ECE471: Embedded Systems – Homework 9

Temperature Display

Due: Friday, 21 November 2025, 5:00pm

1. You will need the i2c display from Homework 5 as well as *either* the MCP3008/TMP36 from Homework 7 *or* the DS18B20 sensor from Homework 8 (your choice).

   Upon completing this assignment you can turn back in the parts you have borrowed.

   Be sure to check for errors and to comment your code!

2. **Make your temperature code modular (2pts)**

   Take one of your temperature reading homeworks as a basis for this project. Copy your code into `read_temp.c`

   The interface needed, as described in `read_temp.h` is

   ```
   double read_temp(void);
   ```

   So have your code read the temperature, and return it as a double value in degrees C.

   If there is an error, report a temperature of less than -1000 degrees.

   The provided `test_temp` program uses this interface, so once your code is ready, running `make` should build `test_temp` and it should print the temperature when you run it.

3. **Make your display code modular (3pts)**

   Take your i2c display code and put it into the `write_display.c` file.

   The interface it should have is described in `write_display.h` is:

   ```
   int init_display(void);
   int write_display(int fd,double value);
   int shutdown_display(int fd);
   ```

   The `init_display()` function should init the display (including turning it on, brightness, etc) and return the file descriptor.

   The `write_display()` function should take the file descriptor and the temperature value and put that onto the display. (See below for more details).

   The `shutdown_display()` function should take the file descriptor and close it, as well as any other cleanup that needs to be done (which might be none, depending on how your code works).

   The `write_display()` function should print the provided floating point temperature value (in F or C, your choice).

   Your code should handle four cases:

   (a) Temperatures from 0 to 99.9 degrees, inclusive. $0.0 \leq temp \leq 99.9$ These should be displayed as two digits, a decimal point, another digit, and then a degree symbol (which is just a crude circle made of the top 4 segments on the display). For example: "24.5°"
   Leading zeros should be suppressed (i.e. display "2.4°" not "02.4°", 0 should be "0.0°")

(b) Temperatures between -99.9 and 0 degrees. $-99.9 \leq temp < 0$ These should display a minus sign and then two digits of temperature, then the degree symbol. For example: "-25°"
For temperatures between 0 and -9.9 be sure to print two digits of result (with decimal point). For example, "-2.5°"

(c) Temperatures between 100 and 999 degrees. $100 \leq temp \leq 999$ should print three digits of temperature, then the degree symbol. For example: "245°"

(d) Invalid temperatures that won't fit the display (and errors reading the thermometer) should be reported (via the display) in a method that isn't a valid temperature. It is your choice how to indicate this.

Once you have the code working, you can use the provided `test_display` program to test that it is printing things properly. It takes one command line argument, which is the floating point value to print.

4. **Testing the display output (1pt)**

   Use the `test_display` program to test various inputs and be sure they meet the standards as described.

   In the README, list 5 test values you used that cover as many cases in the specification as possible, and write a brief note for why you chose those values / why you think they cover the full functionality of the interface.

5. **Temperature Display (1pt)**

   Now modify `display_temp.c` to be a program that uses the interface above to reads the temperature once a second and writes the value to the display.

6. **Something Cool**

   No something cool for this homework. Put any coolness to use in your final project.

7. **Questions (1pt)**

   Edit the `README` file to have your name and answer the following questions. (Note we might not cover this in class until next week).

   (a) Name one example of poorly written embedded code that had disastrous results.

   (b) Why might it be good to always try to write correct, documented, well tested code even if you think it's not going to ever be used in anything important?

8. **Linux Fun (2pts)**

   Do the following on your Raspberry Pi.

   When a file is created or modified on Linux various timestamps are updated. atime (last access time) mtime (last modified time) and ctime (last attribute update).

   The `ls -lt` (that's a lowercase L) will show all files and their last modified time.

   The Linux `touch` command will update the timestamps on a file to the current time (and create the file if it doesn't exist). You can also specify the time. You can do things like

```
touch --date "1983-10-16 14:40" blah
```

which will update the timestamp on the file `blah` to the specified date.
You can also do fun things like

```
touch --date "next Thursday" blah
```

(a) Use touch to change the file modification time of the "fakedate" file (included with the test code) with a date from some other year (not 2025).

(b) What happens if you try to create a date in the year 2044?
Note, on a Pi running a 64-bit OS you can create a year 2044 file just fine. However on a 32-bit system you might get an error like the following:

```
touch: invalid date format '2044-10-16 14:40'
```

Why might creating a year 2044 date on a 32-bit Linux system not work?

(c) You forget to turn in a homework before the deadline, but send a screenshot showing the last-modified timestamp was before the deadline to the professor. Why might this not be the most convincing argument?

9. **Submitting your work**

- Run `make submit` which will create a `hw9_submit.tar.gz` file containing `Makefile`, `README`, `display_temp.c`, and `fakedate`. You can verify the contents with `tar -tzvf hw9_submit.tar.gz`
- e-mail the `hw9_submit.tar.gz` file to me by the homework deadline. Be sure to send the proper file!