         sbi(DDRD, 6);
82     }
83 }
84
85 // Fast PWM (Table 14-6):
86 //   Clear OC0B on Compare Match, set OC0B at BOTTOM (non-inverting mode).
87 // Phase Correct PWM (Table 14-7):
88 //   Clear OC0B on compare match when up-counting. Set OC0B on compare match
89 //   when down-counting.
90 template <class Config>
91 template <uint8_t Idx>
92 inline void Motors<Config>::forward(uint8_t speed) {

```



```

93     if (Idx >= Config::num_faders)
94         return;
95     else if (Idx == 0)
96         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
97             cbi(TCCR2A, COM2B0);
98             cbi(PORTD, 2);
99             OCR2B = speed;
100         }
101     else if (Idx == 1)
102         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
103             cbi(TCCR2A, COM2A0);
104             cbi(PORTD, 7);
105             OCR2A = speed;
106         }
107     else if (Idx == 2)
108         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
109             cbi(TCCR0A, COM0B0);
110             cbi(PORTD, 4);
111             OCR0B = speed;
112         }
113     else if (Idx == 3)
114         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
115             cbi(TCCR0A, COM0A0);
116             if (Config::fader_3_A2)
117                 cbi(PORTC, 2);
118             else
119                 cbi(PORTB, 5);
120             OCR0A = speed;
121         }
122 }
123
124 // Fast PWM (Table 14-6):
125 // Set OC0B on Compare Match, clear OC0B at BOTTOM (inverting mode).
126 // Phase Correct PWM (Table 14-7):
127 // Set OC0B on compare match when up-counting. Clear OC0B on compare match
128 // when down-counting.
129 template <class Config>
130 template <uint8_t Idx>
131 inline void Motors<Config>::backward(uint8_t speed) {
132     if (Idx >= Config::num_faders)
133         return;
134     else if (Idx == 0)
135         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
136             sbi(TCCR2A, COM2B0);
137             sbi(PORTD, 2);
138             OCR2B = speed;
139         }
140     else if (Idx == 1)
141         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
142             sbi(TCCR2A, COM2A0);
143             sbi(PORTD, 7);
144             OCR2A = speed;
145         }
146     else if (Idx == 2)
147         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
148             sbi(TCCR0A, COM0B0);
149             sbi(PORTD, 4);
150             OCR0B = speed;
151         }
152     else if (Idx == 3)
153         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
154             sbi(TCCR0A, COM0A0);
155             if (Config::fader_3_A2)
156                 sbi(PORTC, 2);
157             else
158                 sbi(PORTB, 5);
159             OCR0A = speed;
160         }
161 }
162
163 template <class Config>
164 template <uint8_t Idx>
165 inline void Motors<Config>::setSpeed(int16_t speed) {
166     if (speed >= 0)
167         forward<Idx>(speed);
168     else
169         backward<Idx>(-speed);
170 }

```

Reference.hpp

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

