Herbert Jaeger

ALPhA's Laboratory Immersions

Harnessing the Power of Arduino for the Advanced Lab

(Final Version)

ALPhA Immersion Workshop July 27 – 29, 2017

Department of Physics
Indiana University – Purdue University
Ft. Wayne, Indiana

ARDUINO CONTROLS THE TEMPERATURE

Objectives

1. To construct an electronic thermometer using a thermistor as the sensor and the Arduino as the microcontroller.

2. To use the thermistor thermometer as the feedback element in a temperature controller.

Background

The resistance of a metal wire changes with temperature. Knowing how the resistance changes with temperature, the wire can be used to measure temperature by relating temperature to resistance. Platinum wire is particularly useful as a temperature-sensitive element in a thermometer because it is very stable and has a high melting point of around 1770˚C. However, thermometers made from metals have the disadvantage of being rather insensitive. For example, the resistance of a platinum wire changes by about 30% for a 100˚C temperature change.

The resistance of a semiconductor such as the silicon used in a solid state diode or a transistor changes appreciably, even with small temperature changes. For this reason, semiconductors make sensitive temperature sensors. A device employing a semiconducting material as a temperature sensor is called a *thermistor*. The resistance of a commercial thermistor may change by a factor of 100 if its temperature changes by 100˚C.

From theoretical considerations and experience it is known that the resistance $R_{thermistor}$ of a thermistor is related to the absoulte temperature, T, by

$$R_{thermistor} = A\, e^{\frac{B}{T}} \qquad (1)$$

where A and B are constants characteristic of a given thermistor. The circuit shown in Figure 1 is used to relate the thermistor resistance to a voltage for the Arduino microcontroller. The voltage at the output of the first op-amp is related to the potentiometer voltage by

$$V_{out} = -\frac{R_{thermistor}}{R} V_{pot} \qquad (2)$$

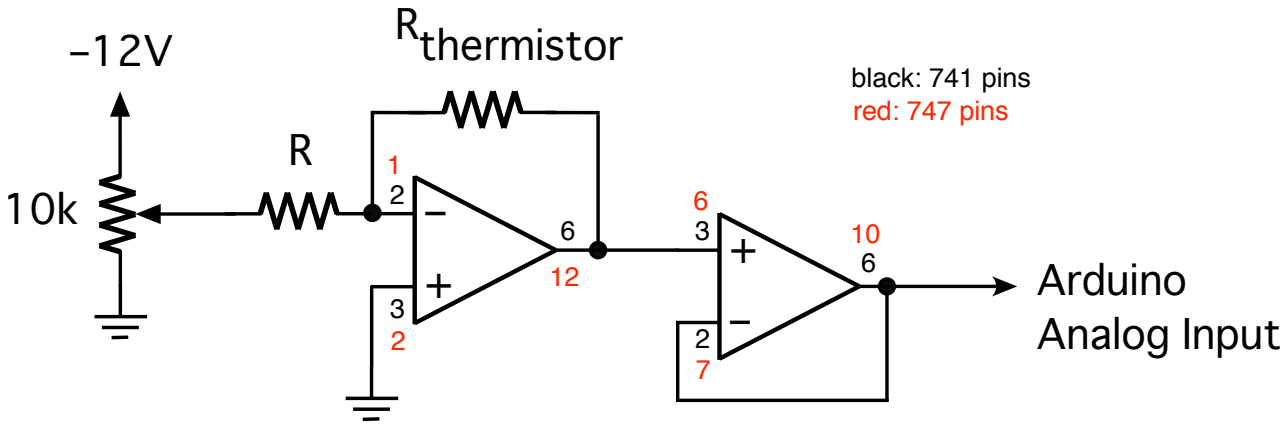


Figure 1: Circuit diagram for the thermistor.

Since the Arduino's analog input can only handle postive voltages, V_{pot} must be a negative voltage. The exact value of V_{pot} will depend on the termistor resistance and your choice of the value for R. For a given choice of R and a constant adjustment of the pot output voltage is proportional to the thermistor resistance, $R_{thermistor}$ in the op-amp's feedback. Here the op-amp functions as an ohmmeter. The second op-amp acts as a voltage follower and is used to avoid loading the analog input of the Arduino.

