Department of Mechanical, Materials and Manufacturing Engineering

The University of Nottingham

Computer Engineering and Mechatronics MMME3085

Laboratory Exercise 1: Overview and Preparatory Programming Assignment

1 Introduction

The overall objectives of this laboratory are:

- To provide practical experience of interfacing a variety of signal sources and sensors to the Arduino Mega 2560 using various peripheral boards
- To give further experience of using the Arduino's dialect of the C language
- To provide experience of the interfacing of a servo motor and incremental encoder using an H bridge, the LS7366R up/down counter and the Arduino itself
- To enable students to see in practice the waveforms associated with pulse width modulation and quadrature encoding.
- To give experience of the use of finite state machines and of the use of interrupts

This laboratory is in five parts:

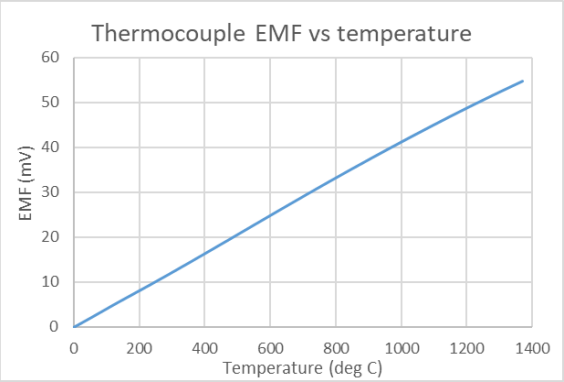
- **Interfacing to different sensors:** you will construct for yourselves the software interfaces to two different types of analogue sensors: a thermistor and a thermocouple (which you will also use in conjunction with the thermistor).
- **PWM and H bridges:** this introduces the use of the L298 module to drive a servomotor. This uses a timer-counter to generate the variable-width pulses which drive the motor with varying effective voltages. You will use an oscilloscope to view the resulting waveform, and see how it varies as the motor speed is altered.
- **Quadrature decoded:** this uses the same motor control program, but additionally involves viewing the quadrature pulses from the encoder which provides the position feedback information from the servomotor. Eventually you will get to try out your own quadrature decoder program with the actual quadrature signal, and compare the results you get with those from a hardware decoder. You will also see how different approaches to calling the decoder behave when time is in short supply.

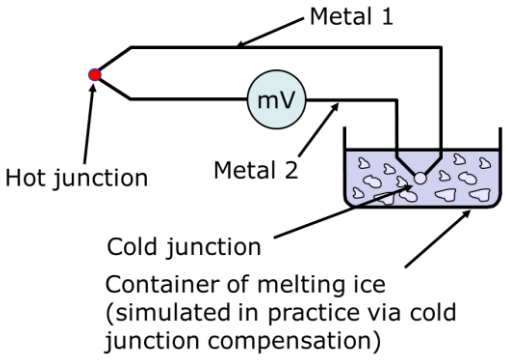
2 Background

This laboratory introduces three types of sensor, two analogue and one digital.

a) The thermistor is a resistor whose resistance changes with temperature. Those used for temperature measurement have a negative temperature coefficient (their resistance reduces with temperature). They are well suited to measuring room temperature and are used in things like electronic thermometers.

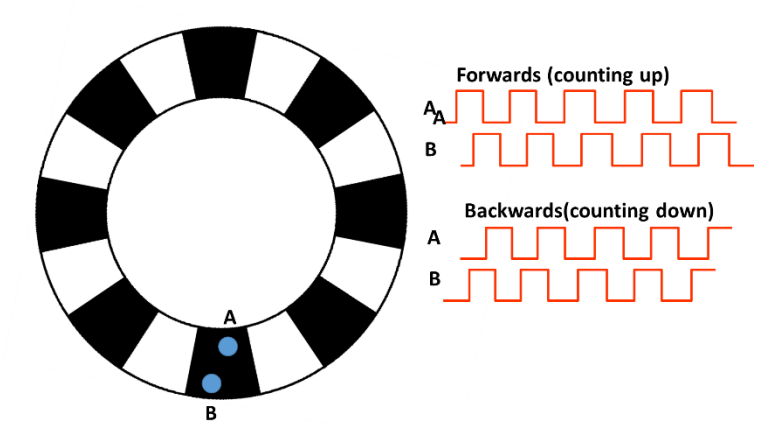
b) The thermocouple consists of a pair of wires of dissimilar materials (the one you will use is K-type, made from two alloys called chromel and alumel. The thermocouple produces a small EMF (a few millivolts) which is approximately proportional to the temperature difference between the temperatures of the two junctions of the wires. In principle one of these junctions is held at 0°C and the other is used for measuring temperature.