```

93 template <class T, size_t N>
94 constexpr size_t len(const T (&)[N]) {
95     return N;
96 }
97
98 /// Class that handles the three main references/setpoints:
99 /// 1. Sequence programmed in PROGMEM
100 /// 2. Test sequence for tuning experiments (activated over Serial)
101 /// 3. Setpoint set by the I2C master (for real-world use)
102 template <class Config>
103 class Reference {
104 public:
105     /// Called from ISR
106     void setMasterSetpoint(uint16_t setpoint) {
107         this->master_setpoint = setpoint;
108     }
109
110     void startExperiment(float speed_div) {
111         this->experiment_speed_div = speed_div;
112         this->index = 0;
113         this->seq_idx = 0;
114     }
115
116     bool experimentInProgress() const { return experiment_speed_div > 0; }
117
118     uint16_t getNextProgmemSetpoint() {
119         uint16_t setpoint = pgm_read_byte(reference_signal + index) * 4;
120         ++seq_idx;
121         if (seq_idx >= Config::test_reference_speed_div) {
122             seq_idx = 0;
123             ++index;
124             if (index == len(reference_signal)) index = 0;
125         }
126         return setpoint;
127     }
128
129     uint16_t getNextExperimentSetpoint() {
130         constexpr uint16_t RAMPUP = 0xFFFF;
131         auto rampup = [](uint16_t idx, uint16_t duration) {
132             return uint32_t(1024) * idx / duration;
133         };
134         constexpr uint16_t RAMPDOWN = 0xFFFE;
135         auto rampdown = [&](uint16_t idx, uint16_t duration) {
136             return 1023 - rampup(idx, duration);
137         };
138         struct TestSeq {
139             uint16_t setpoint;
140             uint16_t duration;
141         };
142         /// This array defines the test sequence
143         constexpr static TestSeq seqs[] {
144             {0, 256}, {RAMPUP, 128}, {1023, 32}, {0, 64}, {333, 32},
145             {666, 32}, {333, 32}, {0, 32}, {512, 256},
146         };
147
148         static uint8_t seq_index = 0;
149         static uint16_t index = 0;
150         uint16_t duration = seqs[seq_index].duration * experiment_speed_div;
151         uint16_t seq_setpoint = seqs[seq_index].setpoint;
152         uint16_t setpoint;
153         switch (seq_setpoint) {
154             case RAMPUP: setpoint = rampup(index, duration); break;
155             case RAMPDOWN: setpoint = rampdown(index, duration); break;
156             default: setpoint = seq_setpoint;
157         }
158         ++index;
159         if (index == duration) {
160             index = 0;
161             ++seq_index;
162             if (seq_index == len(seqs)) {
163                 seq_index = 0;
164                 experiment_speed_div = 0;
165             }
166         }
167         return setpoint;
168     }
169
170     /// Called from main program with interrupts enabled
171     uint16_t getNextSetpoint() {
172         uint16_t setpoint;
173         if (Config::serial_control && experiment_speed_div > 0)
174             /// from the tuning experiment reference
175             setpoint = getNextExperimentSetpoint();
176         else if (Config::test_reference)
177             /// from the test reference
178             setpoint = getNextProgmemSetpoint();
179         else
180             /// from the I2C master
181             ATOMIC_BLOCK(ATOMIC_FORCEON) { setpoint = master_setpoint; }
182         return setpoint;
183     }
184
185 private:
186     uint16_t index = 0;
187     uint8_t seq_idx = 0;

```

```
188     float experiment_speed_div = 0;
189     volatile uint16_t master_setpoint = 0;
190 };
```

Registers.hpp

