

# 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 // - D13: input 3A of L293D dual H-bridge 1 (PB5)
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 // - D7: input 3A of L293D dual H-bridge 2 (PD7)
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. A3 or A4 on an Arduino Nano).
86 // The overrun indicator is only enabled if the number of faders is 1, because
87 // it conflicts with the motor driver pin of Fader 1. You can choose a different
88 // pin instead.
89
90 // ----- Configuration ----- //
91
92 struct Config {
93 // Print the control loop and interrupt frequencies to Serial at startup:
94 static constexpr bool print_frequencies = true;
95 // Print the setpoint, actual position and control signal to Serial.
96 // Note that this slows down the control loop significantly, it probably
97 // won't work if you are using more than one fader without increasing
98 // `interrupt_divisor`:
99 static constexpr bool print_controller_signals = false;
100 static constexpr uint8_t controller_to_print = 0;
101 // Follow the test reference trajectory (true) or receive the target
102 // position over I²C or Serial (false):

```

```

103 static constexpr bool test_reference = false;
104 // Increase this divisor to slow down the test reference:
105 static constexpr uint8_t test_reference_speed_div = 4;
106 // Allow control for tuning and starting experiments over Serial:
107 static constexpr bool serial_control = true;
108 // I²C slave address (zero to disable I²C):
109 static constexpr uint8_t i2c_address = 8;
110
111 // Number of faders, must be between 1 and 4:
112 static constexpr size_t num_faders = 1;
113 // Actually drive the motors:
114 static constexpr bool enable_controller = true;
115 // Use analog pins (A0, A1, A6, A7) instead of (A0, A1, A2, A3), useful for
116 // saving digital pins on an Arduino Nano:
117 static constexpr bool use_A6_A7 = true;
118 // Use pin A2 instead of D13 as the motor driver pin for the second fader.
119 // Can only be used if `use_A6_A7` is set to true:
120 static constexpr bool fader_1_A2 = true;
121
122 // Capacitive touch sensing RC time threshold.
123 // Increase this time constant if the capacitive touch sense is too
124 // sensitive or decrease it if it's not sensitive enough:
125 static constexpr float touch_rc_time_threshold = 150e-6; // seconds
126 // Bit masks of the touch pins (must be on port B):
127 static constexpr uint8_t touch_masks[] = {1 << PB0, 1 << PB1, 1 << PB2,
128                                           1 << PB4};
129
130 // Use phase-correct PWM (true) or fast PWM (false), this determines the
131 // timer interrupt frequency, prefer phase-correct PWM with prescaler 1 on
132 // 16 MHz boards, and fast PWM with prescaler 1 on 8 MHz boards, both result
133 // in a PWM and interrupt frequency of 31.250 kHz
134 // (fast PWM is twice as fast):
135 static constexpr bool phase_correct_pwm = true;
136 // The fader position will be sampled once per `interrupt_divisor` timer
137 // interrupts, this determines the sampling frequency of the control loop.
138 // Some examples include 20 → 320 µs, 30 → 480 µs, 60 → 960 µs,
139 // 90 → 1,440 µs, 124 → 2,016 µs, 188 → 3,008 µs, 250 → 4,000 µs.
140 // 60 is the default, because it works with four faders. If you only use
141 // a single fader, you can go as low as 20 because you only need a quarter
142 // of the computations and ADC time:
143 static constexpr uint8_t interrupt_divisor = 60 / (1 + phase_correct_pwm);
144 // The prescaler for the timer, affects PWM and control loop frequencies:
145 static constexpr unsigned prescaler_fac = 1;
146 // The prescaler for the ADC, affects speed of analog readings:
147 static constexpr uint8_t adc_prescaler_fac = 64;
148
149 // Turn off the motor after this many seconds of inactivity:
150 static constexpr float timeout = 2;
151
152 // EMA filter factor for fader position filters:
153 static constexpr uint8_t adc_ema_K = 2;
154 // SMA filter length for setpoint filters, improves tracking of ramps if the
155 // setpoint changes in steps (e.g. when the DAW only updates the reference
156 // every 20 ms). Powers of two are significantly faster (e.g. 32 works well):
157 static constexpr uint8_t setpoint_sma_length = 0;
158
159 // ----- Computed Quantities ----- //
160
161 // Sampling time of control loop:
162 constexpr static float Ts = 1. * prescaler_fac * interrupt_divisor * 256 *
163     (1 + phase_correct_pwm) / F_CPU;
164 // Frequency at which the interrupt fires:
165 constexpr static float interrupt_freq =
166     1. * F_CPU / prescaler_fac / 256 / (1 + phase_correct_pwm);
167 // Clock speed of the ADC:
168 constexpr static float adc_clock_freq = 1. * F_CPU / adc_prescaler_fac;
169 // Pulse pin D13 if the control loop took too long:
170 constexpr static bool enable_overrun_indicator =
171     num_faders < 2 || fader_1_A2;
172
173 static_assert(use_A6_A7 || !fader_1_A2,
174     "Cannot use A2 for motor driver "
175     "and analog input at the same time");
176 };
177 constexpr uint8_t Config::touch_masks[];
178 constexpr float Ts = Config::Ts;
179
180 // ----- ADC, Capacitive Touch State and Motors ----- //
181
182 ADCManager<Config> adc;
183 TouchSense<Config> touch;
184 Motors<Config> motors;
185
186 // ----- Setpoints and References ----- //
187
188 // Setpoints (target positions) for all faders:
189 Reference<Config> references[Config::num_faders];
190
191 // ----- Controllers ----- //
192
193 // The main PID controllers. Need tuning for your specific setup:
194
195 PID controllers[] {
196     // This is an example of a controller with very little overshoot
197     {
198         6, // Kp: proportional gain
199         2, // Ki: integral gain
200         0.035, // Kd: derivative gain
201         Ts, // Ts: sampling time
202         60, // fc: cutoff frequency of derivative filter (Hz), zero to disable
203     },
204     // This one has more overshoot, but less ramp tracking error
205     {

```

