CSE 474 Lab Report #3

Luke Jiang, 1560831
Hantao Liu, 1428377

Abstract:

In this lab, we are mainly focus on implementing modules such as analog to digital converters, Universal Asynchronous Receiver/Transmitter interface, DMA and Liquid Crystal Display interface. ADC allows us to change any input analog signal into digital signal. UART interface allows us to communicate with application with computer, for example read and write on putty. We use DMA to access memory. We can also display pictures and graphs on LCD as well as using LCD as a control.

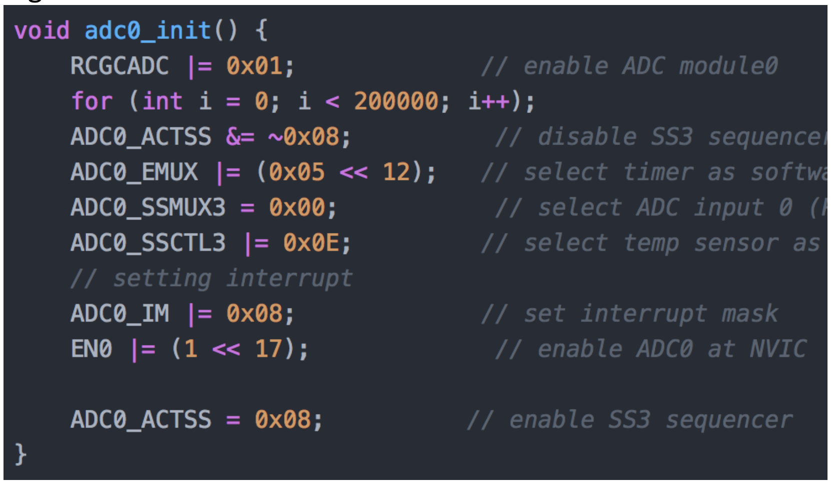
Introduction:

This lab includes four sections. For section A, we implement PLL and ADC modules, which enables us to control clock speed with two switches. ADC will then receive the code which can be converted into temperature. According to the resulting temperature, we can decided which light to turn on. Section B is pretty much like section A, but we implement one more module called UART which let us print temperature of CPU on PUTTY. Then we get to use DMA to access memory in section C. Once the transfer process is complete, we can change the color of light. In the last section, we concentrate on using LCD to implement task such as traffic control and rotating cube.

Procedure:

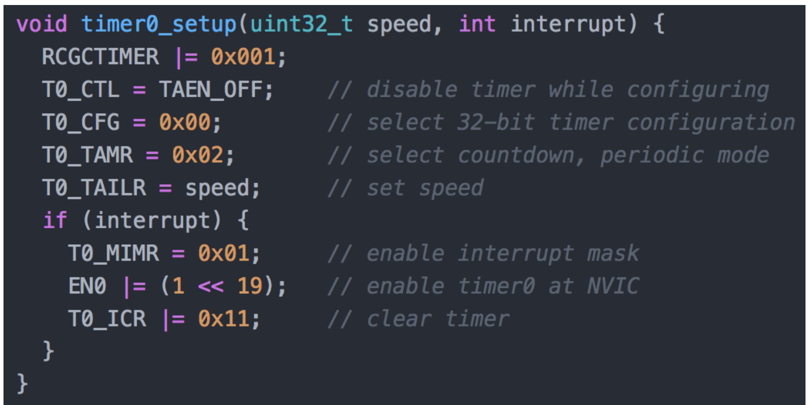
Section A:

Step 1: First of all, we have to do the initialization for all the module that we are going to use. So, the first thing is to initialize the module called ADC which is also refer to analog to digital converter. The code below demonstrates how to initialize ADC

