CSE 121: Lab 3 Report

By Daniel Beltran

10/29/2023

Lab 3: Part 1

In this portion of the lab, like many others, it was my first time soldering. My main focus was understanding the process of soldering and how to use the equipment. Using the link below, provided by the lab instructions, I was able to understand the concept of soldering (using flux), how to deoxidize the solder iron, and dead joints. As we were told to connect ports SDA, SCL, GND, and VCC, connecting to SDA led to a dead joint. The dead joint was dry, meaning that the pin wasn't soldered to the board correctly. Fortunately for me, I was able to catch the mistake when connecting it to the DFRobot LCD and properly soldered my board for the final time.

How to Solder:

https://www.bareconductive.com/blogs/resources/how-to-solder-headers-onto-the-touch-board

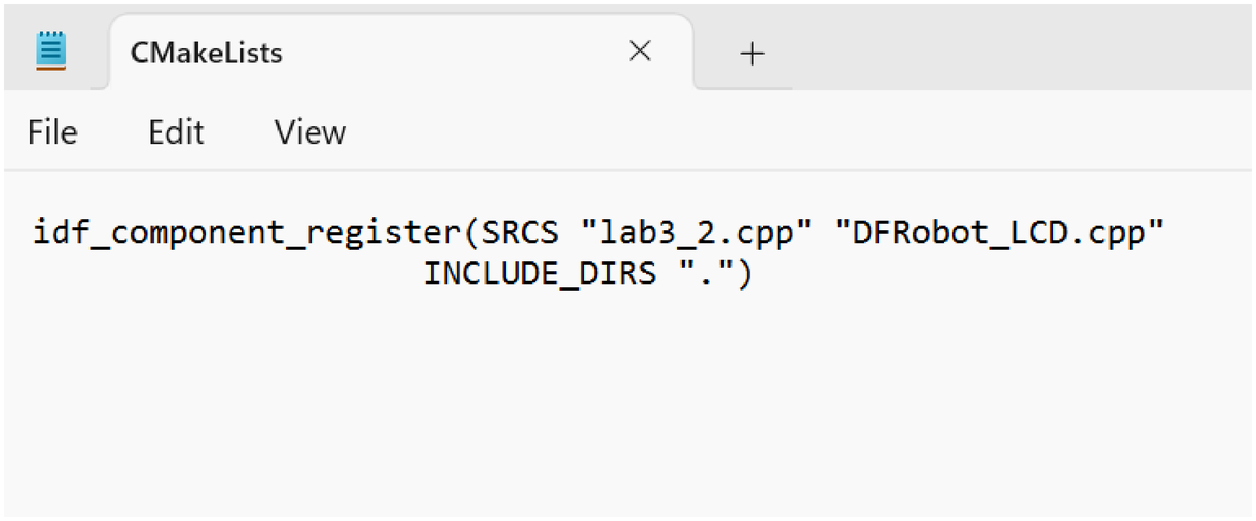
Lab 3: Part 2

The second part of the lab required setting the environment for our LCD screen, provided by the DFRobot_LCD, to print out **Hello CSE 121!** and my last name, **Beltran**. Part two proved to be quite challenging and was the bulk of the work as it was quite difficult understanding how to navigate files. The following is the github linked provided to use to clone on our raspberry pi:

DFRobot_LCD GitHub Source: https://github.com/DFRobotdl/DFRobot_LCD

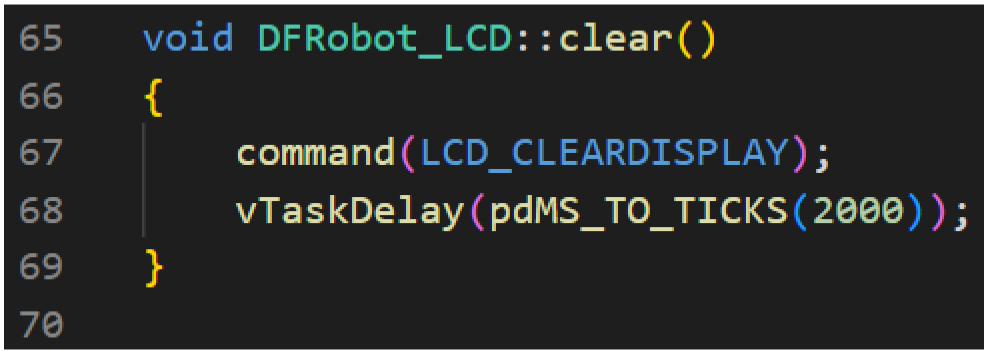
The first few steps consisted of removing the imported header files, used in DFRobot_LCD.cpp to output on an LCD for an Arduino. The following header files that needed to be removed were: Wire.h, Print.h, and Arduino.h. In the header file, DFRobot_LCD.h, it was more of the same concept of removing parts of the Arduino files like **using Print::write** and **#include "Print.h"**.