```

206     4,      // Kp: proportional gain
207     11,     // Ki: integral gain
208     0.028,  // Kd: derivative gain
209     Ts,     // Ts: sampling time
210     40,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
211 },
212 // This is a very aggressive controller
213 {
214     8.55,   // Kp: proportional gain
215     440,    // Ki: integral gain
216     0.043,  // Kd: derivative gain
217     Ts,     // Ts: sampling time
218     70,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
219 },
220 // Fourth controller
221 {
222     6,      // Kp: proportional gain
223     2,      // Ki: integral gain
224     0.035,  // Kd: derivative gain
225     Ts,     // Ts: sampling time
226     60,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
227 },
228 };
229
230 template <uint8_t Idx>
231 void printControllerSignals(int16_t setpoint, int16_t adcval, int16_t control) {
232     // Send (binary) controller signals over Serial to plot in Python
233     if (Config::serial_control && references[Idx].experimentInProgress()) {
234         const int16_t data[3] {setpoint, adcval, control};
235         SLIPSender(Serial).writePacket(reinterpret_cast<const uint8_t *>(data),
236                                         sizeof(data));
237     }
238     // Print signals as text
239     else if (Config::print_controller_signals &&
240             Idx == Config::controller_to_print) {
241         Serial.print(setpoint);
242         Serial.print('\t');
243         Serial.print(adcval);
244         Serial.print('\t');
245         Serial.print((control + 256) * 2);
246         Serial.println();
247     }
248 }
249
250 template <uint8_t Idx>
251 void updateController(uint16_t setpoint, int16_t adcval, bool touched) {
252     auto &controller = controllers[Idx];
253
254     // Prevent the motor from being turned off after being touched
255     if (touched) controller.resetActivityCounter();
256
257     // Set the target position
258     if (Config::setpoint_sma_length > 0) {
259         static SMA<Config::setpoint_sma_length, uint16_t, uint32_t> sma;
260         uint16_t filtsetpoint = sma(setpoint);
261         controller.setSetpoint(filtsetpoint);
262     } else {
263         controller.setSetpoint(setpoint);
264     }
265
266     // Update the PID controller to get the control action
267     int16_t control = controller.update(adcval);
268
269     // Apply the control action to the motor
270     if (Config::enable_controller) {
271         if (touched) // Turn off motor if knob is touched
272             motors.setSpeed<Idx>(0);
273         else
274             motors.setSpeed<Idx>(control);
275     }
276
277     printControllerSignals<Idx>(controller.getSetpoint(), adcval, control);
278 }
279
280 template <uint8_t Idx>
281 void readAndUpdateController() {
282     // Read the ADC value for the given fader:
283     int16_t adcval = adc.read(Idx);
284     // If the ADC value was updated by the ADC interrupt, run the control loop:
285     if (adcval >= 0) {
286         // Check if the fader knob is touched
287         bool touched = touch.read(Idx);
288         // Read the target position
289         uint16_t setpoint = references[Idx].getNextSetpoint();
290         // Run the control loop
291         updateController<Idx>(setpoint, adcval, touched);
292         // Write -1 so the controller doesn't run again until the next value is
293         // available:
294         adc.write(Idx, -1);
295         if (Config::enable_overrun_indicator)
296             cbi(PORTB, 5); // Clear overrun indicator
297     }
298 }
299
300 // ----- Setup & Loop ----- //
301
302 void onRequest();
303 void onReceive(int);
304 void updateSerialIn();
305
306 void setup() {
307     // Initialize some globals
308     for (uint8_t i = 0; i < Config::num_faders; ++i) {

```