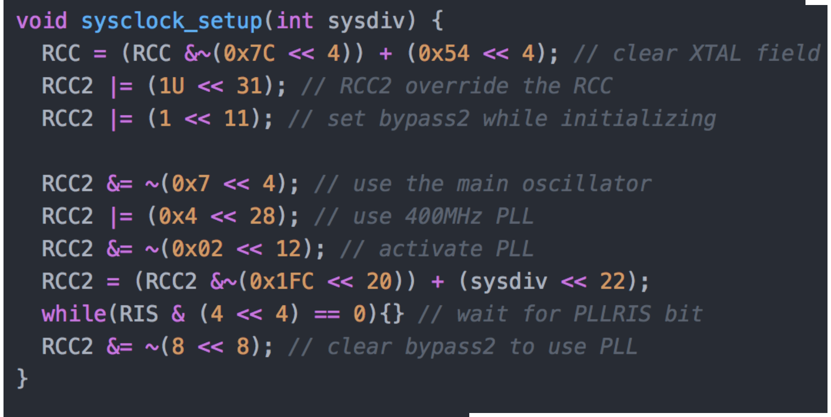


```
void adc0_init() {
    RCGCADC |= 0x01;            // enable ADC module0
    for (int i = 0; i < 200000; i++);
    ADC0_ACTSS &= ~0x08;         // disable SS3 sequence
    ADC0_EMUX |= (0x05 << 12);  // select timer as softw
    ADC0_SSMUX3 = 0x00;          // select ADC input 0 (
    ADC0_SSCTL3 |= 0x0E;        // select temp sensor as
    // setting interrupt
    ADC0_IM |= 0x08;            // set interrupt mask
    EN0 |= (1 << 17);            // enable ADC0 at NVIC

    ADC0_ACTSS = 0x08;         // enable SS3 sequencer
}
```

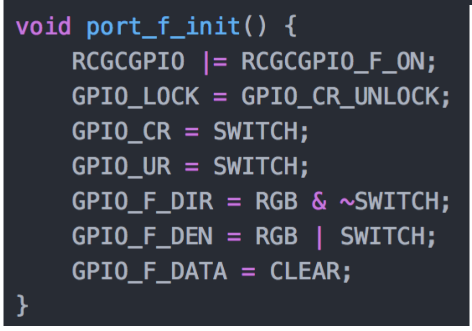
After done with initializing the ADC, we are going to initialize PLL, timer and Port F. Below are all the codes that demonstrates how to initialize those modules respectively.



```c
void timer0_setup(uint32_t speed, int interrupt) {
  RCGCTIMER |= 0x001;
  T0_CTL = TAEN_OFF;    // disable timer while configuring
  T0_CFG = 0x00;        // select 32-bit timer configuration
  T0_TAMR = 0x02;       // select countdown, periodic mode
  T0_TAILR = speed;     // set speed
  if (interrupt) {
    T0_MIMR = 0x01;     // enable interrupt mask
    EN0 |= (1 << 19);   // enable timer0 at NVIC
    T0_ICR |= 0x11;     // clear timer
  }
}
```

