```
/*                                                                    [MX07].
 * Reads VTI's SCP1000-D01 (SPI) 30-120kPa pressure sensor and
 * generates a classic audio variometer "climb" signal, where the
 * pitch represents the (positive) vertical speed.
 *
 * Based on code from http://helpful.knobs-dials.com "SCP1000 pressure
 * sensor notes", and on the arduino.cc forums by a certain Conor.
 * This variometer specific adaption is by Martin Bergman
 * <bergman.martin at gmail dot com> 2011.
 *
 * Using the sensor's DRDY pin to interrupt the Arduino (via digital pin 2),
 * means we don't have to poll from the arduino side, and that we service
 * the sensor more immediately after it's done. Also for this application
 * we can ignore the sensor's temperature data.
 *
 * The sensitivity of the variometer algorithm, in terms of response lag
 * and noise rejection, may be tweaked with the constants a, slack and t.
 *
 * MOSI, MISO and SCLK are fixed when we use the AVR's hardware SPI but
 * it is still necessary to declare them as I/O pins in setup().
 */


#define SPI_SLAVESELECT  10
#define SPI_MOSI         11
#define SPI_MISO         12
#define SPI_CLOCK        13

// The SCP1000 register addresses that we use
#define SCP1000_REG_RSTR      0x06   //reset
#define SCP1000_REG_OPERATION 0x03   //Mode of operation
#define SCP1000_REG_STATUS    0x07   //ASIC Status
#define SCP1000_REG_DATARD8   0x1F   //MS 3 bits of 19-bit pressure
#define SCP1000_REG_DATARD16  0x20   //LS 16 bits of 19-bit pressure
#define SCP1000_OPERATION_HIGHSPEED 0x09   // Refreshing at ~9Hz

// Volatiles because they are updated from the interrupt function:
volatile boolean new_sample;
volatile byte temp_status;
volatile unsigned long pressure_raw; // 19 bits samples @ 0.25Pa resolution


/****************** Variometer specific variables and constants ********************/

int counter;             // Loop-counter for the start-up stabilizing procedure.
boolean gogo;            // Starter's flag. False until everything has settled.
const int typ = 20000;   // Dummy typical sensor value for faking an init input,
                         //   in order to ease initial EWMA adjustment.
const long snip = 380000;// Arbitrary value for reducing sampling magnitude.
long crpm;               // Latest sampled value (Cropped Raw Pressure Measurement).
                         //   The units are just 1/4 Pa, remember.
float s = typ;           // An intermediate variable for the smoothed moving average.
float s10 = typ*10;      // s magnified by 10 for fixed point precision when casting
                         //   to ints in later steps.
unsigned int ewma_state = typ; // The exponentially weighted moving average (EWMA) state itself.
unsigned int ewma_state10;// The "shadow" EWMA derived from s10, in effect corresponding
                          //  to the ewma_state value with a decimal.
unsigned int ewma_last;   // Value of ewma_state last time we looped.
unsigned int ewma_last10; // Value of ewma_state10 last time we looped.
const float a = 0.12;   // Alpha coefficient (a smoothing factor 0 < a < 1) used
                        //   for setting the filter's exponential decay. Adjust
                        //   this as needed: Less is more, i e output's lag and
                        //   "inertia" increases.(.07-.12).
const int slack = 3;    // Limits of the constrainment span for fending off sensor noise
                        //   spikes and other spurious input data. (Value of 3-6 is ok.)
int trendSet;           // Variable for dynamic adjustment of slack's neutral point.
int catchAll[5];        // Primary FIFO array for detection of genuine divergent trends.
int catchUp;            // Absolute value of the sum of items presently in catchAll.
int fleeting_state;     // Trend sensitive var for moving centre of constrainment span.
int d;                  // Delta of two consecutive ewma_state values. Unit is
                        //   still 0.25 Pa here.
int d10;                // Magnified delta value preserves resolution for last
                        //   step audio generation. Unit is now 0.025 (1:40) Pa.