```

309 // all fader positions for the control loop start of as -1 (no reading)
310 adc.write(i, -1);
311 // reset the filter to the current fader position to prevent transients
312 adc.writeFiltered(i, analogRead(adc.channel_index_to_mux_address(i)));
313 // after how many seconds of not touching the fader and not changing
314 // the reference do we turn off the motor?
315 controllers[i].setActivityTimeout(Config::timeout);
316 }
317
318 // Configure the hardware
319 if (Config::print_frequencies || Config::print_controller_signals ||
320     Config::serial_control)
321     Serial.begin(1000000);
322
323 if (Config::enable_overnrun_indicator) sbi(DDRB, 5); // Pin 13 output
324
325 adc.begin();
326 touch.begin();
327 motors.begin();
328
329 if (Config::print_frequencies) {
330     Serial.println();
331     Serial.print(F("Interrupt frequency (Hz): "));
332     Serial.println(Config::interrupt_freq);
333     Serial.print(F("Controller sampling time (µs): "));
334     Serial.println(Config::Ts * 1e6);
335     Serial.print(F("ADC clock rate (Hz): "));
336     Serial.println(Config::adc_clock_freq);
337     Serial.print(F("ADC sampling rate (Sps): "));
338     Serial.println(adc.adc_rate);
339 }
340 if (Config::print_controller_signals) {
341     Serial.println();
342     Serial.println(F("Reference\tActual\tControl\t-"));
343     Serial.println(F("0\t0\t0\t0\r\n0\t0\t0\t024"));
344 }
345
346 // Initialize I2C slave and attach callbacks
347 if (Config::i2c_address) {
348     Wire.begin(Config::i2c_address);
349     Wire.onRequest(onRequest);
350     Wire.onReceive(onReceive);
351 }
352
353 // Enable Timer2 overflow interrupt, this starts reading the touch sensitive
354 // knobs and sampling the ADC, which causes the controllers to run in the
355 // main loop
356 sbi(TIMSK2, TOIE2);
357 }
358
359 void loop() {
360     if (Config::num_faders > 0) readAndUpdateController<0>();
361     if (Config::num_faders > 1) readAndUpdateController<1>();
362     if (Config::num_faders > 2) readAndUpdateController<2>();
363     if (Config::num_faders > 3) readAndUpdateController<3>();
364     if (Config::serial_control) updateSerialIn();
365 }
366
367 // ----- Interrupts ----- //
368
369 // Fires at a constant rate of `interrupt_freq`.
370 ISR(TIMER2_OVF_vect) {
371     // We don't have to take all actions at each interrupt, so keep a counter to
372     // know when to take what actions.
373     static uint8_t counter = 0;
374
375     adc.update(counter);
376     touch.update(counter);
377
378     ++counter;
379     if (counter == Config::interrupt_divisor) counter = 0;
380 }
381
382 // Fires when the ADC measurement is complete. Stores the reading, both before
383 // and after filtering (for the controller and for user input respectively).
384 ISR(ADC_vect) { adc.complete(); }
385
386 // ----- Wire ----- //
387
388 // Send the touch status and filtered fader positions to the master.
389 void onRequest() {
390     uint8_t touched = 0;
391     for (uint8_t i = 0; i < Config::num_faders; ++i)
392         touched |= touch.touched[i] << i;
393     Wire.write(touched);
394     for (uint8_t i = 0; i < Config::num_faders; ++i) {
395         uint16_t filt_read = adc.filtered_readings[i];
396         Wire.write(reinterpret_cast<const uint8_t *>(&filt_read), 2);
397     }
398 }
399
400 // Change the setpoint of the given fader based on the value in the message
401 // received from the master.
402 void onReceive(int count) {
403     if (count < 2) return;
404     if (Wire.available() < 2) return;
405     uint16_t data = Wire.read();
406     data |= uint16_t(Wire.read()) << 8;
407     uint8_t idx = data >> 12;
408     data &= 0x03FF;
409     if (idx < Config::num_faders) references[idx].setMasterSetpoint(data);
410 }
411

```