```
1 #pragma once
2
3 #include <avr/io.h>
4 #include <avr/sfr_defs.h>
5 #include <util/delay.h> // F_CPU
6
7 // ----- Utils ----- //
8
9 #ifndef ARDUINO // Ensures that my IDE sees the correct frequency
10 #undef F_CPU
11 #define F_CPU 16000000UL
12 #endif
13
14 #ifndef sbi
15 /// Set bit in register.
16 template <class R>
17 inline void sbi(R &reg, uint8_t bit) {
18     reg |= (1u << bit);
19 }
20 #define sbi sbi
21 #endif
22 #ifndef cbi
23 /// Clear bit in register.
24 template <class R>
25 inline void cbi(R &reg, uint8_t bit) {
26     reg &= ~(1u << bit);
27 }
28 #define cbi cbi
29 #endif
30 /// Write bit in register.
31 template <class R>
32 inline void wbi(R &reg, uint8_t bit, bool value) {
33     value ? sbi(reg, bit) : cbi(reg, bit);
34 }
35
36 // ----- Timer0 ----- //
37
38 /// Timer 0 clock select (Table 14-9).
39 enum class Timer0Prescaler : uint8_t {
40     None = 0b000,
41     S1 = 0b001,
42     S8 = 0b010,
43     S64 = 0b011,
44     S256 = 0b100,
45     S1024 = 0b101,
46     ExtFall = 0b110,
47     ExtRise = 0b111,
48     Invalid = 0xFF,
49 };
50
51 /// Timer 0 waveform generation mode (Table 14-8).
52 enum class Timer0WGMMode : uint8_t {
53     Normal = 0b000,
54     PWM = 0b001,
55     CTC = 0b010,
56     FastPWM = 0b011,
57     PWM_OCRA = 0b101,
58     FastPWM_OCRA = 0b111,
59 };
60
61 /// Convert the prescaler factor to the correct bit pattern to write to the
62 /// TCCR0B register (Table 14-9).
63 constexpr inline Timer0Prescaler factorToTimer0Prescaler(uint16_t factor) {
64     return factor == 1 ? Timer0Prescaler::S1
65         : factor == 8 ? Timer0Prescaler::S8
66         : factor == 64 ? Timer0Prescaler::S64
67         : factor == 256 ? Timer0Prescaler::S256
68         : factor == 1024 ? Timer0Prescaler::S1024
69         : Timer0Prescaler::Invalid;
70 }
71
72 /// Set the clock source/prescaler of Timer0 (Table 14-9).
73 inline void setTimer0Prescaler(Timer0Prescaler ps) {
74     if (ps == Timer0Prescaler::Invalid)
75         return;
76     wbi(TCCR0B, CS02, static_cast<uint8_t>(ps) & (1u << 2));
77     wbi(TCCR0B, CS01, static_cast<uint8_t>(ps) & (1u << 1));
78     wbi(TCCR0B, CS00, static_cast<uint8_t>(ps) & (1u << 0));
79 }
80
81 /// Set the waveform generation mode of Timer0 (Table 14-8).
82 inline void setTimer0WGMMode(Timer0WGMMode mode) {
83     wbi(TCCR0B, WGM02, static_cast<uint8_t>(mode) & (1u << 2));
84     wbi(TCCR0A, WGM01, static_cast<uint8_t>(mode) & (1u << 1));
85     wbi(TCCR0A, WGM00, static_cast<uint8_t>(mode) & (1u << 0));
86 }
87
88 // ----- Timer2 ----- //
89
90 /// Timer 0 clock select (Table 17-9).
91 enum class Timer2Prescaler : uint8_t {
92     None = 0b000,
```