The resistance of the thermistor decreases as temperature increases. Accordingly, the magnitude of the output voltage of the circuit decreases as temperature increases. In this exercise we are interested in temperatures between room temperature and 100°C. If we choose R equal to the thermistor resistance at room temperature then the magnitude of the output at temperatures above room temperature will always be less than V_{pot}.

The thermistor resistance is given by $R = Ae^{B/T}$ and the output voltage is proportional to the thermistor resistance. Accordingly, the output voltage can be written

$$V_{out} = Ce^{\frac{B}{T}} \qquad (3)$$

The constants C and B must be determined by calibration. Then the absolute temperature can be calculated upon measuring V_{out}.

Calibration of the thermistor thermometer

Quality control in the manufacture of thermistors is such that all thermistors of a given type will not have the same values of C and B. It is up to the user to determine C and B experimentally. In principle, two measurements of voltage and temperature suffice to determine C and B. A more satisfactory technique is to record measurements of voltage and temperature in the temperature range of interest and fit the theoretical expression to the data. Fitting is greatly

simplified if the data are described by a straight line relationship. A graph of V_{out} versus T for the thermistor function is not a straight line but we can "linearize" the exponential relationship between V_O and T by taking the natural logarithm of both sides of the equation

$$\ln V_{out} = \ln C + \frac{B}{T} \qquad (4)$$

Performing measurements of V_{out} and T and plotting $\ln V_{out}$ versus $\frac{1}{T}$ will produce a straight line with slope B and intercept $\ln C$. You will use the linear regression function (LINEST) of Excel to determine slope, intercept, and their uncertainties.

The thermistor is sealed in the bottom of a glass or plastic tube about 1/4" in diameter and 10" long. Electrical connections are made to the thermistor by flexible leads entering the open end of the tube. Connect the thermistor to the circuit as shown in Figure 1. Choose R and V_{pot} to produce an output voltage just under 5 V at room temperature. Since the resistance of the thermistor decreases with increasing temperature, this will be the maximum voltage. In this exercise you will be using a temperature range from room temperature (about 20˚C) to about 100˚C. Before you connect the output ot the Arduino test your circuit by touching the thermistor. As it warms up the output voltage of your circuit should drop.

After you convinced yourself that your circuit works the next step is to calibrate the thermistor by determining the relationship between voltage and temperature. Place a thermometer and the thermistor probe into a beaker of water and place the beaker on a hot plate. Make a series of measurements of temperature and output voltage, V_{out}, from room temperature to the boiling point of water. It is very important to stir the water during the calibration, and to have the thermistor and thermometer tied together so you actually know that thermistor and thermometer are exposed to the same temperature. Record the measurements in a spreadsheet as you proceed. In the spreadsheet, plot $ln\ V_{out}$ versus $1/T$, where T is the absolute temperature. Perform a linear fit (LINEST) to the data and record the slope and intercept; you will need this for the temperature controller in the next part.

Temperature Controller

Suppose you had a jar of water on a stove, a thermometer to measure the water temperature, and wanted to maintain the temperature at 60 ± 2˚C. You would turn the stove on, watch the thermometer, and shut it off when the temperature reads close to 60˚C. When the water cools and the temperature falls below 60˚C, you would turn the stove on. In this exercise a microcontroller (the Arduino) will “watch” the temperature via the thermistor and will turn a heater on or off in order to maintain a constant water temperature.