```

412 // ----- Serial ----- //
413
414 // Read SLIP messages from the serial port that allow dynamically updating the
415 // tuning of the controllers. This is used by the Python tuning script.
416 //
417 // Message format: <command> <fader> <value>
418 // Commands:
419 //   - p: proportional gain Kp
420 //   - i: integral gain Ki
421 //   - d: derivative gain Kd
422 //   - c: derivative filter cutoff frequency f_c (Hz)
423 //   - m: maximum absolute control output
424 //   - s: start an experiment, using getNextExperimentSetpoint
425 // Fader index: up to four faders are addressed using the characters '0' - '3'.
426 // Values: values are sent as 32-bit little Endian floating point numbers.
427 //
428 // For example the message 'c0x00x00x20x42' sets the derivative filter
429 // cutoff frequency of the first fader to 40.
430
431 void updateSerialIn() {
432     static SLIPParser parser;
433     static char cmd = '\0';
434     static uint8_t fader_idx = 0;
435     static uint8_t buf[4];
436     static_assert(sizeof(buf) == sizeof(float), "");
437     // This function is called if a new byte of the message arrives:
438     auto on_char_receive = [&](char new_byte, size_t index_in_packet) {
439         if (index_in_packet == 0) {
440             cmd = new_byte;
441         } else if (index_in_packet == 1) {
442             fader_idx = new_byte - '0';
443         } else if (index_in_packet < 6) {
444             buf[index_in_packet - 2] = new_byte;
445         }
446     };
447     // Convert the 4-byte buffer to a float:
448     auto as_f32 = [&] {
449         float f;
450         memcpy(&f, buf, sizeof(float));
451         return f;
452     };
453     // Read and parse incoming packets from Serial:
454     while (Serial.available() > 0) {
455         uint8_t c = Serial.read();
456         auto msg_size = parser.parse(c, on_char_receive);
457         // If a complete message of 6 bytes was received, and if it addresses
458         // a valid fader:
459         if (msg_size == 6 && fader_idx < Config::num_faders) {
460             // Execute the command:
461             switch (cmd) {
462                 case 'p': controllers[fader_idx].setKp(as_f32()); break;
463                 case 'i': controllers[fader_idx].setKi(as_f32()); break;
464                 case 'd': controllers[fader_idx].setKd(as_f32()); break;
465                 case 'c': controllers[fader_idx].setEMACutoff(as_f32()); break;
466                 case 'm': controllers[fader_idx].setMaxOutput(as_f32()); break;
467                 case 's':
468                     references[fader_idx].startExperiment(as_f32());
469                     controllers[fader_idx].resetIntegral();
470                     break;
471                 default: break;
472             }
473         }
474     }
475 }

```

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
10 template <class Config>
11 struct ADCManager {
12     /// Evenly distribute the analog readings in the control loop period.
13     constexpr static uint8_t adc_start_count =
14         Config::interrupt_divisor / Config::num_faders;
15     /// The rate at which we're sampling using the ADC.
16     constexpr static float adc_rate = Config::interrupt_freq / adc_start_count;
17     /// Check that this doesn't take more time than the 13 ADC clock cycles it
18     /// takes to actually do the conversion. Use 14 instead of 13 just to be safe.