As it is inconvenient to maintain the "cold" junction at 0°C, this temperature is simulated by adding to the actual EMF a value of EMF associated with the cold junction's true temperature, measured otherwise (for example using a thermistor). This is known as cold junction compensation. The calibration curves relating EMF to temperature, and temperature to EMF, are typically given by special functions approximated as polynomials ("NIST polynomials").

c) The incremental rotary encoder, which can take various forms but is typically a circular disc with slots, through which light shines onto two detectors (called Channel A and Channel B). These are aligned so that as the disc rotates, it causes the light detectors to be activated in such a way that they generate a pair of square waves which are a quarter of a wave (90°) out of step with each other. These waves are said to be in quadrature. The sequence in which the signals from the two detectors go high and low indicates the direction in which the disc is rotating.



A special form of counter (an up-down counter or quadrature decoder) is required to count the pulses so that the total number of pulses increments or decrements as required, thus indicating the overall angle through which the encoder has rotated. Incremental rotary encoders are widely used in conjunction with motors to determine how far the motor has rotated.

Two of the sensors considered here (thermistor and thermocouple) produce their output in analogue form i.e. as a varying voltage related, either linearly or nonlinearly, to the quantity they are measuring (temperature). In order to take readings of these values using a computer, it is necessary to convert them to digital form. You have already covered various analog-to-digital converters in the module Electromechanical Devices (MMME2051). The Arduino includes such a device which can be switched (multiplexed) to various analogue inputs. This particular ADC has a resolution of 10 bits, meaning that its output is a 10-bit binary number in the range 0 to 1023; thus it is important to note that the function **analogRead()** does NOT produce a value equal to the voltage input being digitised, but an integer number which must be converted to the voltage.

3 Apparatus

This laboratory is based around the Arduino Mega 2560, which is essentially a user-friendly board on which is mounted the AVR Atmega2560 microcontroller chip. You will be using it with:

- The Mikroelektronika Counter Click board, which is effectively a carrier board for the LS7366R up/down counter which is interfaced to the Arduino using the SPI serial protocol
- The Grove High Temperature Sensor board which includes a thermistor (temperature-dependent resistor) which can be used for measuring changes in ambient temperature, but also includes an interface to a K-type thermocouple which can be used for measuring temperatures in the range −200 °C to +1350 °C provided the thermistor is used for cold junction compensation. The board also includes signal conditioning and amplification circuits.
- A DC motor with optical encoder, used for producing (and measuring) mechanical rotation.

- A L298 H-bridge circuit mounted on a board, used for switching the motor on and off rapidly in forward and reverse directions.
- A variable voltage benchtop power supply with current limiter.

4 Preparatory exercises

4.1 Interfacing to different sensors

Start by creating a program to test the formulae, constants etc. that you will use in your Arduino program. There is a program, **TwoSensorsSkeleton.c**, on Moodle which gives a starting point. Copy this into a folder in VSCode.

There are two functions already defined for use in the thermocouple code, described in section d below.

Use **`scanf`** to read in two values between 0 and 1023 to simulate the value being read from the ADC for each of the inputs A0 and A1 on the Arduino. The thermocouple is input from A0 and the thermistor from A1. (In the Arduino program used in the lab these values will be read using the analogRead function.)

For input values of 256 you should get the following results: thermistor temperature = 14.33°C, thermocouple temperature =416.97°C. If you don't get these, check your formulae.

You will need to write some code to interface the different sensors, coding up the conversions relating to the following situations. Hint: as good programing practice, use constants defined at the start of your program to represent numeric values (e.g. pin numbers, V_{ref}) to avoid the use of numbers in formulae or elsewhere where there meaning is obscure (often known as "magic numbers").

a) Converting ADC value to voltage

In each case the sensor produces a voltage in the range 0-5 V which is converted via the Arduino's built-in analog-to-digital converter (ADC) to a number in the range 0-1023. It is this integer value which is returned when you call **`analogRead`**. According to the Atmega2560 datasheet:

$$V_{ADC} = \frac{n_{ADC} \times V_{ref}}{1024} \qquad (1)$$

where V_{ref}=5V and n_{ADC} is the numeric value from the ADC in the range 0-1023. For example, when the value returned by the function call **`analogRead(A0)`** is 256 the voltage on pin A0 is 1.25 V.

