```
/*
 * Cycle assist in the Sanyo/Sachs hub.
 * Read out sensor values and determine
 * its support from that. January 2020
*/

const int Thermistor_pin_out  = 4; // 5V for
thermistor readout
const int Thermistor_pin_in   = A1; // Analog
thermistor voltage readout
const int Batt_Volt_pin_in    = A2; // Analog
battery voltage readout
const int driver_pin          = 3; // Pin that
steers the motor driver.
bool ifdo_lim25               = true; // global
boolean, whether 25 km/h speed limit should be
enforced.
int Drive_PWM                 = 0; // define the
initial drive support variable
int Drive_PWM_recommend       = 0; // the
recommended drive support from PAS and speed
sensor.
int Drive_PWM_max_safe        = 0; // The
maximum drive support according to safety checks.
bool ifdo_PWR_red_temp        = true; //
initialize the 'safety check' booleans
bool ifdo_PWR_red_batt        = true;
bool ifdo_disable_batt        = true;
// REED SENSOR/WHEEL SPEED:
const int Reed_pin            = 5; // Pin which
reads the reed (speed) sensor.
unsigned long reed_2_LOW      = 0; // time stamp
of the last jump from LOW to HIGH of reed
readout.
unsigned long reed_2_HIGH     = 0; // time stamp
of the last jump from HIGH to LOW of reed
readout.
double wheel_speed            = 0; // Current
```

```
speed [km/h] of the wheel.

// HALL SENSOR/WHEEL SPEED:
const int PAS_pin              = 6; // Pin which
reads the PAS (pedal) sensor.
unsigned long PAS_2_LOW       = 0; // time stamp
of the last jump from LOW to HIGH of Hall sensor
readout.
unsigned long PAS_2_HIGH      = 0; // time stamp
of the last jump from HIGH to LOW of Hall sensor
readout.
double cadence                 = 0; // Current
speed [Hz] of the pedal rotation.

void setup()
{
  // Upon startup:
  Serial.begin(115200);
  // Read if run in 25km-mode or 25km+mode:
  check_25km_limit(PAS_pin);

  // Initial settings:
  digitalWrite(Thermistor_pin_out, HIGH); // Set
thermistor readout output pins to high:
  // set the REED sensor to HIGH as default:
  pinMode(Reed_pin, INPUT_PULLUP);
  digitalWrite(driver_pin, HIGH); // No drive
support at startup.
}

void loop() {
  // First perform safety checks, and determine
the maximum Drive support the motor can safely
provide:
  Drive_PWM_max_safe =
Determine_PWM_max_safe(Thermistor_pin_in,
Batt_Volt_pin_in);
```

```
  // Read the current state of the wheel speed
sensors, and if possible, update the wheel speed:
  update_wheel_speed(Reed_pin);
  // Read the current state of the pedal speed
sensors, and if possible, update the pedal speed:
  update_cadence(PAS_pin);
  // With that information, update the
recommended drive support level:
  update_drive_support(Drive_PWM_max_safe);
  // Write the signal to the motor driver:
  analogWrite(driver_pin, 255-Drive_PWM);
  //delay(100);
  print_drive_variables();
  //print_safety_variables();
}

void check_25km_limit(int PAS_pin)
{ // We wait a while to see if the pedals are
moved sufficiently in those first seconds:
  unsigned long t_0 = millis(); // Time the start
  unsigned long max_wait_total = 1000; //[msec]
  while ((millis() - t_0) < max_wait_total)
  { // Measure the cadence:
    update_cadence(PAS_pin);
  }
  Serial.print("cadence: "); Serial.
print(cadence);  Serial.print("\n");
  if (cadence > 150)
  { // If the cadence has risen enough from
zero, we lift the 25 km/h limitation:
    Serial.print("been here");
    ifdo_lim25 = false;
  }
  cadence = 0; // reset the cadence to zero.
}

// Local function: print variables for debug:
void print_safety_variables(void)
```

```
{
  Serial.print("\n 25 lim:");
  Serial.print(ifdo_lim25);
  Serial.print("\t red_temp?:");
  Serial.print(ifdo_PWR_red_temp);
  Serial.print("\t Voltage?:");
  Serial.
print(Read_battery_voltage(Batt_Volt_pin_in));
  Serial.print("\t Low voltage?:");
  Serial.print(ifdo_PWR_red_batt);
  Serial.print("\t Empty battery?:");
  Serial.print(ifdo_disable_batt);
}
void print_drive_variables(void)
{
  Serial.print("\n wheel_speed:");
  Serial.print(wheel_speed);
  Serial.print("\t cadence:");
  Serial.print(cadence);
  Serial.print("\t PWM_max_safe:");
  Serial.print(Drive_PWM_max_safe);
  Serial.print("\t Drive_PWM:");
  Serial.print(Drive_PWM);
}

// Local function: check wheter the 25 km/h
limitation should be lifted :
bool Enable_lim25(int lim25_pin)
{
  // If the user holds both battery check and on
button on, the 25 km/h limitation is lifted:
  int Threshold_Voltage_from_Voltage_button = 2.