19     static_assert(adc_rate <= Config::adc_clock_freq / 14, "ADC too slow");
20
21     /// Enable the ADC with Vcc reference, with the given prescaler, auto
22     /// trigger disabled, ADC interrupt enabled.
23     /// Called from main program, with interrupts enabled.
24     void begin();
25
26     /// Start an ADC conversion on the given channel.
27     /// Called inside an ISR.
28     void startConversion(uint8_t channel);
29
30     /// Start an ADC conversion at the right intervals.
31     /// @param counter
32     /// Counter that keeps track of how many times the timer interrupt
33     /// fired, between 0 and Config::interrupt_divisor - 1.
34     /// Called inside an ISR.
35     void update(uint8_t counter);
36
37     /// Read the latest ADC result.
38     /// Called inside an ISR.
39     void complete();
40
41     /// Get the latest ADC reading for the given index.
42     /// Called from main program, with interrupts enabled.
43     int16_t read(uint8_t idx);
44     /// Get the latest filtered ADC reading for the given index.
45     /// Called from main program, with interrupts enabled.
46     int16_t readFiltered(uint8_t idx);
47
48     /// Write the ADC reading for the given index.
49     /// Called from main program, with interrupts enabled.
50     void write(uint8_t idx, int16_t val);
51     /// Write the filtered ADC reading for the given index.
52     /// Called only before ADC interrupts are enabled.
53     void writeFiltered(uint8_t idx, int16_t val);
54
55     /// Convert the channel index between 0 and Config::num_faders - 1 to the
56     /// actual ADC multiplexer address.
57     constexpr static inline uint8_t
58     channel_index_to_mux_address(uint8_t adc_mux_idx) {
59         return Config::use_A6_A7
60             ? (adc_mux_idx < 2 ? adc_mux_idx : adc_mux_idx + 4)
61             : adc_mux_idx;
62     }
63
64     /// Index of the ADC channel currently being read.
65     uint8_t channel_index = Config::num_faders;
66     /// Latest ADC reading of each fader (updated in ADC ISR). Used for the
67     /// control loop.
68     volatile int16_t readings[Config::num_faders];
69     /// Filters for ADC readings.
70     EMA<Config::adc_ema_K, uint16_t> filters[Config::num_faders];
71     /// Filtered ADC readings. Used to output over MIDI.
72     volatile uint16_t filtered_readings[Config::num_faders];
73 };
74
75 template <class Config>
76 inline void ADCManager<Config>::begin() {
77     constexpr auto prescaler = factorToADCPrescaler(Config::adc_prescaler_fac);
78     static_assert(prescaler != ADCPrescaler::Invalid, "Invalid prescaler");
79
80     ATOMIC_BLOCK(ATOMIC_FORCEON) {
81         cbi(ADCSRA, ADEN); // Disable ADC
82
83         cbi(ADMUX, REFS1); // Vcc reference
84         sbi(ADMUX, REFS0); // Vcc reference
85
86         cbi(ADMUX, ADLAR); // 8 least significant bits in ADCL
87
88         setADCPrescaler(prescaler);
89
90         cbi(ADCSRA, ADATE); // Auto trigger disable
91         sbi(ADCSRA, ADIE); // ADC Interrupt Enable
92         sbi(ADCSRA, ADEN); // Enable ADC
93     }
94 }
95
96 template <class Config>
97 inline void ADCManager<Config>::update(uint8_t counter) {
98     if (Config::num_faders > 0 && counter == 0 * adc_start_count)
99         startConversion(0);
100     else if (Config::num_faders > 1 && counter == 1 * adc_start_count)
101         startConversion(1);
102     else if (Config::num_faders > 2 && counter == 2 * adc_start_count)