Write a function which converts the ADC value to the voltage as you will need to perform this task twice and you don't want duplicate code.

b) Thermistor

It can be shown that the resistance *R* of a thermistor is related to the absolute temperature *T* by the following equation:

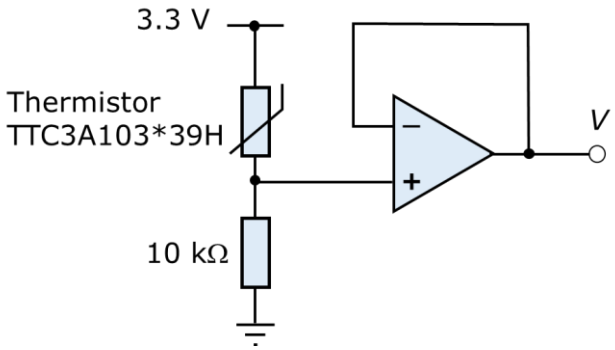
$$R = R_0 e^{B\left(\frac{1}{T}-\frac{1}{T_0}\right)} \qquad (2)$$

where T_0 is 298.15 K (i.e. 25 °C) and in the present case R_0 (the resistance at 25 °C in kΩ) is 10 kΩ and *B* is 3975 K. This expression can be rearranged to give the temperature in K in terms of the resistance:

$$T = \left(\frac{1}{T_0} + \frac{1}{B}\ln\left(\frac{R}{R_0}\right)\right)^{-1} \qquad (3)$$

So, to obtain the temperature in °C you will need to subtract 273.15 K.

The signal conditioning circuit for the thermistor is shown below (simplified from the Seeed Grove High Temperature Sensor documentation):



The resistance can be related to the output voltage *V* as follows:

$$R\,(\mathrm{k\Omega}) = \frac{10 \times 3.3}{V} - 10 \qquad (4)$$

Write code to calculate the thermistor temperature in degrees C, given an ADC input value between 0 and 1023.

- (i) Convert the ADC value to voltage using the function created in the previous section.
- (ii) Convert this voltage to resistance using equation (4)
- (iii) Convert the resistance to temperature in K
- (iv) Write a function to convert degrees K to C
- (v) Convert the temperature in K to C using the function

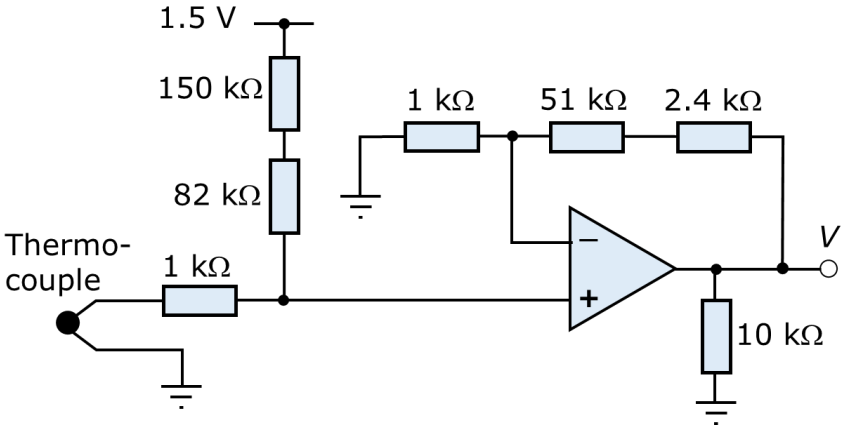
These kinds of calculations are typical if you are coding up your own interfacing code.

c) Thermocouple

The EMF (voltage) given by a thermocouple is approximately proportional to the temperature difference between the "hot" junction (where temperature is being measured) and the "cold" junction, which ideally is held at 0 °C. In practice of course the cold junction is at room temperature, so the actual room temperature must be measured independently and used for compensating the temperature measurement from the thermocouple.

In practice it is not quite as straightforward as might be imagined.

1. The thermocouple EMF is too small to be measured directly and could be outside the range of the ADC, so some signal conditioning circuitry is needed.



