Arduino code: `arduino_thermometry.ino`

Motivation

We want to set up a chain of Arduino DUE units to read temperature sensors and pass their values back to a main computer.

Since there are twelve 12-bit analog inputs on each Arduino DUE, a set of ~100 temperature sensors (thermistors) will require ~10 devices in the chain.

Setup

The basic setup has one Arduino device connected to the main computer via a serial connection, while the Arduino devices themselves communicate via two-wire I^2C connections. Each device has the exact same code, with a boolean `isDev0` flag defaulted to `false`. Upon receipt of a serial command, this flag flips to `true` – indicating the device is Device 0 – and a message chain is initiated.

The Arduino DUE, unlike other models, has two separate I^2C ports, allowing two independent channels of communication. Because only one of these ports (`SDA` and `SCL`; pins 20 and 21) come equipped with pull-up resistors, that will be designated as the WRITE port. The other port (`SDA1` and `SCL1`; pins 70 and 71) will be designated as the READ port.

(Should pull-up resistors be desired for the READ port, connect the two lines in parallel to the Arduino's 3.3 V output through 4.7 kΩ resistors.)

Once the message chain has completed its circuit – i.e., readings have been taken from every device and returned to Device 0 – it is sent back to the main computer by Device 0 via serial, and the `isDev0` flag reverts to `false`. The cycle begins anew.

