

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_counter`:
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 // Allow control for tuning and starting experiments over Serial:
105 static constexpr bool serial_control = true;
106 // I²C slave address (zero to disable I²C):
107 static constexpr uint8_t i2c_address = 8;
108
109 // Number of faders, must be between 1 and 4:
110 static constexpr size_t num_faders = 1;
111 // Actually drive the motors:
112 static constexpr bool enable_controller = true;
113 // Use analog pins (A0, A1, A6, A7) instead of (A0, A1, A2, A3), useful for
114 // saving digital pins on an Arduino Nano:
115 static constexpr bool use_A6_A7 = true;
116 // Use pin A2 instead of D13 as the motor driver pin for the second fader.
117 // Can only be used if `use_A6_A7` is set to true:
118 static constexpr bool fader_1_A2 = true;
119
120 // Capacitive touch sensing RC time threshold.
121 // Increase this time constant if the capacitive touch sense is too
122 // sensitive or decrease it if it's not sensitive enough:
123 static constexpr float touch_rc_time_threshold = 150e-6; // seconds
124 // Bit masks of the touch pins (must be on port B):
125 static constexpr uint8_t touch_masks[] = {1 << PB0, 1 << PB1, 1 << PB2,
126                                           1 << PB4};
127
128 // Use phase-correct PWM (true) or fast PWM (false), this determines the
129 // timer interrupt frequency, prefer phase-correct PWM with prescaler 1 on
130 // 16 MHz boards, and fast PWM with prescaler 1 on 8 MHz boards, both result
131 // in a PWM and interrupt frequency of 31.250 kHz
132 // (fast PWM is twice as fast):
133 static constexpr bool phase_correct_pwm = true;
134 // The fader position will be sampled once per `interrupt_counter` timer
135 // interrupts, this determines the sampling frequency of the control loop.
136 // Some examples include 20 → 320 µs, 30 → 480 µs, 60 → 960 µs,
137 // 90 → 1,440 µs, 124 → 2,016 µs, 188 → 3,008 µs, 250 → 4,000 µs.
138 // 60 is the default, because it works with four faders. If you only use
139 // a single fader, you can go as low as 20 because you only need a quarter
140 // of the computations and ADC time:
141 static constexpr uint8_t interrupt_counter = 60 / (1 + phase_correct_pwm);
142 // The prescaler for the timer, affects PWM and control loop frequencies:
143 static constexpr unsigned prescaler_fac = 1;
144 // The prescaler for the ADC, affects speed of analog readings:
145 static constexpr uint8_t adc_prescaler_fac = 64;
146
147 // Turn off the motor after this many seconds of inactivity:
148 static constexpr float timeout = 2;
149
150 // EMA filter factor for fader position filters:
151 static constexpr uint8_t adc_ema_K = 2;
152 // SMA filter length for setpoint filters, improves tracking of ramps if the
153 // setpoint changes in steps (e.g. when the DAW only updates the reference
154 // every 20 ms). Powers of two are significantly faster (e.g. 32 works well):
155 static constexpr uint8_t setpoint_sma_length = 0;
156
157 // ----- Computed Quantities ----- //
158
159 // Sampling time of control loop:
160 constexpr static float Ts = 1. * prescaler_fac * interrupt_counter * 256 *
161                          (1 + phase_correct_pwm) / F_CPU;
162 // Frequency at which the interrupt fires:
163 constexpr static float interrupt_freq =
164     1. * F_CPU / prescaler_fac / 256 / (1 + phase_correct_pwm);
165 // Clock speed of the ADC:
166 constexpr static float adc_clock_freq = 1. * F_CPU / adc_prescaler_fac;
167 // Pulse pin D13 if the control loop took too long:
168 constexpr static bool enable_overrun_indicator =
169     num_faders < 2 || fader_1_A2;
170
171 static_assert(use_A6_A7 || !fader_1_A2,
172             "Cannot use A2 for motor driver "
173             "and analog input at the same time");
174 };
175 constexpr uint8_t Config::touch_masks[];
176
177 constexpr float Ts = Config::Ts;
178
179 // ----- ADC, Capacitive Touch State and Motors ----- //
180
181 ADCManager<Config> adc;
182 TouchSense<Config> touch;
183 Motors<Config> motors;
184
185 // ----- Setpoints and References ----- //
186
187 // Reference speed for tuning experiments:
188 float serial_experiment_speed[Config::num_faders];
189 // Setpoints (target positions) for all faders (updated in I²C interrupt):
190 volatile int16_t setpoints[Config::num_faders];
191
192 // ----- Controllers ----- //
193
194 // The main PID controllers. Need tuning for your specific setup:
195
196 PID controllers[] {
197     // This is an example of a controller with very little overshoot
198     {
199         5, // Kp: proportional gain
200         2, // Ki: integral gain
201         0.028, // Kd: derivative gain
202         Ts, // Ts: sampling time
203         40, // fc: cutoff frequency of derivative filter (Hz), zero to disable
204     },
205     // This one has more overshoot, but less ramp tracking error

```