From the circuit diagram[1] above (again greatly simplified from the Grove documentation) it can be concluded that the thermocouple voltage E_{TC} is related to the circuit output voltage V (which is the ADC input voltage for the thermocouple) as follows:

$$E_{TC} = \frac{(V - 0.35)}{54.4} \qquad (5)$$

This is the "raw" thermocouple voltage before any cold junction compensation.

2. To compensate for the actual temperature of the cold junction (which is of course actually at room temperature T_{RT}, a compensation voltage must be added by converting the actual cold junction temperature to the EMF (in millivolts) which would arise from a thermocouple with its "hot" junction at room temperature and its "cold" junction at 0°C. This compensation EMF E_{comp} can be calculated using the (very complicated!) "forward" function which relates EMF in millivolts to room temperature $\underline{T}_{RT}$ in °C:

$$E_{comp} = F_{forward}(T_{RT}) \qquad (6)$$

[1] The manufacturer's schematic gives the very slightly different amplifier gain of of 54.15; use the value 54.4 as that is obtainable using the operational amplifier calculations you already know.

The actual temperature of the hot junction can then be calculated using the "inverse" function which relates temperature to (compensated) EMF:

$$T_{TC} = F_{inverse}(E_{TC} + E_{comp}) \qquad (7)$$

So in order to calculate the actual temperature of the hot junction the temperature of the cold junction must first be found using the thermistor (as above), making sure that this is in °C. This is then converted into a compensation EMF, this is added to the thermocouple EMF (which in turn is found from the ADC value using equation (1) and the ADC voltage using equation (6)) and is finally converted to temperature using equation (7).

In practice, all these calculations would be performed within ready-written software such as the software library[2] provided with the Grove board. It is however instructive to perform these calculations for yourselves. To make life much easier for you, the forward and inverse functions (used in equations (6) and (7) above) are given to you in the file **TwoSensorsSkeleton.c** (and also in the corresponding .ino file) in the following form:

```
float NISTdegCtoMilliVoltsKtype(float tempDegC) // returns
EMF in millivolts: forward TC function
float NISTmilliVoltsToDegCKtype(float tcEMFmV)  // returns
temp in degC assuming 0 degC cold jcn: inverse TC function
```

Write code to calculate the temperature at the hot junction in degrees C, given an ADC input value between 0 and 1023.

- (i) Convert the ADC value to voltage using the function created earlier
- (ii) Calculate the thermocouple voltage using equation (5)
- (iii) Convert the thermistor temperature calculated earlier into a compensation EMF using equation (6)
- (iv) Calculate the temperature using equation (7)

[2] Please don't be tempted just to copy the code from the Grove library, if you can find it. That code does things a significantly different way from that proposed here; it gives more or less the same answers but I'd prefer you to do it "my way". And I don't want you to be committing plagiarism!

Programming tips

Experience shows that this exercise holds significant traps for the unwary and inexperienced programmer, notably the "integer divide" trap (which, for example would evaluate 1/2 as 0 as it has a zero integer part). These traps can be avoided as follows:

- Be very careful with units – don't confuse °C with K, or mV with V. Remember to make the necessary conversions.
- Never divide by an integer e.g. don't put use an expression such as `30/5*V` as it will not evaluate correctly; `30.0/5.0*V` will be OK. Even better, define constants as (for example) `const float Vref=5.0;` which provides a double safety net to avoid such a variable being treated as an integer and will force you to think about the data type. Defining them as integers, or just using integers within the formulae where these constants appear, will lead to incorrect results.
- Remember that serial output statements on the Arduino are quite unlike `printf()` and format descriptors simply don't work in `Serial.print()` or `Serial.println()`. Use `printf()` in the C example. When transferring to the .ino Arduino program there are some strong hints on how to present the serial output in the skeleton file.

When your c program is working and giving the correct output for the test values save it as **TwoSensorsXXX.c** where **XXX** is your initials. Save a screenshot of the terminal window output with test values of 256.

Next transfer your tested functions and calculations into the **TwoSensorsSkeleton.ino** file. This gives a good framework for a timed loop as well as an infrastructure for displaying your results.