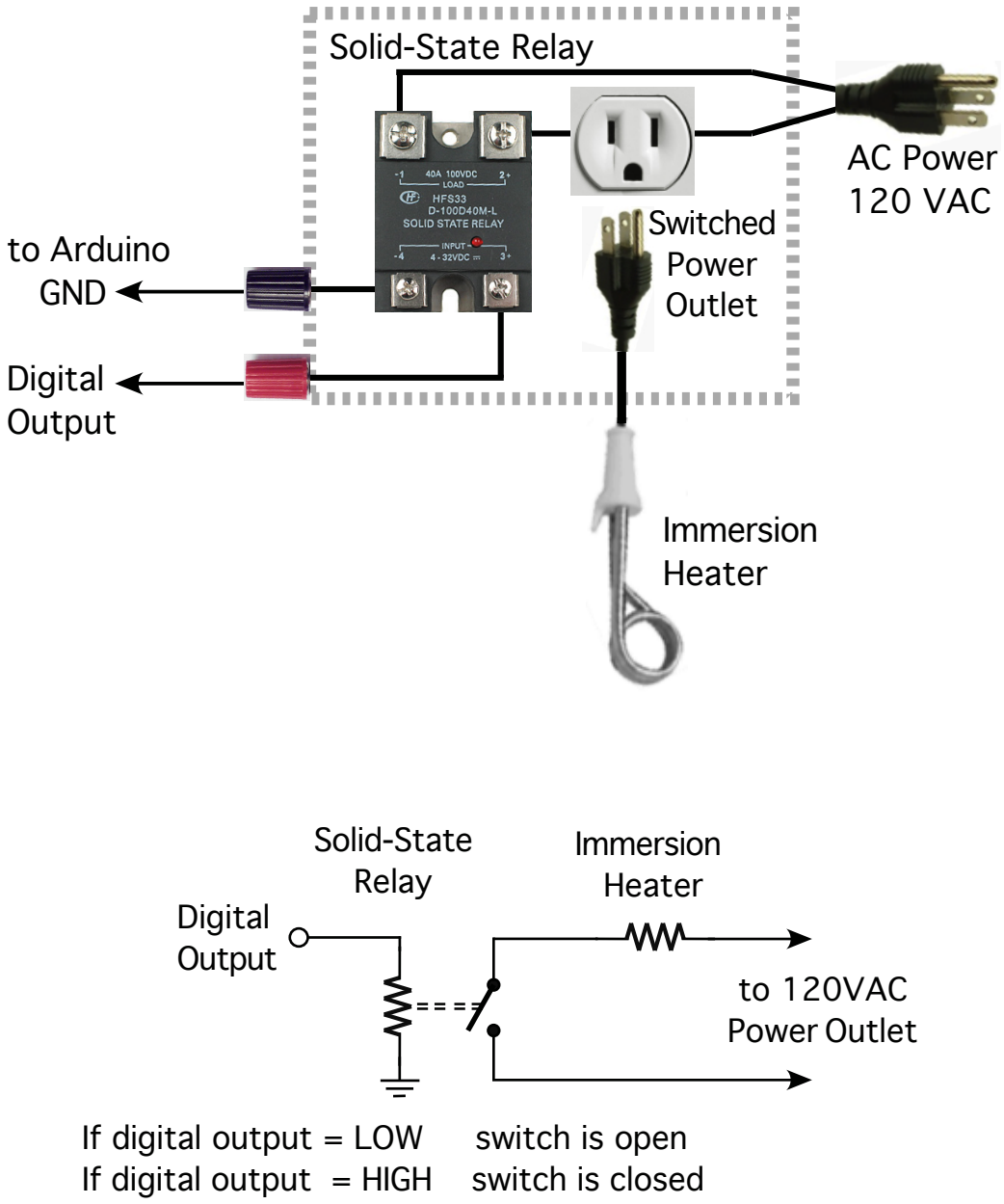


Figure 2: The switch circuit for the temperature controller. Component diagram (top) and equivalent circuit diagram (bottom). The components inside the dashed box are in the blue box with the outlet on your lab station.

The actual switching is done by a relay. A relay is an electro-mechanical switch that is operated by applying a small voltage to the input terminals. A current flows through a coil and attracts an iron lever which in turn closes an electrical switch. We will be using a similar device called a *solid-state relay*; it performs the same action, but entirely without any moving parts, just via semiconductor electronics. The solid-state relay has two terminals to which the voltage is applied and two terminals that constitute the switch. For example, when 3 to 5 V is applied to the input the switch is ON, i.e. the "switch contacts" are closed. If 0 to 0.5 V is applied the switch is OFF, i.e. the "switch contacts" are open. The control voltage is DC, so the polarity of the applied voltage matters greatly. Figure 2 shows the switch circuitry. With the circuit in Figure 1 still connected to the Arduino, we now have to connect the digital output of the Arduino to the switch circuit of Figure 2. The digital output is connected to the red terminal and the Arduino's ground to the black terminal. When the output goes HIGH, the relay switch closes and turns on the immersion heater. When the digital output goes LOW, the switch opens and the immersion heater is turned off.

Next you need to program a sketch to have the Arduino monitor the thermistor and decide if the relay switch needs to be on or off. Here are the steps to you to program:

- Read a pot via analog input (the setpoint, T_{set})
- Read the thermistor via analog input (the water temperature, T)
- Compare setpoint T_{set} and water temperature T
- If $T < T_{set}$ set digital output HIGH
- If $T > T_{set}$ set digital output LOW
- Repeat

As part of the sketch you will need to convert the output voltage to an actual temperature. It is for this step that you will need the calibration constants *C* and *B* from the previous section.