```

93     S1 = 0b001,
94     S8 = 0b010,
95     S32 = 0b011,
96     S64 = 0b100,
97     S128 = 0b101,
98     S256 = 0b110,
99     S1024 = 0b111,
100    Invalid = 0xFF,
101 };
102
103 /// Timer 0 waveform generation mode (Table 17-8).
104 enum class Timer2WGMode : uint8_t {
105     Normal = 0b000,
106     PWM = 0b001,
107     CTC = 0b010,
108     FastPWM = 0b011,
109     PWM_OCRA = 0b101,
110     FastPWM_OCRA = 0b111,
111 };
112
113 /// Convert the prescaler factor to the correct bit pattern to write to the
114 /// TCCR0B register (Table 17-9).
115 constexpr inline Timer2Prescaler factorToTimer2Prescaler(uint16_t factor) {
116     return factor == 1      ? Timer2Prescaler::S1
117        : factor == 8      ? Timer2Prescaler::S8
118        : factor == 32     ? Timer2Prescaler::S32
119        : factor == 64     ? Timer2Prescaler::S64
120        : factor == 128    ? Timer2Prescaler::S128
121        : factor == 256    ? Timer2Prescaler::S256
122        : factor == 1024   ? Timer2Prescaler::S1024
123        : Timer2Prescaler::Invalid;
124 }
125
126 /// Set the clock source/prescaler of Timer2 (Table 17-9).
127 inline void setTimer2Prescaler(Timer2Prescaler ps) {
128     if (ps == Timer2Prescaler::Invalid)
129         return;
130     wbi(TCCR2B, CS22, static_cast<uint8_t>(ps) & (1u << 2));
131     wbi(TCCR2B, CS21, static_cast<uint8_t>(ps) & (1u << 1));
132     wbi(TCCR2B, CS20, static_cast<uint8_t>(ps) & (1u << 0));
133 }
134
135 /// Set the waveform generation mode of Timer2 (Table 17-8).
136 inline void setTimer2WGMode(Timer2WGMode mode) {
137     wbi(TCCR2B, WGM22, static_cast<uint8_t>(mode) & (1u << 2));
138     wbi(TCCR2A, WGM21, static_cast<uint8_t>(mode) & (1u << 1));
139     wbi(TCCR2A, WGM20, static_cast<uint8_t>(mode) & (1u << 0));
140 }
141
142 // ----- ADC ----- //
143
144 /// ADC prescaler select (Table 23-5).
145 enum class ADCPrescaler : uint8_t {
146     S2 = 0b000,
147     S2_2 = 0b001,
148     S4 = 0b010,
149     S8 = 0b011,
150     S16 = 0b100,
151     S32 = 0b101,
152     S64 = 0b110,
153     S128 = 0b111,
154     Invalid = 0xFF,
155 };
156
157 /// Convert the prescaler factor to the correct bit pattern to write to the
158 /// ADCSRA register (Table 23-5).
159 constexpr inline ADCPrescaler factorToADCPrescaler(uint8_t factor) {
160     return factor == 2      ? ADCPrescaler::S2_2
161        : factor == 4      ? ADCPrescaler::S4
162        : factor == 8      ? ADCPrescaler::S8
163        : factor == 16     ? ADCPrescaler::S16
164        : factor == 32     ? ADCPrescaler::S32
165        : factor == 64     ? ADCPrescaler::S64
166        : factor == 128    ? ADCPrescaler::S128
167        : ADCPrescaler::Invalid;
168 }
169
170 /// Set the prescaler of the ADC (Table 23-5).
171 inline void setADCPrescaler(ADCPrescaler ps) {
172     if (ps == ADCPrescaler::Invalid)
173         return;
174     wbi(ADCSRA, ADPS2, static_cast<uint8_t>(ps) & (1u << 2));
175     wbi(ADCSRA, ADPS1, static_cast<uint8_t>(ps) & (1u << 1));
176     wbi(ADCSRA, ADPS0, static_cast<uint8_t>(ps) & (1u << 0));
177 }

```

SerialSLIP.hpp

```
1 #include <Arduino.h>
2
3 namespace SLIP_Constants {
4     const static uint8_t END = 0300;
5     const static uint8_t ESC = 0333;
6     const static uint8_t ESC_END = 0334;
7     const static uint8_t ESC_ESC = 0335;
8 } // namespace SLIP_Constants
9
10 /// Parses SLIP packets: https://datatracker.ietf.org/doc/html/rfc1055
11 class SLIPParser {
12 public:
13     template <class Callback>
14     size_t parse(uint8_t c, Callback callback);
15
16     void reset() {
17         size = 0;
18         escape = false;
19     }
20
21 private:
22     size_t size = 0;
23     bool escape = false;
24 };
25
26 template <class Callback>
27 size_t SLIPParser::parse(uint8_t c, Callback callback) {
28     using namespace SLIP_Constants;
29     /*
30      * handle bytestuffing if necessary
31      */
32     switch (c) {
33         /*
34          * if it's an END character then we're done with
35          * the packet
36          */
37         case END: {
38             /*
39              * a minor optimization: if there is no
40              * data in the packet, ignore it. This is
41              * meant to avoid bothering IP with all
42              * the empty packets generated by the
43              * duplicate END characters which are in
44              * turn sent to try to detect line noise.
45              */
46             auto packetLen = size;
47             reset();
48             if (packetLen) return packetLen;
49         } break;
50
51         /*
52          * if it's the same code as an ESC character, wait
53          * and get another character and then figure out
54          * what to store in the packet based on that.
55          */
56         case ESC: {
57             escape = true;
58         } break;
59
60         /*
61          * here we fall into the default handler and let
62          * it store the character for us
63          */
64         default: {
65             if (escape) {
66                 /*
67                  * if "c" is not one of these two, then we
68                  * have a protocol violation. The best bet
69                  * seems to be to leave the byte alone and
70                  * just stuff it into the packet
71                  */
72                 switch (c) {
73                     case ESC_END: c = END; break;
74                     case ESC_ESC: c = ESC; break;
75                     default: break; // LCOV_EXCL_LINE (protocol violation)
76                 }
77                 escape = false;
78             }
79             callback(c, size);
80             ++size;
81         }
82     }
83     return 0;
84 }
85
86 /// Sends SLIP packets: https://datatracker.ietf.org/doc/html/rfc1055
87 class SLIPSender {
88 public:
89     SLIPSender(Stream &stream) : stream(stream) {}
90
91     size_t beginPacket() { return stream.write(SLIP_Constants::END); }
92     size_t endPacket() { return stream.write(SLIP_Constants::END); }
```