Use the following inputs with the `analogRead()` function to obtain the input values (replacing the scanf statements that you had in your c test code):

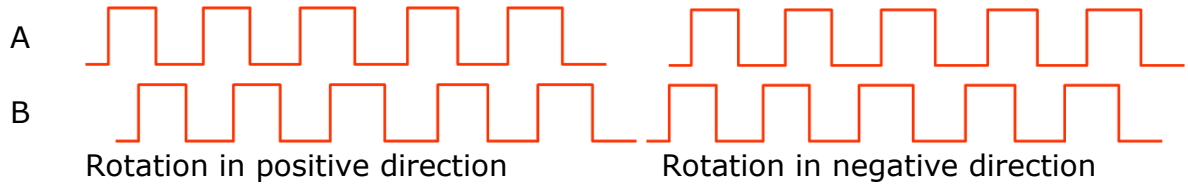
Thermistor: Analog input 1 (pin A1)
Thermocouple: Analog input 0 (pin A0)

- You can test the program with the Arduino by temporarily replacing the `analogRead()` statements with a hard-coded value of 256 and checking that you get the results above(but don't forget to reinstate the `analogRead()` statements when you have finished!).

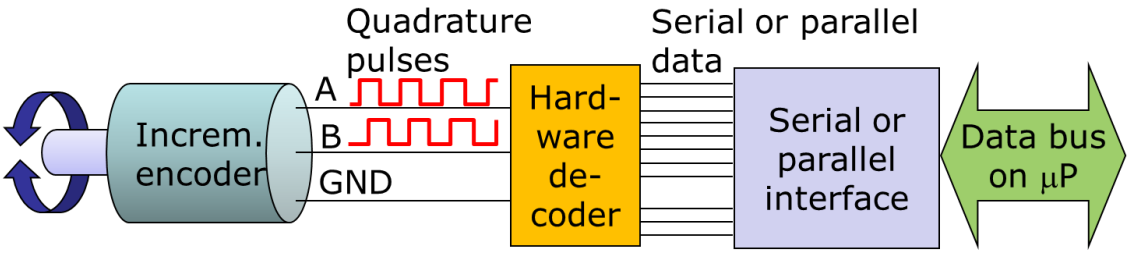
When you are comfortable that your coding is correct (and with the `analogRead()` functions re-inserted!) , make sure the correct version is present in the Arduino code, and save it as **TwoSensorsXXX.ino** where **XXX** is your initials.

4.2 Quadrature Decoded: a simple application of a state machine architecture

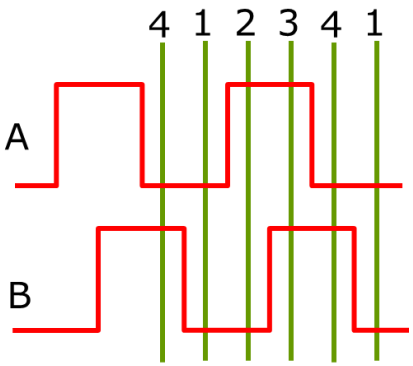
The incremental encoder described earlier is a method of measuring angular rotation, and generates a pair of trains of pulse which are 90° out of phase with each other. This is known as a quadrature signal, and allows the direction to be determined from the sequence of rising and falling edges of the waveforms on the two channels A and B:



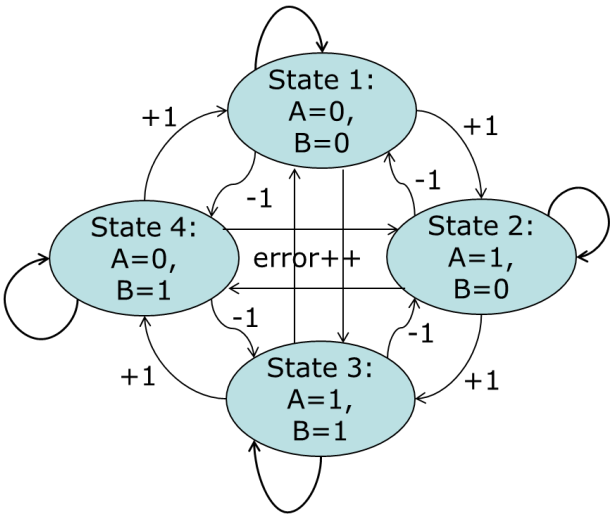
Interpreting these signals is not just a matter of counting the pulses – the direction must be taken into account. One approach to this is to use a specialist counter chip as will be demonstrated during the laboratory.



However, an alternative approach is to use a program based on a "finite state machine" (usually known simply as a "state machine"), which considers the different states (in this case, the four possible combinations of channels A and B being high (1) and low (0)) and the transitions from one state to the next These can be represented in a state transition diagram, which describes how each transition can correspond to either incrementing or decrementing the counter giving the position relative to the starting point.