```

206 {
207     4,      // Kp: proportional gain
208     11,     // Ki: integral gain
209     0.028, // Kd: derivative gain
210     Ts,     // Ts: sampling time
211     40,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
212 },
213 // This is a very aggressive controller
214 {
215     8.55,   // Kp: proportional gain
216     440,    // Ki: integral gain
217     0.043,  // Kd: derivative gain
218     Ts,     // Ts: sampling time
219     70,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
220 },
221 // Fourth controller
222 {
223     4,      // Kp: proportional gain
224     11,     // Ki: integral gain
225     0.028,  // Kd: derivative gain
226     Ts,     // Ts: sampling time
227     40,     // fc: cutoff frequency of derivative filter (Hz), zero to disable
228 },
229 };
230
231 template <uint8_t Idx>
232 void printControllerSignals(int16_t setpoint, int16_t adcval, int16_t control) {
233     // Send (binary) controller signals over Serial to plot in Python
234     if (Config::serial_control && serial_experiment_speed[Idx] > 0) {
235         const int16_t data[3] {setpoint, adcval, control};
236         SLIPSender(Serial).writePacket(reinterpret_cast<const uint8_t *>(data),
237                                       sizeof(data));
238     }
239     // Print signals as text
240     else if (Config::print_controller_signals &&
241             Idx == Config::controller_to_print) {
242         Serial.print(setpoint);
243         Serial.print('\t');
244         Serial.print(adcval);
245         Serial.print('\t');
246         Serial.print((control + 256) * 2);
247         Serial.println();
248     }
249 }
250
251 template <uint8_t Idx>
252 void updateController(uint16_t setpoint, int16_t adcval, bool touched) {
253     auto &controller = controllers[Idx];
254
255     // Prevent the motor from being turned off after being touched
256     if (touched) controller.resetActivityCounter();
257
258     // Set the target position
259     if (Config::setpoint_sma_length > 0) {
260         static SMA<Config::setpoint_sma_length, uint16_t, uint32_t> sma;
261         uint16_t filtsetpoint = sma(setpoint);
262         controller.setSetpoint(filtsetpoint);
263     } else {
264         controller.setSetpoint(setpoint);
265     }
266
267     // Update the PID controller to get the control action
268     int16_t control = controller.update(adcval);
269
270     // Apply the control action to the motor
271     if (Config::enable_controller) {
272         if (touched) // Turn off motor if knob is touched
273             motors.setSpeed<Idx>(0);
274         else
275             motors.setSpeed<Idx>(control);
276     }
277
278     printControllerSignals<Idx>(controller.getSetpoint(), adcval, control);
279 }
280
281 template <uint8_t Idx>
282 void readAndUpdateController() {
283     // Read the ADC value for the given fader:
284     int16_t adcval = adc.read(Idx);
285     // If the ADC value was updated by the ADC interrupt, run the control loop:
286     if (adcval >= 0) {
287         // Check if the fader knob is touched
288         bool touched = touch.read(Idx);
289         // Read the target position
290         uint16_t setpoint;
291
292         if (Config::serial_control && serial_experiment_speed[Idx] > 0)
293             // from the tuning experiment reference
294             setpoint = getNextExperimentSetpoint<Idx>(serial_experiment_speed[Idx]);
295         else if (Config::test_reference)
296             // from the test reference
297             setpoint = getNextSetpoint<Idx>(4);
298         else
299             // from the I2C master
300             ATOMIC_BLOCK(ATOMIC_FORCEON) { setpoint = ::setpoints[Idx]; }
301         updateController<Idx>(setpoint, adcval, touched);
302         // Write -1 so the controller doesn't run again until the next value is
303         // available:
304         adc.write(Idx, -1);
305         if (Config::enable_overrun_indicator)
306             cbi(PORTB, 5); // Clear overrun indicator
307     }
308 }

```