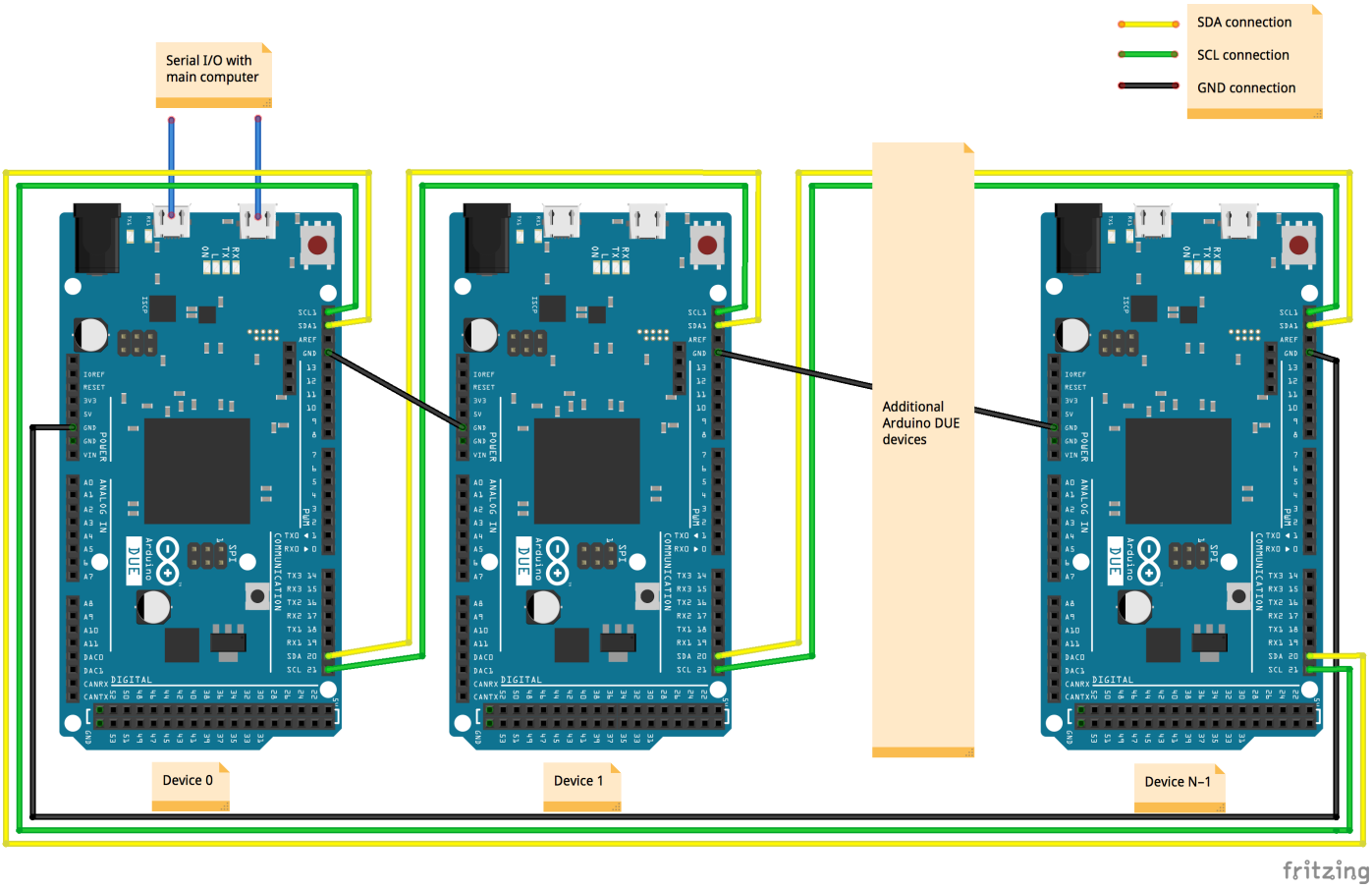

Figure 1: The chain of Arduino DUEs set up for message passing via I2C communication. The board communicating with the main computer via the serial port is designated Device 0.

I^{2}C communication

The `Wire` library contains a suite of functions useful for two-wire I^{2}C communication. See https://www.arduino.cc/en/Reference/Wire.

The library is installed (on a Mac) in `~/Library/Arduino15/packages/arduino/hardware/sam/<version>/libraries/`. For Arduino boards other than a DUE, `sam` is replaced by the board type.

Usage

N.B.: The `Wire` library functions mostly assume a master–slave setup, despite this not being a requirement for I^{2}C. Though somewhat restricting, the two separate I^{2}C ports can be set independently as master and slave. For our purposes, we initiate the WRITE port (pins 20 and 21) as master, and the READ port (pins 70 and 71) as a slave on **address 9**. (Because each master will ever only see one slave, all slaves can be set to the same address.)

The WRITE port (`SDA` and `SCL`; pins 20 and 21) are called using `Wire`, e.g., `Wire.begin()`.
The READ port (`SDA1` and `SCL1`; pins 70 and 71) are called using `Wire1`, e.g., `Wire1.begin(9)` – the `9` designating the slave address.

Clock speed

The Atmel SAM3X8E chip on the Arduino DUE is capable of communicating via I^2C at a clock speed up to 400 kHz (default is 100 kHz). In the `Wire` library, the clock speed is set by the function `Wire.setClock(400000);` (and the equivalent with `Wire1`).

The default speed was found to noticeably slow down the 500 Hz sampling.

Noise filtering

During testing, the 12-bit analog input was found to have an error of 3–4 counts (out of 4096), reducing the effective number of bits from 12 to 10 (i.e., the two lowest bits fluctuate too much to be useful). We could either lowpass-filter this with an analog or a digital filter. The latter was preferred because hardware calibration would add an additional source of error. However, the software solution requires there be sufficient CPU cycles.

Moving average filter

The easiest digital lowpass filter to implement is probably the moving average (boxcar) filter. A length-L boxcar filter has an impulse response given by

$$h[n] = \sum_{i=0}^{L-1} b_i \cdot \delta[n-i], \quad b_i \equiv \frac{1}{L}$$

and a frequency response of

$$H(\omega) = \sum_{m=-\infty}^{+\infty} h[m]\, e^{-i\omega m} = \frac{1}{L} \cdot \frac{1 - e^{-i\omega L}}{1 - e^{-i\omega}} \,.$$

It is therefore a comb filter with notches evenly spaced at frequencies $f = n/L$, where n is any non-zero integer.

Application

Since we don't expect temperatures to change drastically within a second, we only want a sample once every, say, 200 ms (5 Hz). If we simply sampled at this rate, our standard variance would be 3–4 counts as before. We can do much better by applying the boxcar filter: By sampling 100 times faster (every 2 ms; 500 Hz), we'd have 100 samples every 200 ms; the average of those 100 samples has 10 times lower noise than before – the standard deviation of means decrease as $\sim 1/\sqrt{L}$. The limit to which we can do this is determined entirely by the CPU cycles available.

Without invoking (hardware) interrupts, the Arduino DUE was found to be capable of reliably sampling at 500 Hz (2000 μs) using the following implementation.