Timing diagram showing the four different states



State transition diagram showing how the transitions from state to state relate to incrementing or decrementing the counter and error counter

A typical state machine program in C involves a continuously-repeating loop in which there is a **`switch`** statement with four **`cases`**: each corresponds to the one possible current state of the system, and contains the decision-making process which decides on the action (incrementing or decrementing the counter and the error count) and the new state arising from the newly-read inputs from the encoder. For example, if we are currently in state 1 then read the inputs on A and B, the newly-read inputs are interpreted as follows:

- If A is 0 and B is 0, do nothing and stay in State 1
- If A is 1 and B is 0, add 1 to main counter and go to State 2
- If A is 0 and B is 1, subtract 1 to main counter and go to State 4
- If A is 1 and B is 1, do nothing to main counter but add 1 to error counter and go to state 3

We also need to initialise the state:

- If initially A is 0 and B is 0, system starts in State 1
- If initially A is 1 and B is 0, system starts in State 2
- If initially A is 1 and B is 1, system starts in State 3
- If initially A is 0 and B is 1, system starts in State 4

We will cover state machines and a typical state machine program structure in some detail in the lectures – for the present time, concentrate on the form of the signal and the logic required to interpret it.

Two different versions of a test program are available for you to implement and test your logic.

- If you own an Arduino Uno or Mega, use **TestEncoder.ino** which enables you to enter the values for Channel A and Channel B via the serial monitor, and view the count (and error count).
- If you don't own an Arduino, use **TestEncoder.c** which will run in VSCode. The details of the program structure are different but the functions for initialising and updating the state machine are the same in each program. Be warned that to maintain compatibility with the Arduino program structure, some poor programming practice (notably the use of global variables) has had to be implemented. As noted previously, don't do this in your main C programming exercises!

Before you start to write the code for the state machine draw flowcharts for the `initialiseEncoderStateMachine()` and `updateEncoderStateMachine()` functions to show the logic and program flow required. As `state`, `channelAState` and `channelBState` variables are global variables in the Arduino program you do not need to show initialisation of these in the flowcharts. Files showing the flowcharts for the part of the code shown in the skeleton files are available in Moodle. **InitialiseEncoderStateMachineSkeleton.drawio and UpdateEncoderStateMachineSkeleton.drawio** can be loaded into drawio and used as a starting point for the full flowcharts. Pdf versions of the files are also available on Moodle. Save the finished flowcharts as a single pdf file in the form **StateMachineFlowchartXXX.pdf** where **XXX** are your initials.

To give you a helping hand, the enumerated constants and the integer constants have already been defined for you in both versions of the test program.

The text description of the first state is given, along with part of the implementation. You are on your own for the other states! Use your flowchart to inform how you program the state machine.

When you have finished, **please test the code** by running it and manually setting the A and B inputs to high and low by typing in the appropriate pairs of digits e.g. 00, 10, 11, 01, 00. You should be able to simulate the sequence involved in positive and negative movement by typing the appropriate pairs of digits in and checking that the correct transitions have been made. Try some error situations as well, e.g. 00 followed by 11. Save your work as **TestEncoderXXX.ino** or **.c** as appropriate, where **XXX** are your initials**.**
Save a screenshot of the terminal window output with the following test data: 00, 10, 11, 01, 11, 00, 99. (Note that the '99' to terminate is only required if using the .c skeleton. It is not required if you have written the code straight into the .ino file).

If you are confident that you have got the coding correct, copy the logic of your test program into the Arduino sketch **MotorEncoderSkeleton.ino** and save it as **MotorEncoderXXX.ino.** Be careful that your final version of the function **`updateEncoderStateMachine`** includes the lines of code to read the inputs:

```
channelAState = digitalRead(channelA);
channelBState = digitalRead(channelB);
```

and that you have not accidentally deleted them!

You will need to bring your programs to the laboratory.

Finally, please create a zip file containing your work (programs, terminal window output and flowcharts) as **Lab1PrepXXX.zip** and submit it online via Moodle no later than 3pm on Thursday 26th October 2023.

Sources of information:

Grove High Temperature Sensor documentation (including schematics)
http://wiki.seeedstudio.com/Grove-High_Temperature_Sensor/

Atmega2560 data sheet
http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf

NIST thermocouple data
https://srdata.nist.gov/its90/download/type_k.tab