```

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

```

```

412 // received from the master.
413 void onReceive(int count) {
414     if (count < 2) return;
415     if (Wire.available() < 2) return;
416     uint16_t data = Wire.read();
417     data |= uint16_t(Wire.read()) << 8;
418     uint8_t idx = data >> 12;
419     data &= 0x03FF;
420     if (idx < Config::num_faders) setpoints[idx] = data;
421 }
422
423 // ----- Serial ----- //
424
425 // Read SLIP messages from the serial port that allow dynamically updating the
426 // tuning of the controllers. This is used by the Python tuning script.
427 //
428 // Message format: <command> <fader> <value>
429 // Commands:
430 //   - p: proportional gain Kp
431 //   - i: integral gain Ki
432 //   - d: derivative gain Kd
433 //   - c: derivative filter cutoff frequency f_c (Hz)
434 //   - m: maximum absolute control output
435 //   - s: start an experiment, using getNextExperimentSetpoint
436 // Fader index: up to four faders are addressed using the characters '0' - '3'.
437 // Values: values are sent as 32-bit little Endian floating point numbers.
438 //
439 // For example the message 'c0\x00\x00\x20\x42' sets the derivative filter
440 // cutoff frequency of the first fader to 40.
441
442 void updateSerialIn() {
443     static SLIPParser parser;
444     static char cmd = '\0';
445     static uint8_t fader_idx = 0;
446     static uint8_t buf[4];
447     static_assert(sizeof(buf) == sizeof(float), "");
448     // This function is called if a new byte of the message arrives:
449     auto on_char_receive = [&](char new_byte, size_t index_in_packet) {
450         if (index_in_packet == 0) {
451             cmd = new_byte;
452         } else if (index_in_packet == 1) {
453             fader_idx = new_byte - '0';
454         } else if (index_in_packet < 6) {
455             buf[index_in_packet - 2] = new_byte;
456         }
457     };
458     // Convert the 4-byte buffer to a float:
459     auto as_f32 = [&] {
460         float f;
461         memcpy(&f, buf, sizeof(float));
462         return f;
463     };
464     // Read and parse incoming packets from Serial:
465     while (Serial.available() > 0) {
466         uint8_t c = Serial.read();
467         auto msg_size = parser.parse(c, on_char_receive);
468         // If a complete message of 6 bytes was received, and if it addresses
469         // a valid fader:
470         if (msg_size == 6 && fader_idx < Config::num_faders) {
471             // Execute the command:
472             switch (cmd) {
473                 case 'p': controllers[fader_idx].setKp(as_f32()); break;
474                 case 'i': controllers[fader_idx].setKi(as_f32()); break;
475                 case 'd': controllers[fader_idx].setKd(as_f32()); break;
476                 case 'c': controllers[fader_idx].setEMACutoff(as_f32()); break;
477                 case 'm': controllers[fader_idx].setMaxOutput(as_f32()); break;
478                 case 's':
479                     serial_experiment_speed[fader_idx] = as_f32();
480                     controllers[fader_idx].resetIntegral();
481                     break;
482                 default: break;
483             }
484         }
485     }
486 }

```

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_counter / 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_counter - 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_counter;
18     /// Check that the threshold is smaller than the control loop period:
19     static_assert(touch_sense_thres < Config::interrupt_counter,
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_counter - 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 template <class Config>
45 struct Motors {
46     void begin();
47     template <uint8_t Idx>
48     void setSpeed(int16_t speed);
49
50     template <uint8_t Idx>
51     void setupGPIO();
52     template <uint8_t Idx>
53     void forward(uint8_t speed);
54     template <uint8_t Idx>
55     void backward(uint8_t speed);
56 };
57
58 template <class Config>
59 inline void Motors<Config>::begin() {
60     setupMotorTimers<Config>();
61
62     if (Config::num_faders > 0) {
63         sbi(DDRD, 2);
64         sbi(DDRD, 3);
65     }
66     if (Config::num_faders > 1) {
67         if (Config::fader_1_A2)
68             sbi(DDRC, 2);
69         else
70             sbi(DDRB, 5);
71         sbi(DDRB, 3);
72     }
73     if (Config::num_faders > 2) {
74         sbi(DDRD, 4);
75         sbi(DDRD, 5);
76     }
77     if (Config::num_faders > 3) {
78         sbi(DDRD, 7);
79         sbi(DDRD, 6);
80     }
81 }
82
83 // Fast PWM (Table 14-6):
84 // Clear OC0B on Compare Match, set OC0B at BOTTOM (non-inverting mode).
85 // Phase Correct PWM (Table 14-7):
86 // Clear OC0B on compare match when up-counting. Set OC0B on compare match
87 // when down-counting.
88 template <class Config>
89 template <uint8_t Idx>
90 inline void Motors<Config>::forward(uint8_t speed) {
91     if (Idx >= Config::num_faders)
92         return;
93     else if (Idx == 0)
94         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
95             cbi(TCCR2A, COM2B0);
96             cbi(PORTD, 2);
97             OCR2B = speed;
98         }
99     else if (Idx == 1)
100         ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
101             cbi(TCCR2A, COM2A0);
102             if (Config::fader_1_A2)