Creating the lab3_2 project was just like before but with a few alterations. With our newly altered DFRobot_LCD files, copied them over to the lab directory and made changes to the **CMakeLists.txt** file to properly compile two different files (since our ESP32C3 served as the master and DFRobot_LCD served as the slave) and adding them as shown below:



```
idf_component_register(SRCS "lab3_2.cpp" "DFRobot_LCD.cpp"
                    INCLUDE_DIRS ".")
```

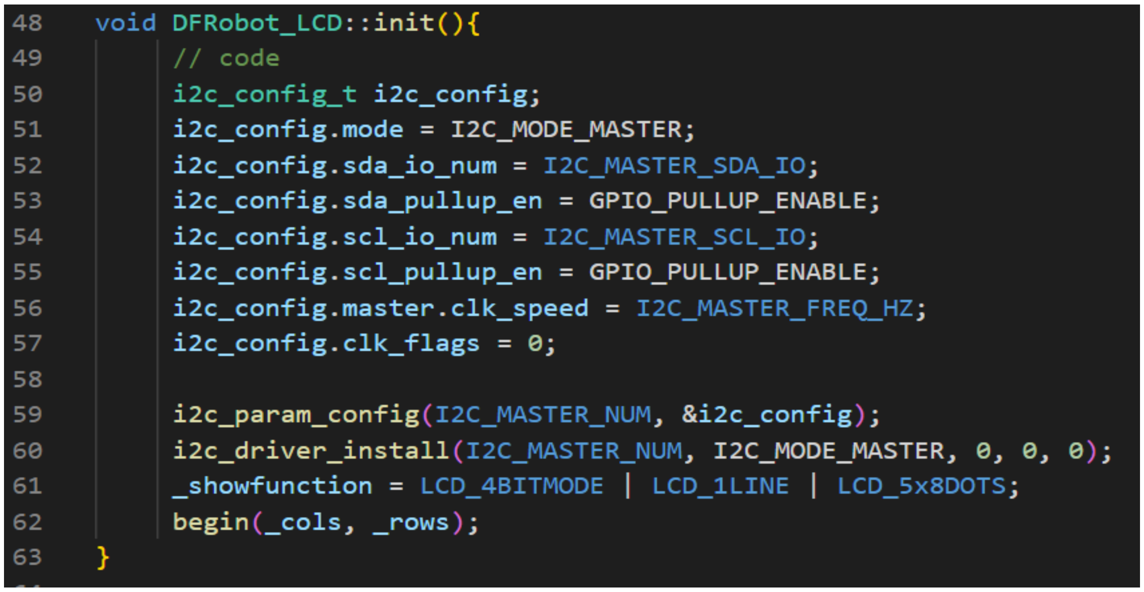
After this, DFRobot_LCD.cpp needed a few more logical alterations as the previous code only worked with an Arduino program. One of the first things that I noticed was that in many functions, **delayMicrosecond();** and I've never seen this beforehand. Looking at the link below, it was an Arduino function call so I had changed all of those function calls to:



```
void DFRobot_LCD::clear()
{
    command(LCD_CLEARDISPLAY);
    vTaskDelay(pdMS_TO_TICKS(2000));
}
```

From here, it was a lot of confusion on where the logic needed to be altered. Though for this lab, we really cared about **printstr()** since it called on Arduino files, **send()** since it called on Wire.h, **setReg()** for the same reasons, and also **init()** to initialize. These functions were the only ones that were dependent on Arduino programs for its operation, so they were the functions that needed to be converted to the I2C protocol which we've been using.