We initiate a (circular) buffer of sufficient length and two pointers to demarcate a boxcar of length L. The sum of the values within the boxcar is stored in a separate array (one value for each channel). We store the sum rather than the average because division is expensive, and known constant factors can always be handled offline (basically a unit conversion). For maximum efficiency, the sum is updated by subtracting the value at the end of the boxcar (i.e., the element at `sub_ptr`) and adding the new value at the front of the boxcar (i.e., the element at `add_ptr`).

The value in the sum array is the value to be read out. The rate at which this happens depends on the desired filter and is handled by the main computer. Using the values below and a constant voltage source (a DC power supply), the standard deviation of the output values was found to be ~0.2 counts (out of 4096).

To do

There a still a few things to clean up before the system can be deployed, but they aren't too difficult – I just ran out of time.

Serial input

We envision the main computer to be the one requesting readouts via USB (serial port) every 200 ms (5 Hz).

Current setup: Device 0 sends initiates the data chain every 200 ms
To do: the `isDev0` flag should be triggered by the request from serial; default value = `false`; every time data is returned to the main computer, `isDev0` should revert to `false`

Serial output

There is a buffer of 128 bytes. Sending all 12 channels as binary should work (at least for two devices), but definitely not ASCII.

The buffer size is defined in `~/Library/Arduino15/packages/arduino/hardware/sam/<version>/cores/arduino/Ringbuffer.h`, on the line `#define SERIAL_BUFFER_SIZE 128`. The buffer size need not be changed if the communications was handled smartly. Keep in mind that there is limited onboard memory on the Arduino. Think about how to separate the message packets (if necessary).

Timer overflow

The onboard timer is a key component of the code, responsible for determining the sample rate. Because it is stored as a 32-bit integer, overflow is inevitable. **This is a crucial check to perform: make sure everything still works as planned with the overflow.**

From the Arduino documentation, `micros()` overflows in about 70 minutes, whereas `millis()` overflows in about 50 days.