```

```

103     startConversion(2);
104     else if (Config::num_faders > 3 && counter == 3 * adc_start_count)
105         startConversion(3);
106 }
107
108 template <class Config>
109 inline void ADCManager<Config>::startConversion(uint8_t channel) {
110     channel_index = channel;
111     ADMUX &= 0xF0;
112     ADMUX |= channel_index_to_mux_address(channel);
113     sbi(ADCSRA, ADSC); // ADC Start Conversion
114 }
115
116 template <class Config>
117 inline void ADCManager<Config>::complete() {
118     if (Config::enable_overnrun_indicator && readings[channel_index] >= 0)
119         sbi(PORTB, 5); // Set overrrun indicator
120     uint16_t value = ADC; // Store ADC reading
121     readings[channel_index] = value;
122     // Filter the reading
123     auto &filter = filters[channel_index];
124     filtered_readings[channel_index] = filter(value << (6 - Config::adc_ema_K));
125 }
126
127 template <class Config>
128 inline int16_t ADCManager<Config>::read(uint8_t idx) {
129     int16_t v;
130     ATOMIC_BLOCK(ATOMIC_FORCEON) { v = readings[idx]; }
131     return v;
132 }
133
134 template <class Config>
135 inline void ADCManager<Config>::write(uint8_t idx, int16_t val) {
136     ATOMIC_BLOCK(ATOMIC_FORCEON) { readings[idx] = val; }
137 }
138
139 template <class Config>
140 inline int16_t ADCManager<Config>::readFiltered(uint8_t idx) {
141     int16_t v;
142     ATOMIC_BLOCK(ATOMIC_FORCEON) { v = filtered_readings[idx]; }
143     return v;
144 }
145
146 template <class Config>
147 inline void ADCManager<Config>::writeFiltered(uint8_t idx, int16_t val) {
148     filters[idx].reset(val);
149     filtered_readings[idx] = val;
150 }

```

## 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 (counter == 0) {
77         DDRB &= ~gpio_mask; // input mode, start charging
78     } else if (counter == touch_sense_thres) {
79         uint8_t touched_bits = PINB;
80         DDRB |= gpio_mask; // output mode, start discharging
81         for (uint8_t i = 0; i < Config::num_faders; ++i) {
82             bool touch_i = (touched_bits & Config::touch_masks[i]) == 0;
83             if (touch_i) {
84                 touch_timers[i] = touch_sense_stickiness;
85                 touched[i] = true;
86             } else if (touch_timers[i] > 0) {
87                 --touch_timers[i];
88                 if (touch_timers[i] == 0) touched[i] = false;
89             }
90         }
91     }
92 }
93
94 template <class Config>
95 bool TouchSense<Config>::read(uint8_t idx) {
96     bool t;
97     ATOMIC_BLOCK(ATOMIC_FORCEON) { t = touched[idx]; }
98     return t;
99 }

```