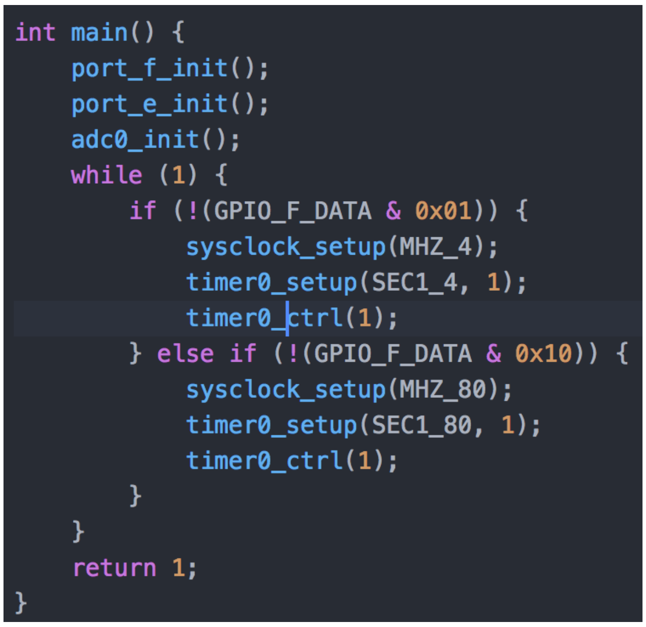


```c
void sysclock_setup(int sysdiv) {
  RCC = (RCC &~(0x7C << 4)) + (0x54 << 4); // clear XTAL field
  RCC2 |= (1U << 31); // RCC2 override the RCC
  RCC2 |= (1 << 11); // set bypass2 while initializing

  RCC2 &= ~(0x7 << 4); // use the main oscillator
  RCC2 |= (0x4 << 28); // use 400MHz PLL
  RCC2 &= ~(0x02 << 12); // activate PLL
  RCC2 = (RCC2 &~(0x1FC << 20)) + (sysdiv << 22);
  while(RIS & (4 << 4) == 0){} // wait for PLLRIS bit
  RCC2 &= ~(8 << 8); // clear bypass2 to use PLL
}
```



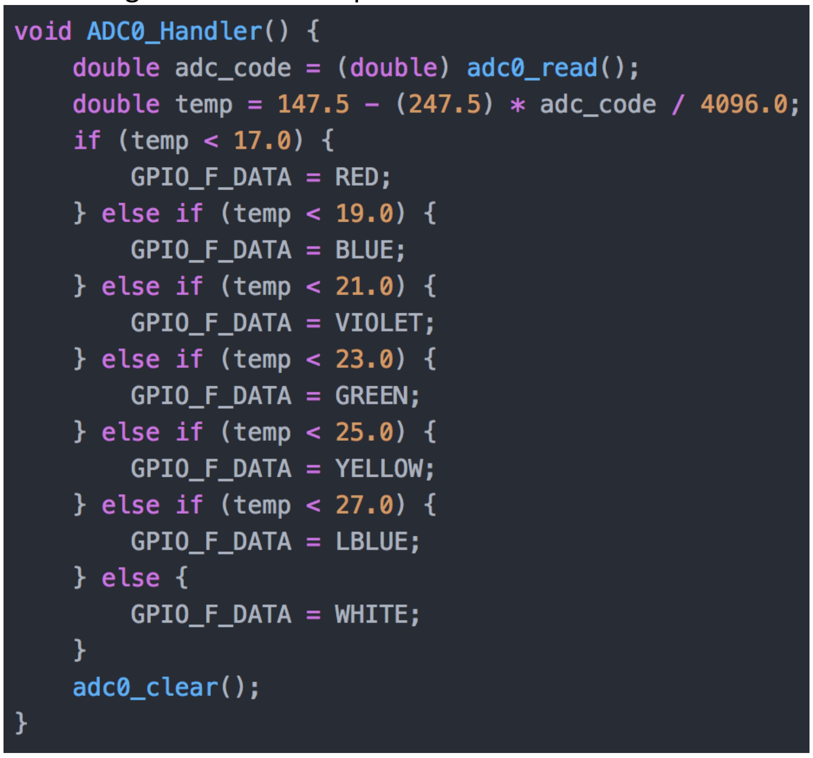
```c
void port_f_init() {
    RCGCGPIO |= RCGCGPIO_F_ON;
    GPIO_LOCK = GPIO_CR_UNLOCK;
    GPIO_CR = SWITCH;
    GPIO_UR = SWITCH;
    GPIO_F_DIR = RGB & ~SWITCH;
    GPIO_F_DEN = RGB | SWITCH;
    GPIO_F_DATA = CLEAR;
}
```