5; //[Volt] between GND and button.
  bool ifdo_lim25   = true; // Boolean whether
motor support should be limited to 25 km/h top
speed.
  if (analogRead(lim25_pin) <=
Threshold_Voltage_from_Voltage_button)
```

```
    {
      // The button was held pressed (connects
to GND), therefore we disable the 25 km/h limit:
      ifdo_lim25 = false;
    }
  return ifdo_lim25;
}

// Local function, determine the maximum Drive
support the motor can safely provide:
int Determine_PWM_max_safe(int
Thermistor_pin_in, int Batt_Volt_pin_in)
{
  // Check whether motor temperature is too high
for full power:
  ifdo_PWR_red_temp =
Read_motor_temp(Thermistor_pin_in);
  // Then check the battery voltage, if it is
low, we should reduce power:
  ifdo_PWR_red_batt =
is_Batt_low(Batt_Volt_pin_in);
  // Then check the battery voltage, if it is
empty, we should disable the drive:
  ifdo_disable_batt =
is_Batt_empty(Batt_Volt_pin_in);
  // From the above safety checks, determine the
maximum drive PWM duty cycle possible:
  int Drive_PWM_max =
drive_support_max_safe(ifdo_PWR_red_temp,
ifdo_PWR_red_batt, ifdo_disable_batt);
  return Drive_PWM_max;
}


// Local function: read out the motor
temperature, decide whether reduced power mode
should be enabled:
bool Read_motor_temp(int Thermistor_pin_in)
```

```
{
  // The thermal resistor value is read out by a
voltage divider bridge
  // consisting of the thermistor and an
external resistor.
  int ext_resistor = 12; // [kOhm]
  // At critically high temperatures, the
thermistor has a value of:
  int thermistor_threshold = 8; // [kOhm]
  // Now measure the voltage and compare to the
critical one:
  int readout_threshold = 870 ;
//1023*ext_resistor/(ext_resistor +
thermistor_threshold);
  int readout = analogRead(Thermistor_pin_in);
  int PWR_red = true;
  if (readout < readout_threshold)
  {
    PWR_red = false;
  }
  return PWR_red;
}

// Local function: judge whether the drive needs
to reduce power, or disable, due to too low
battery voltage:
bool is_Batt_low(int Batt_Volt_pin_in)
{
  bool PWR_red = true;
  // Define the constants and thresholds:
  int thres_PWR_red_volt = 3.2*6; //[Volt] Below
this voltage, reduced power mode should be
enabled.
  // Read the battery voltage:
  double Batt_volt =
Read_battery_voltage(Batt_Volt_pin_in);

  // Compare the battery voltage to the
```

```
thresholds for reduced power and drive disable:
  if (Batt_volt > thres_PWR_red_volt)
  {
    PWR_red = false;
  }
  return PWR_red;
}

// Local function: judge whether the drive needs
to disable, due to too low battery voltage:
bool is_Batt_empty(int Batt_Volt_pin_in)
{
  bool ifdo_drive_disable = true;
  // Define the constants and thresholds:
  int thres_PWR_drive_disable = 3.0*6; //[Volt]
Below this voltage, the drive should be disabled.
  // Read the battery voltage:
  double Batt_volt =
Read_battery_voltage(Batt_Volt_pin_in);
  // Compare the battery voltage to the
thresholds for reduced power and drive disable:
  if (Batt_volt > thres_PWR_drive_disable)
  {
    ifdo_drive_disable = false;
  }
  return ifdo_drive_disable;
}

// Local function: read out battery voltage, to
be within safe cell voltage limits:
double Read_battery_voltage(int Batt_Volt_pin_in)
{
  // The battery voltage is read by constructing
a voltage dividing bridge, with the resistor
values:
  int R1  = 12; //[kOhm] Resistor from Battery
plus to analogRead pin.
  int R2  = 2; // [kOhm] Resistor from
```

```
analogRead to GND.