int rw[5];              // Simplistic running window FIFO array for secondary
```

```
                        //   smoothing of jitter traces. An array size of 3 to 5
                        //   items is a reasonable compromise, we've found.
int vAppa;              // Apparent vertical velocity vector, being the average
                        //   of deltas presently in rw.
int vTrue;              // vAppa offset by sinkref to compensate for a typical
                        //   glider's reference sink rate, thereby reflecting the
                        //   true vertical vector of the surrounding airmass.
const int sinkref = 0 ; // Normal (best) sinkrate. E.g. a value of 3 equals a
                        //   reference sink rate of 30*(8.3 mm/4)*9 per second,
                        //   or .56 m/s (circa 110 fpm).  Unit is 0.025 Pa.
const int piezoPin = 9; // Hook up the beeper to digital pin 9.
const int f = 0;        // An offset from vTrue for adjusting pitch curve linearity.
                        //   (This is not essential for a limited expected range of,
                        //   say, 0-3 m/s. It's an obscure ploy anyway...).
const int t = 5;        // Threshold value for signalling climb. (Any sink is
                        //   quietly ignored). Unit is 0.025 Pa. (3 to 5 recommended)
int pitch;              // Proportional tone variable (Hz) mapped from vTrue.
const int dura = 75;    // Beep duration in millisecs. (Remember that the interrupts
                        //   occur every 111 ms.)
int onoff = 1;          // Toggling variable for restricting beeps to odd interrupt cycles.
/* ---------------------------------------*/
float pascalitos = 0.00;// for debugging (Pa units)
float metros = 0.00;    // for debugging (m units)
/* ---------------------------------------*/


void setup() {

  Serial.begin(9600);         // For monitoring

  pinMode(piezoPin, OUTPUT);  // Vario audio output
  counter = 0;
  gogo = false;

  // SPI init
  byte clr;                   //dummy variable
  pinMode(SPI_MOSI,        OUTPUT);
  pinMode(SPI_MISO,        INPUT );
  pinMode(SPI_CLOCK,       OUTPUT);
  pinMode(SPI_SLAVESELECT, OUTPUT);
  digitalWrite(SPI_SLAVESELECT, HIGH); // HIGH: do not select
  digitalWrite(SPI_CLOCK, HIGH);
  SPCR = B01010011;           // MPIE=0 (no interrupt enable)
                              //  SPE=1 (enable SPI)
                              // DORD=0 (MSB first)
                              // MSTR=1 (AVR is master)
                              // CPOL=0 (clock idle when low)
                              // CPHA=0 (samples on rising edge)
                              // SPR1=1 & SPR0=1 (250kHz)
  clr=SPSR;        // clear status register (by just reading it)
  clr=SPDR;        // clear data register (by just reading it)

/*  SCP1000 init procedure:
 *  Reset;
 *  Wait for startup (takes ~60ms because we explicitly wait that long);
 *  Set sampling mode      .                                          */

  scp_write_register(SCP1000_REG_RSTR, 0x01, SPI_SLAVESELECT);
  delay(60);
  while (scp_read_register8(SCP1000_REG_STATUS, SPI_SLAVESELECT)&B00000001)
delay(10);

  scp_write_register(SCP1000_REG_OPERATION, SCP1000_OPERATION_HIGHSPEED,
SPI_SLAVESELECT);


/* Use a rising edge on Arduino pin 2 (connected to the SCP1000's DRDY pin)
 *  to service the sensor via an interrupt.                          */

  pinMode(2,INPUT);
  attachInterrupt(0, service_scp1000, RISING); // arduino digital pin 2: interrupt 0

  new_sample = false;

```

```
  Serial.println(" ");
  Serial.println("            Vario SCP1000 version mx07\n");
  Serial.println("   Please wait 10 seconds while we settle down.");
  Serial.println("         -- Shall I put the kettle on?");
  Serial.println("_____________________________________________");
  Serial.print("         a: "); Serial.print(a); Serial.print("     slack: ");
  Serial.print(slack); Serial.print("     t: "); Serial.print(t); Serial.println("\n");
}


void loop() {

   // When the interrupt function marks that a new sample is available:
   if (new_sample) {
      new_sample = false;

   // if the fun has just begun: wait awhile...