```

```

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

```

Reference.hpp

```

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

```

```

103     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
104     127, 127,
105     // Jump low
106     25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
107     25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
108     // Jump middle
109     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
110     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
111     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
112     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
113     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
114     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
115     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
116     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
117     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
118     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
119     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
120     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
121     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
122     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
123     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
124     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
125     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
126     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
128     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
129     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
130     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
131     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
132     127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
133 #endif
134 // Ramp down
135     127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113,
136     112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97,
137     96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78,
138     77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59,
139     58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40,
140     39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
141     20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
142 // Low
143     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,
144     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,
145     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,
146     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,
147     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,
148     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,
149     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,
150     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,
151     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,
152     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,
153     0, 0, 0, 0, 0, 0,
154 #if 0
155 ,
156 // Low
157     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,
158     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,
159     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,
160     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,
161     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,
162     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,
163     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,
164     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,
165     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,
166     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,
167     0, 0, 0, 0, 0, 0,
168 // Low
169     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,
170     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,
171     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,
172     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,
173     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,
174     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,
175     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,
176     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,
177     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,
178     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,
179     0, 0, 0, 0, 0, 0,
180 // Low
181     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,
182     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,
183     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,
184     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,
185     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,
186     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,
187     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,
188     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,
189     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,
190     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,
191     0, 0, 0, 0, 0, 0,
192 #endif
193 };
194 #else
195 const uint8_t reference_signal[] PROGMEM = {
196     127,
197 };
198 #endif
199
200 template <class T, size_t N>
201 constexpr size_t len(const T (&)[N]) {
202     return N;
203 }
204 const size_t reference_len = len(reference_signal);
205

```



```

206 template <uint8_t>
207 inline uint16_t getNextSetpoint(uint8_t speed_fac) {
208     static uint8_t counter = 0;
209     static size_t index = 0;
210
211     uint16_t setpoint = pgm_read_byte(reference_signal + index) * 4;
212     ++counter;
213     if (counter >= speed_fac) {
214         counter = 0;
215         ++index;
216         if (index == reference_len) index = 0;
217     }
218     return setpoint;
219 }
220
221 template <uint8_t>
222 inline uint16_t getNextExperimentSetpoint(float &speed) {
223     static uint32_t index = 0;
224
225     constexpr uint16_t rampup = 0xFFFF;
226     constexpr uint16_t rampdown = 0xFFFE;
227     struct TestSeq {
228         uint16_t setpoint;
229         uint32_t duration;
230     };
231     constexpr TestSeq seqs[] {
232         {0, 256}, {rampup, 128}, {1023, 32}, {0, 64}, {333, 32},
233         {666, 32}, {333, 32}, {0, 32}, {512, 256},
234     };
235
236     uint16_t setpoint = 0xFFFF;
237     uint32_t start = 0;
238     for (uint8_t i = 0; i < len(seqs); ++i) {
239         uint32_t duration = seqs[i].duration * speed;
240         uint16_t setpoint_i = seqs[i].setpoint;
241         uint32_t end = start + duration;
242         if (index < end) {
243             if (setpoint_i == rampup) {
244                 setpoint = 1024 * (index - start) / duration;
245             } else if (setpoint_i == rampdown) {
246                 setpoint = 1023 - 1024 * (index - start) / duration;
247             } else {
248                 setpoint = setpoint_i;
249             }
250             break;
251         }
252         start = end;
253     }
254
255     if (setpoint == 0xFFFF) {
256         index = 0;
257         speed = 0;
258         return 0;
259     }
260
261     ++index;
262     return setpoint;
263 }

```

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 class SLIPParser {
11 public:
12     template <class Callback>
13     size_t parse(uint8_t c, Callback callback);
14
15     void reset() {
16         size = 0;
17         escape = false;
18     }
19
20 private:
21     size_t size = 0;
22     bool escape = false;
23 };
24
25 template <class Callback>
26 size_t SLIPParser::parse(uint8_t c, Callback callback) {
27     // https://datatracker.ietf.org/doc/html/rfc1055
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 class SLIPSender {
87 public:
88     SLIPSender(Stream &stream) : stream(stream) {}
89
90     size_t beginPacket() { return stream.write(SLIP_Constants::END); }
91     size_t endPacket() { return stream.write(SLIP_Constants::END); }
92
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 private:

```

```

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

```

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 };

```