  int Batt_volt_readout =
analogRead(Batt_Volt_pin_in);

  double Batt_volt = (double) (R1+R2)/(R2) *
Batt_volt_readout*5/1024;
  return Batt_volt;
}

// Local function to define a maximum Drive
support, using the safety checks performed
before:
int drive_support_max_safe(bool
ifdo_PWR_red_temp, bool ifdo_PWR_red_batt, bool
ifdo_disable_batt)
{
  // Determine the maximum amount of support we
can safely deliver:
  int PWM_max = 255;
  // Check whether the power should be reduced:
  if (ifdo_PWR_red_temp || ifdo_PWR_red_batt)
  { PWM_max = 125;}
  if (ifdo_disable_batt)
  { PWM_max = 0;}
  return PWM_max;
}

// Local function to calculate the current wheel
rotation speed, if possible:
void update_wheel_speed(int REED_pin)
{
  // 25 km/h: 1/12.5= 80 ms between reed pulses
  double dt_HIGH_max = 500; //[msec] Larger than
this means wheel standstill
  double dt_LOW_max = 400; //[msec] Larger than
this means wheel standstill
  double nof_running_averages = 20; // The
cadence is updated with running averages.
```

```
  unsigned long t_now = millis();   // [msec]
measure the current time
  int reed_status = digitalRead(REED_pin);
//'LOW' (means that a magnet is in front of the
sensor, high means there is not):

  // Which status was read last?
  if (reed_2_HIGH == 0 && reed_2_LOW == 0)
  { // This probably means that neither value
has ever been recorded, we therefore have no
speed information and recommend no support:
    wheel_speed = 0;
    if (reed_status == LOW)
    { reed_2_LOW = t_now;  } // Initialize the
first timestamp}
    if (reed_status == HIGH)
    { reed_2_HIGH = t_now;  }// Initialize the
first timestamp}
    return;
  }

  if (reed_2_HIGH > reed_2_LOW){ // The switch
from LOW to HIGH was read last:
      //Serial.print("\t LOW to HIGH switch was
read last");
    if (reed_status == LOW) { // This means we
have a switch of level compared to the last
readout.
      // We overwrite the reed_2_LOW time stamp:
      reed_2_LOW = t_now;
    }
    // If the level has not switched, it might
have been too long in the same status (standing
still):
    if ((reed_status == HIGH && (t_now -
reed_2_HIGH) > dt_HIGH_max) || reed_2_LOW == 0)
    { // This means the pedals are practically
```

```
standing still, or have not moved yet:
        wheel_speed = 0;//recommend no support.
        //Serial.print("\t Too long in HIGH! no
support");
        return;
    }
    return;
  }

  if (reed_2_HIGH < reed_2_LOW)  { // The 'LOW'
status was read last:
  //Serial.print("\t HIGH to LOW switch was read
last");
    if (reed_status == HIGH) { // This means we
have a switch of level.
      // We overwrite the reed_2_HIGH time stamp:
      reed_2_HIGH = t_now;
      // We can measure how long this switch of
level took:
      double dt_HIGH = (t_now - reed_2_LOW);
      // From this, we can measure the speed:
      double current_wheel_speed =
(25*80)/dt_HIGH;
      // Here we assume that the wheel speed is
within acceptable limits:
      if (current_wheel_speed > wheel_speed){ //
If the cadence increases, we let it increase
slowly through running average:
        wheel_speed =
(nof_running_averages*wheel_speed +
current_wheel_speed)/(nof_running_averages+1);
// [km/h]
      }
      if (current_wheel_speed <= wheel_speed){
// If it has decreased, we must act quickly in
case of a stop.
        wheel_speed = current_wheel_speed; //
[km/h]
```

```
      }
    }
    if ((reed_status == LOW && (t_now -
reed_2_LOW) > dt_LOW_max) || reed_2_HIGH == 0)
    { // This means the pedals are practically
standing still:
       wheel_speed = 0;//recommend no support.
       //Serial.print("\t Too long in LOW! no
support");
        return;
    }
    return;
  }
}

// Local function to calculate the current wheel
rotation speed, if possible:
void update_cadence(int PAS_pin)
{
//  // 8 magnets in the PAS sensor, 8 block
signals per rotation
//  // The minimum rotation speed of the pedal
is 0.5 Hz, so 4 Hz (250 millisecond) for the
signal:
  double dt_HIGH_max = 400; //[msec] Larger than
this means standstill of pedals
  double dt_LOW_max = 250; //[msec] Larger than
this means standstill of pedals
  double nof_running_averages = 4; // The
cadence is updated with running averages.