Now that you have all your ducks in a row, it is time to see if your temperature controller works as advertised. Plug the immersion heater into the switched outlet (the one in the blue box on your lab station), and place thermistor and immersion heater in a beaker of water. Be careful that the thermistor does not touch the heater directly. You also need to add the thermometer so you can see if the controller maintains the correct temperature. Once the relay turns the heater on, the temperature will rise. When it reaches the setpoint temperature the controller will turn the heater off. Be patient, it will take some time for the water temperature to react to the heater. After 10-20 minutes you should have some sort of equilibrium. Carefully observe and note how high and low the temperature gets as the controller cycles on and off.

Arduino Programming Cheat Sheet

Primary source: Arduino Language Reference
http://arduino.cc/en/Reference/

Structure & Flow

Basic Program Structure

```
void setup() {
  // runs once when sketch starts
}
void loop() {
  // runs repeatedly
}
```

Control Structures

```
if (x < 5) { ... } else { ... }
while (x < 5) { ... }
do { ... } while ( x < 5);
for (int i = 0; i < 10; i++) { ... }
break;  // exit a loop immediately
continue;  // go to next iteration
switch (myVar) {
   case 1:
     ...
     break;
   case 2:
     ...
     break;
   default:
     ...
}
return x; // just return; for voids
```

Operators

General Operators

```
=  (assignment operator)
+  (add)          -  (subtract)
*  (multiply)     /  (divide)
%  (modulo)
== (equal to)     != (not equal to)
<  (less than)    >  (greater than)
<= (less than or equal to)
>= (greater than or equal to)
&& (and)    || (or)    ! (not)
```

Compound Operators

```
++ (increment)
-- (decrement)
+= (compound addition)
-= (compound substraction)
*= (compound multiplication)
/= (compound division)
&= (compound bitwise and)
|= (compound bitwise or)
```

Bitwise Operators

```
&  (bitwise and)    |  (bitwise or)
^  (bitwise xor)    ~  (bitwise not)
<< (shift left)    >> (shift right)
```

Variables, Arrays, and Data

Data types

```
void
boolean   (0, 1, true, false)
char      (e.g. 'a'  -128 to 127)
int       (-32768 to 32767)
long      (-2147483648 to 2147483647)
unsigned char  (0 to 255)
byte           (0 to 255)
unsigned int   (0 to 65535)
word           (0 to 65535)
unsigned long  (0 to 4294967295)
float     (-3.4028e+38 to 3.4028e+38)
double    (currently same as float)
```

Qualifiers

```
static    (persists between calls)
volatile  (in RAM (nice for ISR))
const     (make read only)
PROGMEM   (in flash)
```

Arrays

```
int myInts[6]; // array of 6 ints
int myPins[]={2, 4, 8, 3, 6};
int mySensVals[6]={2, 4, -8, 3, 2};
myInts[0]=42;  // assigning first
               // index of myInts
myInts[6]=12;  // ERROR! Indexes
               // are 0 though 5
```

Constants

```
HIGH  | LOW
INPUT | OUTPUT
true  | false
143         (Decimal)
0173        (Octal - base 8)
0b11011111  (Binary)
0x7B        (Hexadecimal - base 16)
7U          (force unsigned)
10L         (force long)
15UL        (force long unsigned)
10.0        (force floating point)
2.4e5       (2.4*10^5 = 240000)
```

Pointer Access

```
&  (reference: get a pointer)
*  (dereference: follow a pointer)
```

Strings

```
char S1[8] =
 {'A','r','d','u','i','n','o'};
 // unterminated string; may crash
char S2[8] =
 {'A','r','d','u','i','n','o','\0'};
 // includes \0 null termination
char S3[]="Arduino";
char S4[8]="Arduino";
```

Built-in Functions

Pin Input/Output

```
Digital I/O (pins: 0-13 A0-A5)
  pinMode(pin,[INPUT, OUTPUT])
  int digitalread(pin)
  digitalWrite(pin, value)
    // Write HIGH to an input to
    // enable pull-up resistors
Analog In (pins: 0-5)
  int analogRead(pin)
  analogReference(
    [DEFAULT, INTERNAL, EXTERNAL])
PWM Out (pins: 3 5 6 9 10 11)
  analogWrite(pin, value)
```