Step 2: Then we can give the input signal from Port F to activate timer and PLL. The code below demonstrate if PF0 is pressed, it is going to activate PLL with frequency of 4 MHz as well as timer.



```c
int main() {
    port_f_init();
    port_e_init();
    adc0_init();
    while (1) {
        if (!(GPIO_F_DATA & 0x01)) {
            sysclock_setup(MHZ_4);
            timer0_setup(SEC1_4, 1);
            timer0_ctrl(1);
        } else if (!(GPIO_F_DATA & 0x10)) {
            sysclock_setup(MHZ_80);
            timer0_setup(SEC1_80, 1);
            timer0_ctrl(1);
        }
    }
    return 1;
}
```

Step 3: Finally, the ACD is going to interrupt and help with calculating the temperature of running CPU. The code below demonstrate how we receive the adc code and convert it into the temperature of CPU with a formula. Then with the temperature, we are able to control the LED light on the launchpad.

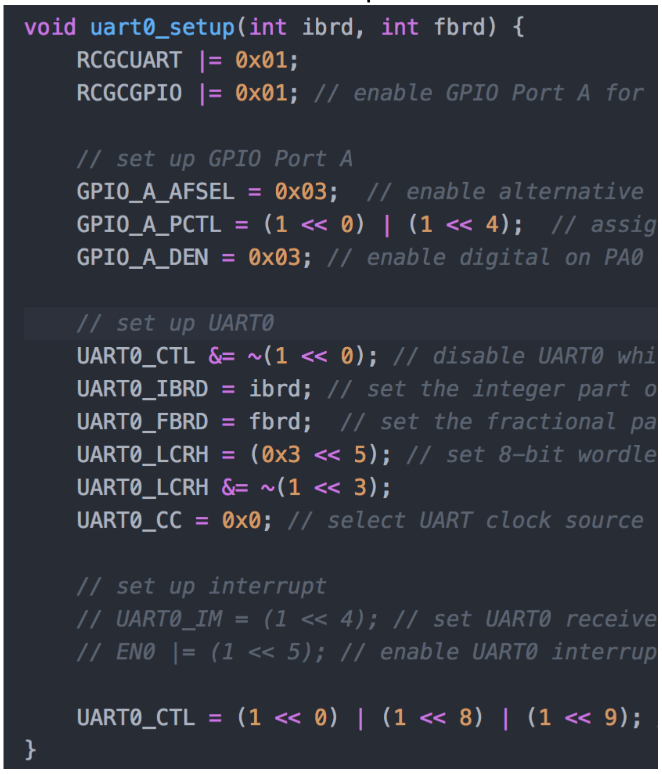


```c
void ADC0_Handler() {
    double adc_code = (double) adc0_read();
    double temp = 147.5 - (247.5) * adc_code / 4096.0;
    if (temp < 17.0) {
        GPIO_F_DATA = RED;
    } else if (temp < 19.0) {
        GPIO_F_DATA = BLUE;
    } else if (temp < 21.0) {
        GPIO_F_DATA = VIOLET;
    } else if (temp < 23.0) {
        GPIO_F_DATA = GREEN;
    } else if (temp < 25.0) {
        GPIO_F_DATA = YELLOW;
    } else if (temp < 27.0) {
        GPIO_F_DATA = LBLUE;
    } else {
        GPIO_F_DATA = WHITE;
    }
    adc0_clear();
}
```

Section B:

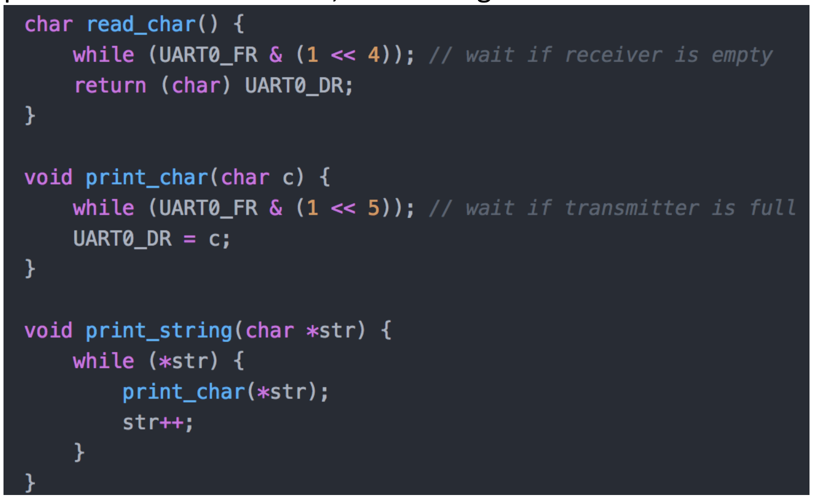
Step 1: The first step is pretty much like that of in section A. However, the biggest different is that we have to implement a module called UART in order to be able to print on PUTTY.

This module will allow us to read as well as print characters on PUTTY. The code below demonstrates how we set up UART module.



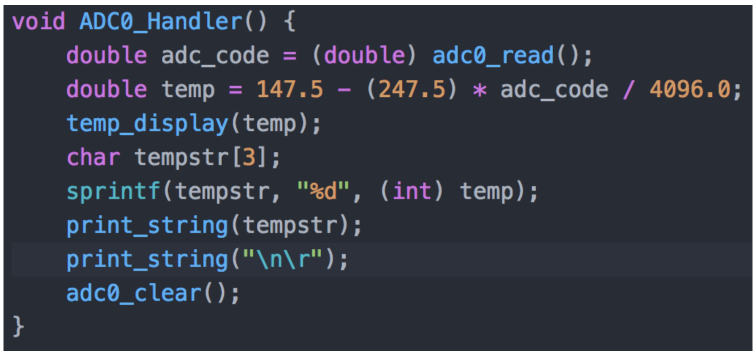
```
void uart0_setup(int ibrd, int fbrd) {
    RCGCUART |= 0x01;
    RCGCGPIO |= 0x01; // enable GPIO Port A for

    // set up GPIO Port A
    GPIO_A_AFSEL = 0x03;  // enable alternative
    GPIO_A_PCTL = (1 << 0) | (1 << 4);  // assig
    GPIO_A_DEN = 0x03; // enable digital on PA0

    // set up UART0
    UART0_CTL &= ~(1 << 0); // disable UART0 whi
    UART0_IBRD = ibrd; // set the integer part o
    UART0_FBRD = fbrd;  // set the fractional pa
    UART0_LCRH = (0x3 << 5); // set 8-bit wordle
    UART0_LCRH &= ~(1 << 3);
    UART0_CC = 0x0; // select UART clock source

    // set up interrupt
    // UART0_IM = (1 << 4); // set UART0 receive
    // EN0 |= (1 << 5); // enable UART0 interrup

    UART0_CTL = (1 << 0) | (1 << 8) | (1 << 9);
}
```

Step 2: After setting up all those modules, we can start implementing what we want. However, everything is very similar to those in section A, except we added the methods that we derived from UART. The code below demonstrates how we can use UART register to print and read characters, even strings.



```
char read_char() {
    while (UART0_FR & (1 << 4)); // wait if receiver is empty
    return (char) UART0_DR;
}

void print_char(char c) {
    while (UART0_FR & (1 << 5)); // wait if transmitter is full
    UART0_DR = c;
}

void print_string(char *str) {
    while (*str) {
        print_char(*str);
        str++;
    }
}
```

Step 3: Finally, there is one more function that we need to actually fulfill the requirement of the task. The code below, like in section A, gets the adc code, put it into a formula to calculate temperature. After that, it is going to convert the temperature into characters and print them on the screen.



```c
void ADC0_Handler() {
    double adc_code = (double) adc0_read();
    double temp = 147.5 - (247.5) * adc_code / 4096.0;
    temp_display(temp);
    char tempstr[3];
    sprintf(tempstr, "%d", (int) temp);
    print_string(tempstr);
    print_string("\n\r");
    adc0_clear();
}
```

Section C:

We modified the provided code in DMATestMain.c:

1) We first updated the name of enabling interrupt from `enableInterrupt` to `__enable_interrupt` to match the definition in our library.
2) We then disable the timer interrupt by calling `__disable_interrupt` in `main`.
3) We updated the `COLORWHEEL` array to include only red, yellow, and green.