```

93     size_t write(const uint8_t *data, size_t len);
94     size_t writePacket(const uint8_t *data, size_t len) {
95         size_t sent = 0;
96         sent += beginPacket();
97         sent += write(data, len);
98         sent += endPacket();
99         return sent;
100     }
101 }
102
103 private:
104     Stream &stream;
105 };
106
107 inline size_t SLIPSender::write(const uint8_t *data, size_t len) {
108     // https://datatracker.ietf.org/doc/html/rfc1055
109     using namespace SLIP_Constants;
110     size_t sent = 0;
111     /*
112      * for each byte in the packet, send the appropriate character
113      * sequence
114      */
115     while (len--) {
116         switch (*data) {
117             /*
118              * if it's the same code as an END character, we send a
119              * special two character code so as not to make the
120              * receiver think we sent an END
121              */
122             case END:
123                 sent += stream.write(ESC);
124                 sent += stream.write(ESC_END);
125                 break;
126
127             /*
128              * if it's the same code as an ESC character,
129              * we send a special two character code so as not
130              * to make the receiver think we sent an ESC
131              */
132             case ESC:
133                 sent += stream.write(ESC);
134                 sent += stream.write(ESC_ESC);
135                 break;
136
137             /*
138              * otherwise, we just send the character
139              */
140             default: sent += stream.write(*data);
141         }
142         data++;
143     }
144     return sent;
145 }

```


SMA.hpp

```

1  #pragma once
2
3  #include <Arduino_Helpers.h>
4
5  #include <AH/Math/Divide.hpp>
6  #include <AH/STL/algorithm> // std::fill
7  #include <AH/STL/cstdint>
8
9  /**
10 * @brief Simple Moving Average filter.
11 *
12 * Returns the average of the N most recent input values.
13 *
14 * @f[
15 * y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]
16 * @f]
17 *
18 * @see https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Simple%20Moving%20Average/Simple-Moving-Average.html
19 *
20 * @tparam N
21 * The number of samples to average.
22 * @tparam input_t
23 * The type of the input (and output) of the filter.
24 * @tparam sum_t
25 * The type to use for the accumulator, must be large enough to fit
26 * N times the maximum input value.
27 */
28 template <uint8_t N, class input_t = uint16_t, class sum_t = uint32_t>
29 class SMA {
30 public:
31     /// Default constructor (initial state is initialized to all zeros).
32     SMA() = default;
33
34     /// Constructor (initial state is initialized to given value).
35     ///
36     /// @param initialValue
37     /// Determines the initial state of the filter:
38     /// @f$ x[-N] = \ldots = x[-2] = x[-1] = \text{initialValue} $ @f$
39     ///
40     SMA(input_t initialValue) : sum(N * (sum_t)initialValue) {
41         std::fill(std::begin(previousInputs), std::end(previousInputs),
42             initialValue);
43     }
44
45     /// Update the internal state with the new input @f$ x[n] $ @f$ and return the
46     /// new output @f$ y[n] $ @f$.
47     ///
48     /// @param input
49     /// The new input @f$ x[n] $ @f$.
50     /// @return The new output @f$ y[n] $ @f$.
51     input_t operator()(input_t input) {
52         sum -= previousInputs[index];
53         sum += input;
54         previousInputs[index] = input;
55         if (++index == N) index = 0;
56         return AH::round_div<N>(sum);
57     }
58
59 private:
60     uint8_t index = 0;
61     input_t previousInputs[N] {};
62     sum_t sum = 0;
63 };

```