To start off, init() was the first function that needed to be altered but a lot was reused from lab2_2.c and can be seen below. From here, there was additional logic needed as compiling with this code as it would error out on line 60. The link below goes over the initialization with the use of a master (esp32c3) in detail but also indicates that **.clk_flags** must be added to this. Now looking at **_showfunction** it was used for the initialization to display the contents onto the LCD and **begin()** is a function that starts displaying the LCD by going through the rows/columns individually to print out. Essentially, it helps the display on the LCD which requires it to go one-by-one in rows and columns.



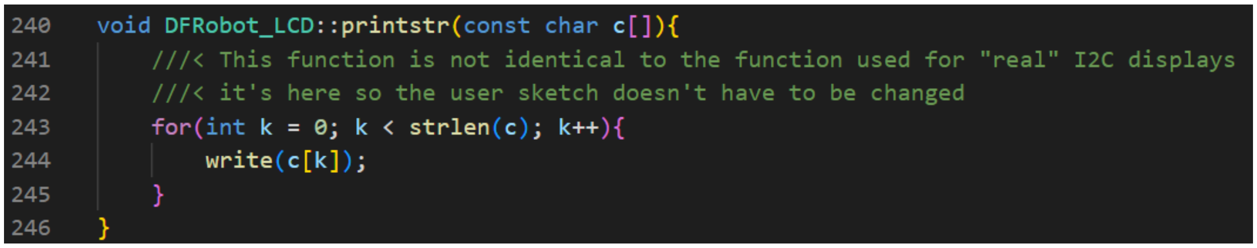
```
void DFRobot_LCD::init(){
    // code
    i2c_config_t i2c_config;
    i2c_config.mode = I2C_MODE_MASTER;
    i2c_config.sda_io_num = I2C_MASTER_SDA_IO;
    i2c_config.sda_pullup_en = GPIO_PULLUP_ENABLE;
    i2c_config.scl_io_num = I2C_MASTER_SCL_IO;
    i2c_config.scl_pullup_en = GPIO_PULLUP_ENABLE;
    i2c_config.master.clk_speed = I2C_MASTER_FREQ_HZ;
    i2c_config.clk_flags = 0;

    i2c_param_config(I2C_MASTER_NUM, &i2c_config);
    i2c_driver_install(I2C_MASTER_NUM, I2C_MODE_MASTER, 0, 0, 0);
    _showfunction = LCD_4BITMODE | LCD_1LINE | LCD_5x8DOTS;
    begin(_cols, _rows);
}
```

I2C Source:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/i2c.html#i2c-api-configure-driver

From here, printstr() was by far the most simplest function to complete as all what was needed was to iterate through the string and print out each character.



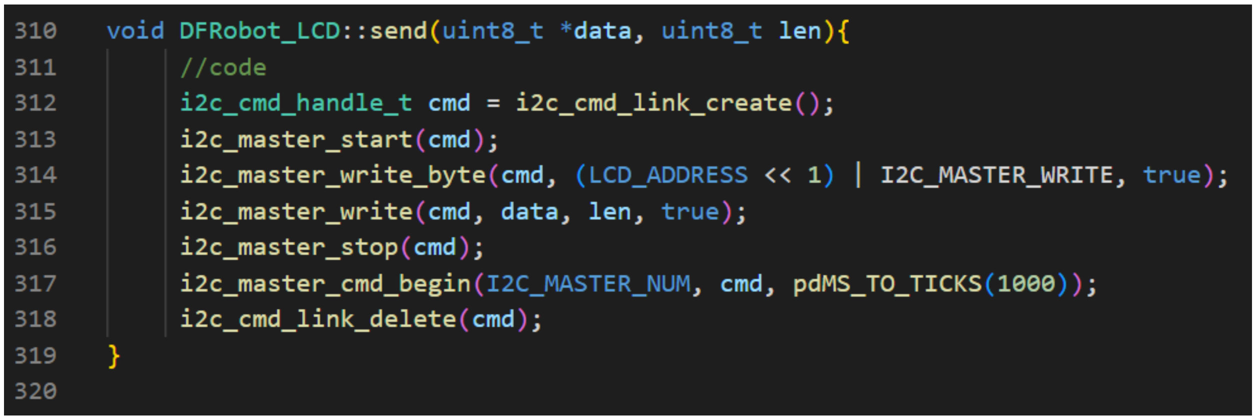
```
240  void DFRobot_LCD::printstr(const char c[]){
241      ///< This function is not identical to the function used for "real" I2C displays
242      ///< it's here so the user sketch doesn't have to be changed
243      for(int k = 0; k < strlen(c); k++){
244          write(c[k]);
245      }
246  }
```

The call on line 244, the write() function, as it uses address 0x40. The reason this is so important is due to its definition in the header file as **LCD_SETCGRAMADDR** with the same address. Looking up on the matter, address 0x40 assists with the storing of characters which will be needed in our case to store and print out onto the LCD. Essentially, it will take in each character from our forloop and store it in CG-RAM.