      if (gogo == false){
        if (counter < 90){     // less than 10 seconds
          counter = counter + 1;
        } else {
          gogo = true;
        }
      } else {

   // now beep like a bat!
      if (onoff == 1){
        if (vTrue >= t){
          pitch = map((vTrue + f), 1, 160, 50, 1500); // 10, 160, 75, 1500
          tone(piezoPin, pitch, dura);
          onoff = 0;
        }
      } else {
        onoff = 1;
      }

   // Select various variables to print to screen for inspection:
/*-----------------------------------------------------------------*
        Serial.print(crpm);                             Serial.print("\t");
        Serial.print(ewma_state);                       Serial.print("\t");
        Serial.print("d: "); Serial.print(d);           Serial.print("\t");
        Serial.print(catchAll[4]);                      Serial.print("\t");
        Serial.print(vAppa);                            Serial.print("\t");
        Serial.print(vTrue);                            Serial.print("\t");
        Serial.print(pascalitos);                       Serial.print("\t");
        Serial.print("m/s: ");Serial.print(metros);     Serial.print("\t");
        Serial.print("tS: "); Serial.print(trendSet);
 *-----------------------------------------------------------------*/
        if (vTrue >= t){
          int j;
          Serial.print("  BEEP!  ");
//          for (j=0; j<vTrue-t+1; j++){   // (graphic aid for debugging: sloow)
//          Serial.print(".");}
        }
        Serial.print("\n");
      }
   }
}


void service_scp1000() {
/* Reads the sensor, sets mostly-raw values in global volatile variables, then does
the actual sensor data processing part. Theoretically we won't miss a DRDY when it
interrupts the uC, but it can't hurt to check */
    temp_status = scp_read_register8(SCP1000_REG_STATUS, SPI_SLAVESELECT);
    if (temp_status&B00010000) {
      // if either RTERR was set, clear it by reading data and skip real output:
      scp_read_register16(SCP1000_REG_DATARD16, SPI_SLAVESELECT);
    } else { // No error, go ahead and read.
      pressure_raw = ((unsigned long)scp_read_register8(SCP1000_REG_DATARD8, SPI_SLAVESELECT))
<<16;
      pressure_raw = pressure_raw|scp_read_register16(SCP1000_REG_DATARD16, SPI_SLAVESELECT);

      new_sample = true;
```

```
222
223
224  /********************** VARIO CODE GOES HERE: ***************************/
225
226        crpm = pressure_raw - snip;          // First we crop everything arbitrarily for con-
227                                             //   venience, effectively scratching 4-5 MSB's.
228
229        catchAll[0] = catchAll[1];           // Now nudge the array one tick...
230        catchAll[1] = catchAll[2];
231        catchAll[2] = catchAll[3];
232        catchAll[3] = catchAll[4];
233
234        if (abs(crpm - ewma_state) > slack){ // Is the latest crpm value off limits?
235          if (crpm > ewma_state) {           // If so: Is it an increase (aka descent)?
236            catchAll[4] = 1;                 // -- score it as such.
237          } else {                           // Or perhaps a decrease (aka ascent)?
238            catchAll[4] = -1;                // -- score it as such.
239          }
240          trendSet = (trendSet * 70 + (crpm - ewma_state) * 30) / 100; // (Keep tracking)
241        } else {                             // Nah, it's another middleoftheroader.
242          catchAll[4] = 0;                   // -- no score.
243        }
244
245        catchUp = abs(catchAll[0]+catchAll[1]+catchAll[2]+catchAll[3]+catchAll[4]);
246
247        if (catchUp !=4 && catchUp !=5) {
248  // Do the basic recursive filter EWMA routine, using a possibly constrained input value:
249          crpm         = constrain(crpm, (ewma_state - slack), (ewma_state + slack));
250          s            = crpm * a + s * (1 - a);    // s is a float, you'll recall.