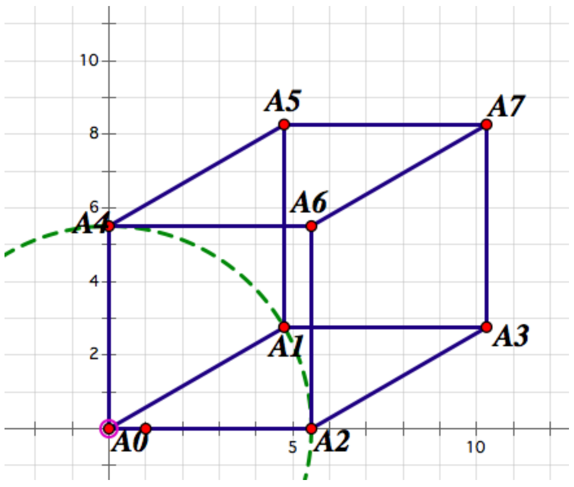
Section D:

Step 1: Fill the TODO functions and #defines in file SSD2119.c.
We filled the dimensions of the LCD in pixels, `LCD_GPIOInit`, `LCD_WriteCommand` and `Touch_Init` according to the pseudocode provided in SSD2119.c as comment as well as the datasheet for the LCD. The filled version of SSD2119.c can be found in the reference.

Step 2: Display the temperature from section A on the LCD.
The only modification we made based on Section A is changing the print_string function in uart0.h with LCD_PrintInteger in SSD2119.c

Step 3: Draw a cube in the center of the screen, and use the touchscreen to start and stop the rotation of the cube. Print the coordinates of the touch point on the LCD.
The calculations and set-ups we made before implementing the functionality in C are the following:

1) Calculate the coordinates of A_0 to A_7 according to the cube in the following picture. The length of each edge is 0.6 in. To simplify our calculation, we set the angle of $\measuredangle A_1A_0A_2$ as 30 degree ($\pi/6$ radians).



Let the coordinate of A_0 be (a, b). Then, the coordinates of A_1 to A_7 are:

A_1	$(a + 0.3\sqrt{3},\ b + 0.3)$
A_2	$(a + 0.6, b)$
A_3	$(a + 0.3\sqrt{3} + 0.6,\ b + 0.3)$
A_4	$(a,\ b + 0.6)$
A_5	$(a + 0.3\sqrt{3},\ b + 0.9)$
A_6	$(a + 0.6, b + 0.6)$
A_7	$(a + 0.3\sqrt{3} + 0.6,\ b + 0.9)$

2) The 2-D rotation formula we used to transform the cube is the following:

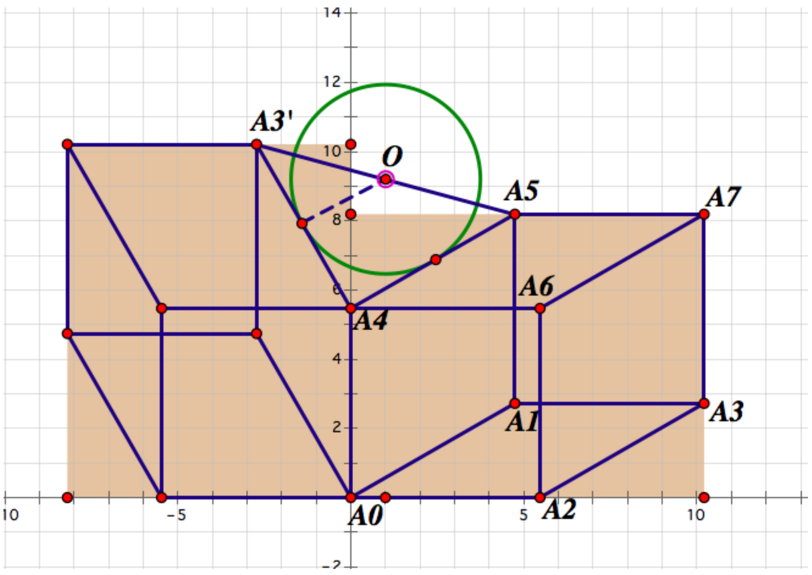
$$\begin{cases} x' = x\cos(\theta) - y\sin(\theta) \\ y' = x\sin(\theta) - y\cos(\theta) \end{cases}$$

and θ is the rotation angle. We chose θ = π / 2 as our rotation angle. Therefore, the previous formula becomes：

$$\begin{cases} x' = -y \\ y' = x \end{cases}$$

This is the formula we use to update the coordinates of eight vertices after each rotation.

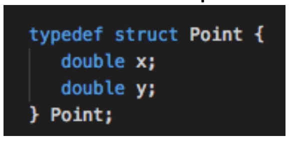
3) In order to shade the cube with color, for each cube, we first draw a rectangle from A_0 to A_7 to shadow the cube region and two extra triangular regions. Then, we draw a black circle at point O to remove as much extra pixels as possible:



The distance from point O to line A_4A_3' and line A_4A_5 is exactly 0.3 in, and this radius is same for all four circles between four cubes. The coordinate of point O can be calculated as the midpoint of A_3' and A_5. Once the first point O is known, the rest three points can be calculated from the formula in 2).

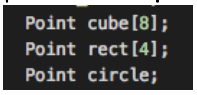
The implementation in C contains the following steps:

1) We first dealt with the conversion between pixels and inches. According to step 1, we know that the height of the LCD is 240 pixels and the width is 320 pixels. From the datasheet, we know the diagonal of the LCD is 3.5 inch. Based on this information, using Pythagorean theorem, we found out that 1 inch is equivalent to 68.57 pixels. Therefore, the values stored in our data structures would be in inches and we would convert inches to pixels when calling library functions. This would increase the accuracy of our transformation.
2) In order to facilitate calculations with 2-D points, we define a `struct` that contains two doubles representing the x coordinate and y coordinate of a point:



```
typedef struct Point {
    double x;
    double y;
} Point;
```

The data structure we used to store a cube is an array of eight Points, where each Point representing a vertex according to the second picture in procedure 3.
To reduce calculation for each rotation, we pre-calculated the top-left corner of each shading rectangle and cached the values in an array of four `Points`.
The center of the black circles is stored at a `Point`.
All these data structures are declared at main and functions may use them via pointers as parameters:

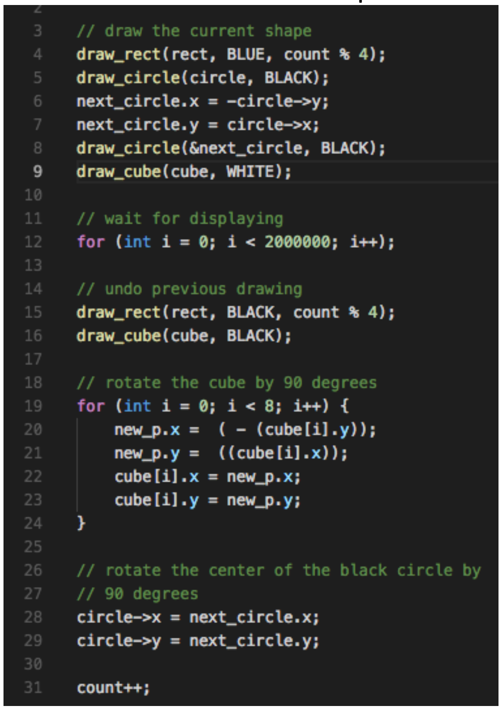


```
Point cube[8];
Point rect[4];
Point circle;
```

3) We then implemented an `init_cube` function, which initializes the initial locations of the vertices of the cube, the center of the black circle, as well as the top-left corner of four shading rectangles. The code can be found at the reference.