Code

Below is the code in its entirety.

```
#include <Wire.h>

// constants
```

```
const int dt = 2000; // sampling rate [microseconds]
const int Dt = 200;  // readout rate [milliseconds]
const int N = 250;   // buffer length
const int L = 200;   // boxcar length
const byte Nch = 12;  // number of channels per device
const byte Ndev = 10; // number of devices in chain

unsigned long sum[Nch] = {0};    // sums from current board
unsigned long rcvsum[Ndev][Nch] = {0}; // sums received from another board

bool isDev0 = true; // is (not) device connected via serial

void setup() {
  unsigned short add_ptr = 0;
  unsigned short sub_ptr = add_ptr + N - L;
  unsigned long old_micros = 0, new_micros = 0;
  unsigned long old_millis = 0, new_millis = 0;
  unsigned short buffdata[N][Nch] = {0};
  unsigned short old_voltage = 0, new_voltage = 0;

  Serial.begin(250000);
  analogReadResolution(12);
  analogWriteResolution(12);
  // analog in pins on the DUE run from pins 54 to 65 (total 12 pins)
  for (byte i=54; i<65; i++) {
    pinMode(i, INPUT);
  }

  // start the first I2C bus as master
  Wire.begin();
  Wire.setClock(400000);

  // begin I2C bus on address 9
  Wire1.begin(9);
  Wire1.setClock(400000);
  Wire1.onReceive(receiveEvent);

  while (true) {
    // take samples every dt microseconds
    new_micros = micros();
    if ((new_micros - old_micros) >= dt) {
      for (byte j=0; j<Nch; j++) {
        new_voltage = analogRead(54+j);
        old_voltage = buffdata[sub_ptr][j];
        buffdata[add_ptr][j] = new_voltage;
        sum[j] += new_voltage - old_voltage;
      }

      old_micros = new_micros;
      (add_ptr==N-1) ? add_ptr = 0 : add_ptr++;
      (sub_ptr==N-1) ? sub_ptr = 0 : sub_ptr++;
```

```
    }

    // if Device 0, send out sums every Dt MILLIseconds
    // TO DO: message chain initialized by serial request; once
    // once that is set up, this shouldn't be timer-based
    if (isDev0) {
      new_millis = millis();
      if ((new_millis - old_millis) >= Dt) {
        Wire.beginTransmission(9);
        Wire.write(1);
        Wire.write(1);
        for (byte i=0; i<Nch; i++) {
          for (byte j=0; j<3; j++) {
            Wire.write(sum[i] >> (8*j));
          }
        }
        Wire.endTransmission();
        old_millis = new_millis;
      }
    }
  }
}

void receiveEvent(int Nbytes) {
  unsigned short packetsTotal = 0;
  unsigned short packetNumber = 0;
  unsigned int x = 0;

  while (Wire1.available()) {
    // first two bytes are meta-data
    packetsTotal = Wire1.read();
    packetNumber = Wire1.read();

    // fill the buffer with the received packets
    for (byte i=0; i<Nch; i++) {
      x = 0;
      for (byte j=0; j<3; j++) {
        x = (Wire1.read() << (8*j)) | x;
      }
      rcvsum[packetNumber-1][i] = x;
    }
  }

  // if Device 0, output to serial once all packets received;
  // else pass along message on `Wire`
  if (isDev0) {
    if (packetNumber == packetsTotal) {
      for (byte k=0; k<packetsTotal; k++) {
        Serial.print(k+1);
        // TO DO: fix serial output; currently limited by buffer size
//        for (byte i=0; i<Nch; i++) {
```

```
      for (byte i=0; i<8; i++) {
        Serial.print(" ");
        Serial.print(rcvsum[k][i]);
      }
      Serial.println();
    }
   }
  } else {
   // send the values received from other devices
   // have to break into one packet per device due to 32-byte limit
   for (byte k=0; k<packetsTotal; k++) {
    Wire.beginTransmission(9);
    Wire.write(packetsTotal+1);
    Wire.write(k+1);
    for (byte i=0; i<Nch; i++) {
      for (byte j=0; j<3; j++) {
        Wire.write(rcvsum[k][i] >> (8*j));
      }
    }
    Wire.endTransmission();
   }

   // send the values on current device
   Wire.beginTransmission(9);
   Wire.write(packetsTotal+1);
   Wire.write(packetsTotal+1);
   for (byte i=0; i<Nch; i++) {
    for (byte j=0; j<3; j++) {
      Wire.write(sum[i] >> (8*j));
    }
   }
   Wire.endTransmission();
  }
}

void loop() {
  // empty because the loop is in setup()
}

```

Demonstration

To test our setup, we have two Arduino DUE devices in the chain. A thermistor is connected to Device 1 via a 1-meter long wire on each end. While no detectors are connected to Device 0 for now, it requests a readout every 200 ms (5 Hz) and sends the data back to the main computer.

Our circuit was set up as a voltage divider, with the thermistor as the first resistor; $V_{\text{out}} \propto 1/R_1$. See Figure 2.

The thermistor used is Digi-key #490-7170-ND (Murata Electronics NXR-T-15-XM-202-E-A-1-B-040). For

specs, see https://www.digikey.ca/product-detail/en/murata-electronics-north-america/NXRT15XM202EA1B040/490-7170-ND/3900401. The resistance is 2 kΩ at 25°C and decreases exponentially with temperature, $R(T) \propto \exp(1/T)$. **N.B.:** one consequence of this scaling is that the sensitivity (dynamic range) is reduced at high temperatures, i.e., a large change in T will register only a little change in R.

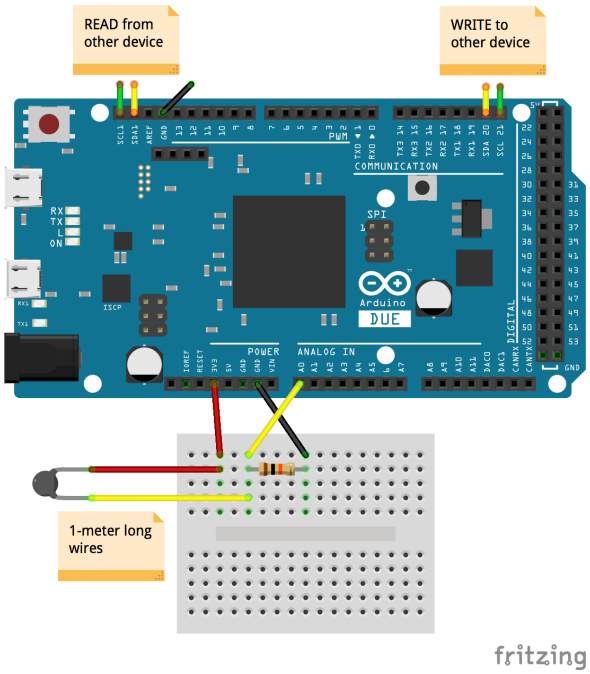

Figure 2: A simple thermometer using a thermistor in a voltage divider setup. Shown here is Device 1, which only communicates via I2C.