Address Source:

https://www.electronicsforu.com/technology-trends/learn-electronics/16x2-lcd-pinout-diagram#:~:text=CG%2DRAM%20address%20starts%20from,and%20printing%20commands%20are%20below.

The send() function was another that needed to be changed to suit the I2C environment. The purpose of this function is to be able to transmit to the MASTER and prevent further transmissions with the LCD screen. Since we dealt with sending/transmitting data in lab2, a lot of it could have been reused, as shown below. Order here matters so it took a lot of trial and error that led up to this point. However, since we are working with our MASTER we must do all the necessary changes by using the given inputs and adding in **i2c_master_cmd_begin**, but why? After reading the source below, this function call allows for the execution of the command link to which it will be need to delete as it cannot be modified once executed.



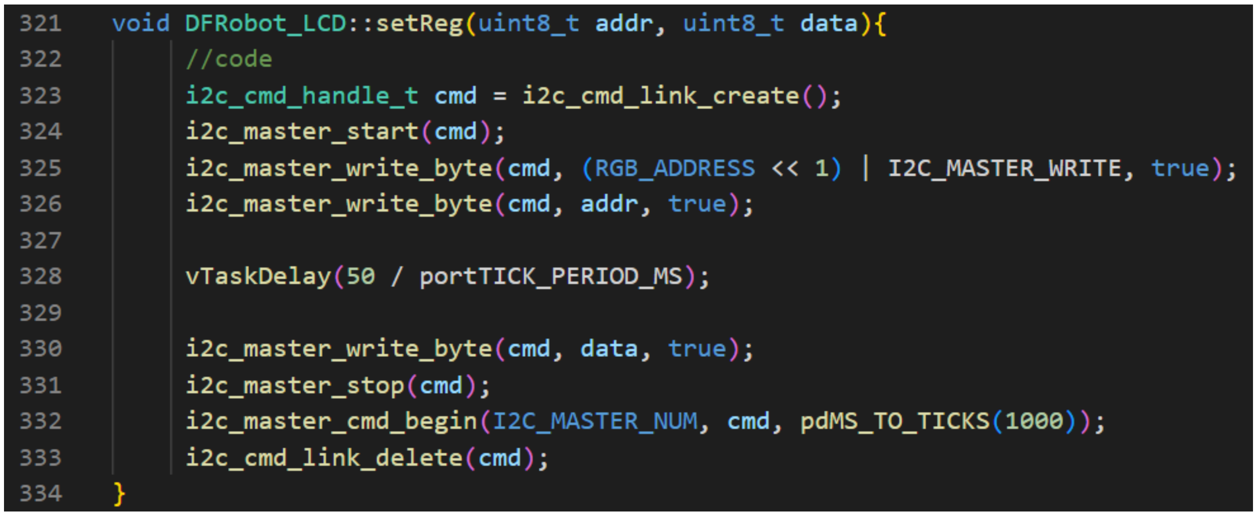
```
void DFRobot_LCD::send(uint8_t *data, uint8_t len){
    //code
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (LCD_ADDRESS << 1) | I2C_MASTER_WRITE, true);
    i2c_master_write(cmd, data, len, true);
    i2c_master_stop(cmd);
    i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, pdMS_TO_TICKS(1000));
    i2c_cmd_link_delete(cmd);
}

```

I2C Begin:

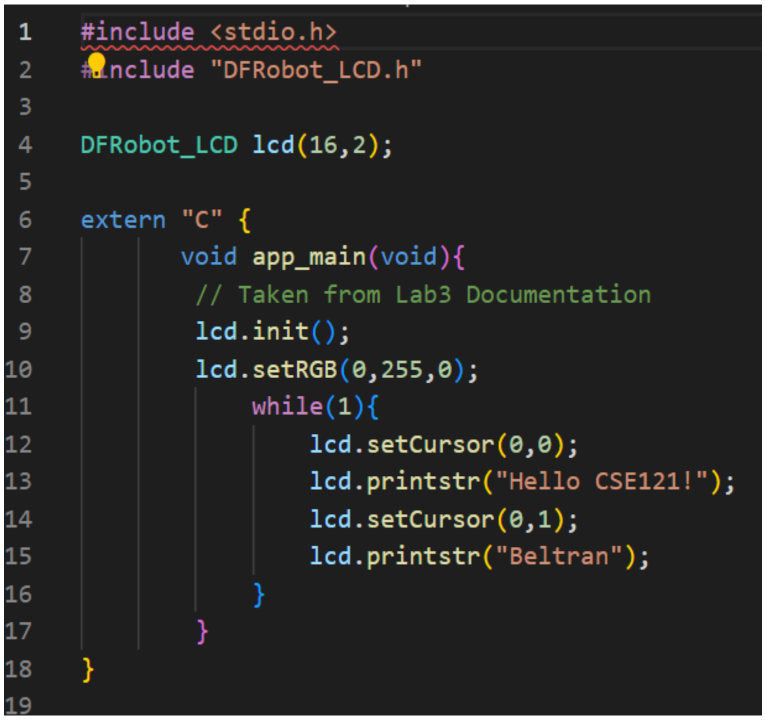
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/i2c.html

With the setreg() function, it looks over the RGB address as it assists in the coloring of our LCD as it transmits data to it. Like the send() function, there was a lot of the same code being used here to create, write/read, begin, and delete the command. Adding the delay on line 328 assists the program as it allows the process to take a break before writing again. Essentially, we follow the same format but have to consider the RGB address along with the data that is being transmitted for coloring.



```
321  void DFRobot_LCD::setReg(uint8_t addr, uint8_t data){
322      //code
323      i2c_cmd_handle_t cmd = i2c_cmd_link_create();
324      i2c_master_start(cmd);
325      i2c_master_write_byte(cmd, (RGB_ADDRESS << 1) | I2C_MASTER_WRITE, true);
326      i2c_master_write_byte(cmd, addr, true);
327
328      vTaskDelay(50 / portTICK_PERIOD_MS);
329
330      i2c_master_write_byte(cmd, data, true);
331      i2c_master_stop(cmd);
332      i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, pdMS_TO_TICKS(1000));
333      i2c_cmd_link_delete(cmd);
334  }
```

Overall, this part of the lab was the most challenging and covered a lot of reading of the documentation. The only part left to do was build lab3_2.cpp and display the contents of my name. Using inspiration from the lab documentation, I had the following below. By initializing the lcd, printing out the same RBG given, and setting the proper row/column each print statement has to be located. Using extern "C" allowed for the mesh between C/C++ files that we have.