  unsigned long t_now = millis();   // [msec]
measure the current time
  int PAS_status = digitalRead(PAS_pin); //'LOW'
(means that a magnet is in front of the sensor,
high means there is not):

  // Which status was read last?
```

```
  if (PAS_2_HIGH == 0 && PAS_2_LOW == 0)
  { // This probably means that neither value
has ever been recorded, we therefore have no
speed information and recommend no support:
    cadence = 0;
    if (PAS_status == LOW)
    { PAS_2_LOW = t_now;}
    if (PAS_status == HIGH)
    { PAS_2_HIGH = t_now; }
  }

  if (PAS_2_HIGH > PAS_2_LOW){ // The switch
from LOW to HIGH was read last:
      //Serial.print("\t LOW to HIGH switch was
read last");
    if (PAS_status == LOW) { // This means we
have a switch of level compared to the last
readout.
      // We overwrite the PAS_2_LOW time stamp:
      PAS_2_LOW = t_now;
    }
    // If the level has not switched, it might
have been too long in the same status (standing
still):
    if ((PAS_status == HIGH && (t_now -
PAS_2_HIGH) > dt_HIGH_max) || PAS_2_LOW == 0)
    { // This means the pedals are practically
standing still, or have not moved yet:
        cadence = 0;//recommend no support.
        //Serial.print("\t Too long in HIGH! no
support");
        return;
    }
    return;
  }

  if (PAS_2_HIGH < PAS_2_LOW)  { // The 'LOW'
status was read last:
```

```
  //Serial.print("\t HIGH to LOW switch was read
last");
    if (PAS_status == HIGH) { // This means we
have a switch of level.
      // We overwrite the PAS_2_HIGH time stamp:
      PAS_2_HIGH = t_now;
      // We can measure how long this switch of
level took:
      double dt_HIGH = (t_now - PAS_2_LOW);
      // From this, we can measure the speed:
      double current_cadence =
max(max(dt_HIGH_max, dt_LOW_max) - dt_HIGH, 1);
// [Hz]
      //Serial.print("\t current cadence: ");
Serial.print(current_cadence);
      // Here we assume that the cadence is
within acceptable limits:
      if (current_cadence > cadence){ // If the
cadence increases, we let it increase slowly
through running average:
        cadence = (nof_running_averages*cadence
+ current_cadence)/(nof_running_averages+1); //
[Hz]
      }
      if (current_cadence <= cadence){ // If it
has decreased, we must act quickly in case of a
stop.
        cadence = current_cadence; // [Hz]
      }

    }
    if ((PAS_status == LOW && (t_now -
PAS_2_LOW) > dt_LOW_max) || PAS_2_HIGH == 0)
    { // This means the pedals are practically
standing still:
       cadence = 0;//recommend no support.
       //Serial.print("\t Too long in LOW! no
support");
```

```
        return;
      }
    return;
  }
}

// Local function to determine the Drive_PWM:
int update_drive_support(int PWM_max_safe)
{
  // Set constants and Parameters:
  int dPWM_max_increase   = 1; // Maximum
increase of drive support compared to the
previous
  int dPWM_max_decrease_25= 1; // Maximum
decrease of drive support compared to the
previous, in 25 km/h mode.
  float cadence_minimum = 200; // Minimum
cadence, below no drive support
  float wheel_speed_minimum = 0.3; // Minimum
wheel speed, below that: no drive support
  // 25 km/h: 1/12.5= 80 ms between reed pulses
  double dt_25  = 80; //[msec]

  // Disable the support if the cadence or speed
is at zero, or safety checks command a stop of
support:
  if (wheel_speed <= wheel_speed_minimum ||
cadence <= cadence_minimum || PWM_max_safe == 0)
{
    Drive_PWM = 0;
    return 0;
  }

  // Determine whether the 25 limit is enabled:
  if (ifdo_lim25 == true && wheel_speed > 25)
{// We are driving faster than 25 km/h, while we
should not, reduce support:
    Drive_PWM = max((Drive_PWM -
```

```
dPWM_max_decrease_25), 0);
  }
  else { // Write the PWM_drive, with the
limitation of the maximum increase of drive
support:
    Drive_PWM_recommend =
(cadence-cadence_minimum)*1.5 +
(wheel_speed-wheel_speed_minimum)*4;
    Drive_PWM = min(Drive_PWM +
dPWM_max_increase, Drive_PWM_recommend) ;
  }
  // Overwrite the drive PWM in case of any
Power reduction or disabling:
  Drive_PWM = min(Drive_PWM, PWM_max_safe);

  return Drive_PWM;
}
```