Advanced I/O

```
tone(pin, freqhz)
tone(pin, freqhz, duration_ms)
noTone(pin)
shiftOut(dataPin, clockPin,
  [MSBFIRST,LSBFIRST], value)
unsigned long pulseIn(pin,
  [HIGH,LOW])
```

Time

```
unsigned long millis()
  // overflows at 50 days
unsigned long micros()
  // overflows at 70 minutes
delay(msec)
delayMicroseconds(usec)
```

Math

```
min(x, y)    max(x, y)    abs(x)
sin(rad)     cos(rad)     tan(rad)
sqrt(x)      pow(base, exponent)
constrain(x, minval, maxval)
map(val, fromL, fromH, toL, toH)
```

Random Numbers

```
randomSeed(seed) // long or int
long random(max)
long random(min, max)
```

Bits and Bytes

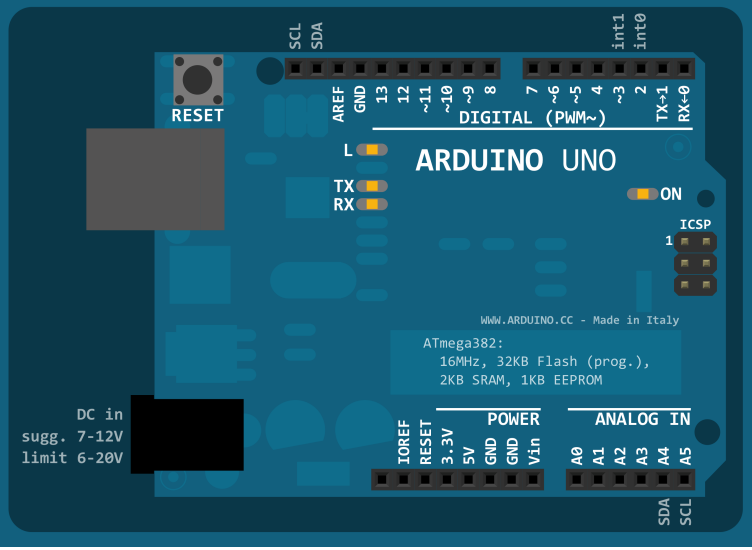
```
lowByte(x)    highByte(x)
bitRead(x, bitn)
bitWrite(x, bitn, bit)
bitSet(x, bitn)
bitClear(x, bitn)
bit(bitn)  // bitn: 0=LSB 7=MSB
```

Type Conversions

```
char()        byte()
int()         word()
long()        float()
```

External Interrupts

```
attachInterrupt(interrupt, func,
 [LOW, CHANGE, RISING, FALLING])
detachInterrupt(interrupt)
interrupts()
noInterrupts()
```



Libraries

Serial (communicate with PC or via RX/TX)

```
begin(long Speed) // up to 115200
end()
int available() // #bytes available
byte read() // -1 if none available
byte peek()
flush()
print(myData)
println(myData)
write(myBytes)
SerialEvent() // called if data rdy
```

SoftwareSerial (serial comm. on any pins)

```
  (#include <softwareSerial.h>)
SoftwareSerial(rxPin, txPin)
begin(long Speed) // up to 115200
listen()       // Only 1 can listen
isListening() // at a time.
read, peek, print, println, write
  // all like in Serial library
```

EEPROM (#include <EEPROM.h>)

```
byte read(intAddr)
write(intAddr, myByte)
```

Servo (#include <Servo.h>)

```
attach(pin, [min_uS, max_uS])
write(angle)  // 0 to 180
writeMicroseconds(uS)
   // 1000-2000; 1500 is midpoint
int read()    // 0 to 180
bool attached()
detach()
```

Wire (I²C comm.) (#include <Wire.h>)

```
begin()      // join a master
begin(addr) // join a slave @ addr
requestFrom(address, count)
beginTransmission(addr) // Step 1
send(myByte)            // Step 2
send(char * mystring)
send(byte * data, size)
endTransmission()       // Step 3
int available() // #bytes available
byte receive()  // get next byte
onReceive(handler)
onRequest(handler)
```


by Mark Liffiton

Adapted from:
- Original by Gavin Smith
- SVG version by Frederic Dufourg
- Arduino board drawing
  original by Fritzing.org