```

1  #pragma once
2
3  #include <stddef.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                 integral = newIntegral;
84                 return 0;
85             } else {
86                 errThres = 1;
87             }
88         } else {
89             ++activityCount;
90             errThres = 1;
91         }
92
93         bool backward = false;
94         int32_t calcIntegral = backward ? newIntegral : integral;
95
96         // Standard PID rule
97         float output = kp * error + ki_Ts * calcIntegral + kd_Ts * diff;
98
99         // Clamp and anti-windup
100         if (output > maxOutput)
101             output = maxOutput;
102         else if (output < -maxOutput)

```

```

102         output = -maxOutput;
103     else
104         integral = newIntegral;
105
106     return output;
107 }
108
109 void setKp(float kp) { this->kp = kp; }          ///< Proportional gain
110 void setKi(float ki) { this->ki_Ts = ki * this->Ts; } ///< Integral gain
111 void setKd(float kd) { this->kd_Ts = kd / this->Ts; } ///< Derivative gain
112
113 float getKp() const { return kp; }              ///< Proportional gain
114 float getKi() const { return ki_Ts / Ts; }       ///< Integral gain
115 float getKd() const { return kd_Ts * Ts; }       ///< Derivative gain
116
117 /// Set the cutoff frequency (-3 dB point) of the exponential moving average
118 /// filter that is applied to the input before taking the difference for
119 /// computing the derivative term.
120 void setEMACutoff(float f_c) {
121     float f_n = f_c * Ts; // normalized sampling frequency
122     this->emaAlpha = f_c == 0 ? 1 : calcAlphaEMA(f_n);
123 }
124
125 /// Set the reference/target/setpoint of the controller.
126 void setSetpoint(uint16_t setpoint) {
127     if (this->setpoint != setpoint) this->activityCount = 0;
128     this->setpoint = setpoint;
129 }
130 /// @see @ref setSetpoint(int16_t)
131 uint16_t getSetpoint() const { return setpoint; }
132
133 /// Set the maximum control output magnitude. Default is 255, which clamps
134 /// the control output in [-255, +255].
135 void setMaxOutput(float maxOutput) { this->maxOutput = maxOutput; }
136 /// @see @ref setMaxOutput(float)
137 float getMaxOutput() const { return maxOutput; }
138
139 /// Reset the activity counter to prevent the motor from turning off.
140 void resetActivityCounter() { this->activityCount = 0; }
141 /// Set the number of seconds after which the motor is turned off, zero to
142 /// keep it on indefinitely.
143 void setActivityTimeout(float s) {
144     if (s == 0)
145         activityThres = 0;
146     else
147         activityThres = uint16_t(s / Ts) == 0 ? 1 : s / Ts;
148 }
149
150 /// Reset the sum of the previous errors to zero.
151 void resetIntegral() { integral = 0; }
152
153 private:
154     float Ts = 1;          ///< Sampling time (seconds)
155     float maxOutput = 255;  ///< Maximum control output magnitude
156     float kp = 1;          ///< Proportional gain
157     float ki_Ts = 0;        ///< Integral gain times Ts
158     float kd_Ts = 0;        ///< Derivative gain divided by Ts
159     float emaAlpha = 1;     ///< Weight factor of derivative EMA filter.
160     float prevInput = 0;    ///< (Filtered) previous input for derivative.
161     uint16_t activityCount = 0; ///< How many ticks since last setpoint change.
162     uint16_t activityThres = 0; ///< Threshold for turning off the output.
163     uint8_t errThres = 1;    ///< Threshold with hysteresis.
164     int32_t integral = 0;    ///< Sum of previous errors for integral.
165     uint16_t setpoint = 0;   ///< Position reference.
166 };

```

## 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         setTimer0WGMMode(phase_correct ? Timer0WGMMode::PWM
11                                : Timer0WGMMode::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         setTimer2WGMMode(phase_correct ? Timer2WGMMode::PWM
23                                : Timer2WGMMode::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         if (Config::fader_1_A2)
69             sbi(DDRC, 2);
70         else
71             sbi(DDRB, 5);
72         sbi(DDRB, 3);
73     }
74     if (Config::num_faders > 2) {
75         sbi(DDRD, 4);
76         sbi(DDRD, 5);
77     }
78     if (Config::num_faders > 3) {
79         sbi(DDRD, 7);
80         sbi(DDRD, 6);
81     }
82 }
83
84 // Fast PWM (Table 14-6):
85 // Clear OC0B on Compare Match, set OC0B at BOTTOM (non-inverting mode).
86 // Phase Correct PWM (Table 14-7):
87 // Clear OC0B on compare match when up-counting. Set OC0B on compare match
88 // when down-counting.
89 template <class Config>
90 template <uint8_t Idx>
91 inline void Motors<Config>::forward(uint8_t speed) {
92     if (Idx >= Config::num_faders)
93         return;
94     else if (Idx == 0)
95         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
96             cbi(TCCR2A, COM2B0);
97             cbi(PORTD, 2);
98             OCR2B = speed;
99         }
100     else if (Idx == 1)
101         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
102             cbi(TCCR2A, COM2A0);