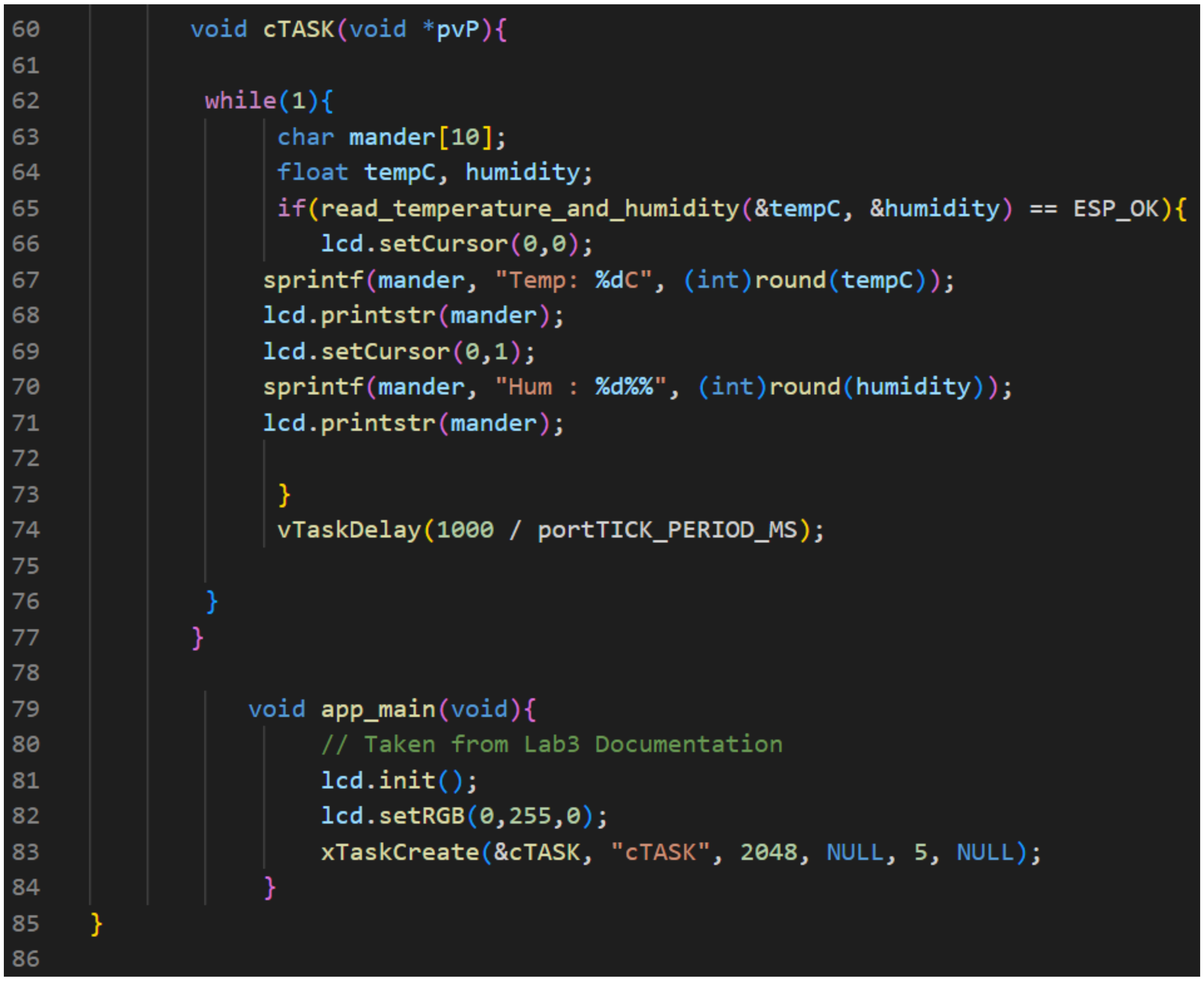
```
#include <stdio.h>
#include "DFRobot_LCD.h"

DFRobot_LCD lcd(16,2);

extern "C" {
        void app_main(void){
         // Taken from Lab3 Documentation
         lcd.init();
         lcd.setRGB(0,255,0);
             while(1){
                 lcd.setCursor(0,0);
                 lcd.printstr("Hello CSE121!");
                 lcd.setCursor(0,1);
                 lcd.printstr("Beltran");
             }
         }
}
```

Lab 3: Part 3

The last part of the lab proved to be quite simple as I knew everything I needed to know as part two covered all the necessary information. Since lab2 required us to measure the temperature and humidity (using the SHTC3 sensor) and lab3 required the same but outputted on the LCD, all the code was reused from lab2. As a note, every function was within **Extern "C"** to mesh with C++ files for the LCD. The following screenshot shows **app_main** and **cTASK** functions that I had used to implement the temperature/humidity sensor. cTASK handled the printing of the temperature/humidity in the same way it did in lab2 but using the proper syntax of DFRobot_LCD. Using a buffer, mander is passed into **sprintf()** to print the string as it contains integers. From here we call on our lcd function printstr() and setCursor() to adjust the print statements with its rows/columns. From here, in app_main I call initiate lcd, set the RGB, and call on xTaskCreate.



```
       void cTASK(void *pvP){

        while(1){
             char mander[10];
             float tempC, humidity;
             if(read_temperature_and_humidity(&tempC, &humidity) == ESP_OK){
                lcd.setCursor(0,0);
            sprintf(mander, "Temp: %dC", (int)round(tempC));
            lcd.printstr(mander);
            lcd.setCursor(0,1);
            sprintf(mander, "Hum : %d%%", (int)round(humidity));
            lcd.printstr(mander);

             }
             vTaskDelay(1000 / portTICK_PERIOD_MS);

        }
       }

           void app_main(void){
                // Taken from Lab3 Documentation
                lcd.init();
                lcd.setRGB(0,255,0);
                xTaskCreate(&cTASK, "cTASK", 2048, NULL, 5, NULL);
            }
}
```