251          s10          = s * 10;
252          ewma_state   = (int) s;
253          ewma_state10 = (int) s10;
254        } else {
255  // This means that none of the latest five samples has indicated an opposite direction
256  //   of movement, and that we _seem_ to be diverging from a stable altitude state.
257  // Oboy, we may have found a real trend here! Better update the EWMA with the latest
258  //   divergent value -- still constrained within the +/- span defined by "slack",
259  //   of course, but now centered on an "agile and totally trendy" neutral point:
260          fleeting_state = ewma_state + trendSet / 2;
261          crpm         = constrain(crpm, fleeting_state - slack, fleeting_state + slack);
262          s            = crpm * a + s * (1 - a);
263          s10          = s * 10;
264          ewma_state   = (int) s;
265          ewma_state10 = (int) s10;
266        }
267
268        d     = ewma_state - ewma_last;      // Latest pressure delta (0.25 Pa units)
269        d10   = ewma_state10 - ewma_last10;
270        rw[0] = rw[1];                       // "Nudge, nudge -- know what I mean?"
271        rw[1] = rw[2];
272        rw[2] = rw[3];
273        rw[3] = rw[4];
274        rw[4] = d10;                         // Note "magnification" by 10 (0.025 Pa)
275
276        vAppa = (rw[0]+rw[1]+rw[2]+rw[3]+rw[4])/5; // Crude and simple polisher.
277        vAppa = -1 * vAppa;          // Sign reversal: negative deltas mean up we go!
278        vTrue = vAppa + sinkref;     // Bias vAppa with the net vertical velocity.
279  /* ------------------------------------* /  // for debugging
280        pascalitos  = (float) vAppa/40;         // (i.e.  Pa units per 9 Hz sample)
281        metros      = pascalitos * .747;        // (i.e.  m units per sec) 83mm * 9 = 747mm
282  / * ------------------------------------*/
283        ewma_last   = ewma_state;
284        ewma_last10 = ewma_state10;
285      }
286  }
287
288
289  /********************** SPI helper functions ***************************/
290
291  char spi_transfer(volatile char data=0x00) { //0x00 is dummy data in case you want to receive
292      SPDR = data;                      // triggers send
293      while (!(SPSR & (1<<SPIF))) {}; // wait for transmission end
294      return SPDR;                      // return the received byte
295  }
```

```
296
297
298  /*  The SCP wants its addresses shifted <<2 , with the new
299      bit 0 always 0, and bit 1 controlling read(0)/write(1). */
300
301  char scp_read_register8(char register_address, byte select_pin) {
302      char in_byte;
303      register_address = register_address<<2;
304      digitalWrite(select_pin,LOW); //Select SPI Device
305      spi_transfer(register_address); //Write byte to device
306      in_byte = spi_transfer();
307      digitalWrite(select_pin,HIGH);
308      return in_byte;
309  }
310
311
312  unsigned int scp_read_register16(char register_address, byte select_pin) {
313      byte in_byte1,in_byte2;
314      unsigned int in_word;
315      register_address = register_address<<2;
316      digitalWrite(select_pin,LOW); //Select SPI Device
317      spi_transfer(register_address);
318      in_byte1 = spi_transfer();
319      in_byte2 = spi_transfer();
320      digitalWrite(select_pin,HIGH);
321      in_word = (in_byte1<<8)|in_byte2;
322      return in_word;
323  }
324
325
326  void scp_write_register(char register_address, char register_value, byte select_pin)
327  {
328      register_address = (register_address<<2)|B00000010;
329      digitalWrite(select_pin,LOW); //Select SPI device
330      spi_transfer(register_address); //Send register location
331      spi_transfer(register_value); //Send value to record into register
332      digitalWrite(select_pin,HIGH);
333  }
```