Cases

We produced 2-minute (600-sample) timestreams for each of the four cases:

1. Untwisted wire, thermistor exposed
2. Untwisted wire, thermistor in a box
3. Twisted wire, thermistor exposed
4. Twisted wire, thermistor in a box

The timestreams are shown in Figure 3.

When the thermistor is exposed (cases 1 and 3), the readings are subject to shifting air currents in the laboratory (result of AC, people walking by, etc.) and the associated temperature changes. Placing the thermistor in a box (cases 2 and 4) stabilizes the temperature quite noticeably; the temperature inside the unventilated box was also likely higher.

The difference in mean temperature between cases 2 and 4 (~0.2°C) is within expectation for fluctuations temperature inside the box – the box had to be opened to retrieve the thermistor for case 3.

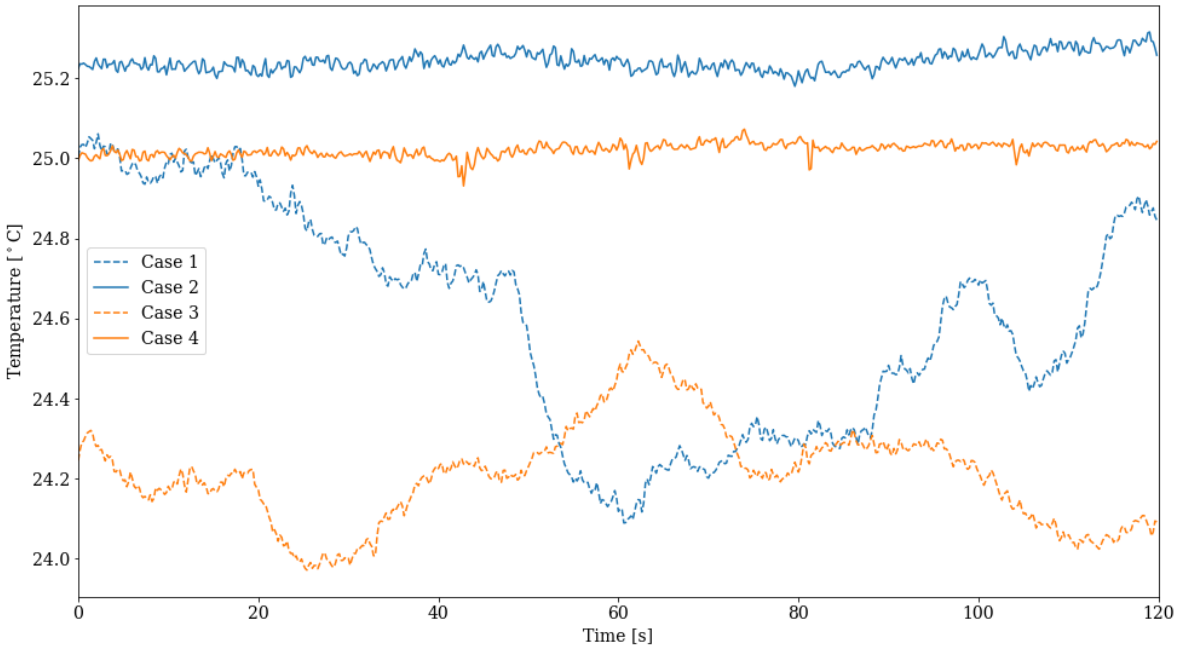

Figure 3: The four timestreams. As expected, the exposed thermistor (cases 1 and 3) show much greater variability in its readings due to shifting air currents in the laboratory.

The standard deviation (rms) of each timestream in Figure 3 are:

- Case 1: 0.279°C
- Case 2: 0.023°C
- Case 3: 0.125°C
- Case 4: 0.016°C