```

```

103         if (Config::fader_1_A2)
104             cbi(PORTC, 2);
105         else
106             cbi(PORTB, 5);
107         OCR2A = speed;
108     }
109     else if (Idx == 2)
110         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
111             cbi(TCCR0A, COM0B0);
112             cbi(PORTD, 4);
113             OCR0B = speed;
114         }
115     else if (Idx == 3)
116         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
117             cbi(TCCR0A, COM0A0);
118             cbi(PORTD, 7);
119             OCR0A = speed;
120         }
121     }
122 }
123 // Fast PWM (Table 14-6):
124 // Set OC0B on Compare Match, clear OC0B at BOTTOM (inverting mode).
125 // Phase Correct PWM (Table 14-7):
126 // Set OC0B on compare match when up-counting. Clear OC0B on compare match
127 // when down-counting.
128 template <class Config>
129 template <uint8_t Idx>
130 inline void Motors<Config>::backward(uint8_t speed) {
131     if (Idx >= Config::num_faders)
132         return;
133     else if (Idx == 0)
134         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
135             sbi(TCCR2A, COM2B0);
136             sbi(PORTD, 2);
137             OCR2B = speed;
138         }
139     else if (Idx == 1)
140         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
141             sbi(TCCR2A, COM2A0);
142             if (Config::fader_1_A2)
143                 sbi(PORTC, 2);
144             else
145                 sbi(PORTB, 5);
146             OCR2A = speed;
147         }
148     else if (Idx == 2)
149         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
150             sbi(TCCR0A, COM0B0);
151             sbi(PORTD, 4);
152             OCR0B = speed;
153         }
154     else if (Idx == 3)
155         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
156             sbi(TCCR0A, COM0A0);
157             sbi(PORTD, 7);
158             OCR0A = speed;
159         }
160     }
161 }
162 template <class Config>
163 template <uint8_t Idx>
164 inline void Motors<Config>::setSpeed(int16_t speed) {
165     if (speed >= 0)
166         forward<Idx>(speed);
167     else
168         backward<Idx>(-speed);
169 }

```

## 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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 Timer0WGMode : 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 setTimer0WGMode(Timer0WGMode 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
94     size_t write(const uint8_t *data, size_t len);
95     size_t writePacket(const uint8_t *data, size_t len) {
96         size_t sent = 0;
97         sent += beginPacket();
98         sent += write(data, len);
99         sent += endPacket();
100         return sent;
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 };

```