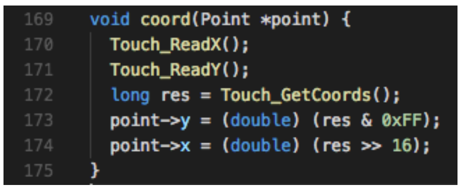
The values came from the calculation part of this step.

4) We then implemented `draw_cube`, `draw_circle` and `draw_rect` that convert inches into pixels with type unsigned short and call library functions `LCD_DrawLine`, `LCD_DrawFilledCircle` and `LCD_DrawFilledRect` to reflect the shapes on the LCD screen. The implementation of these functions can be found at the reference.
5) We then implemented the rotation logic. First, we define an `int count` that keeps track of how many rotations we have performed. Then, for each rotation, we need to display the original picture; wait for a while to let the shape stay; and then draw the black version of the picture to undo previous drawing and update the coordinates for the next rotation. Finally, we update count to keep track of the number of rotation performed:



```c
// draw the current shape
draw_rect(rect, BLUE, count % 4);
draw_circle(circle, BLACK);
next_circle.x = -circle->y;
next_circle.y = circle->x;
draw_circle(&next_circle, BLACK);
draw_cube(cube, WHITE);

// wait for displaying
for (int i = 0; i < 2000000; i++);

// undo previous drawing
draw_rect(rect, BLACK, count % 4);
draw_cube(cube, BLACK);

// rotate the cube by 90 degrees
for (int i = 0; i < 8; i++) {
    new_p.x =  ( - (cube[i].y));
    new_p.y =  ((cube[i].x));
    cube[i].x = new_p.x;
    cube[i].y = new_p.y;
}

// rotate the center of the black circle by
// 90 degrees
circle->x = next_circle.x;
circle->y = next_circle.y;

count++;
```

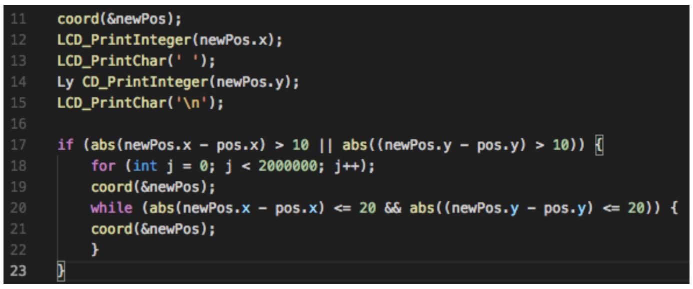
6) We then implemented the logic for reading touch coordinates. We used library function `Touch_readX`, `Touch_readY` and `Touch_Coords` to get the coordinate of each touch. For better readability, we factored this procedure in function `coord`:



```c
void coord(Point *point) {
  Touch_ReadX();
  Touch_ReadY();
  long res = Touch_GetCoords();
  point->y = (double) (res & 0xFF);
  point->x = (double) (res >> 16);
}
```

The function takes a pointer to a `Point` as parameter and write the coordinates of current touch into the fields of that `Point`.

7) We then implemented the logic for start/stop on touch. We observed that when the LCD is not touched, the x and y coordinates returned from `coord` fluctuates around two values. Therefore, we define a valid touch as an event that either x or y coordinate of current read from `coord` differs from their default values by 10. When a valid touch happens, we first delay for about 1 second to avoid multiple reads caused by one touch. Then, we wait until another valid touch happens, after which the rotating procedure is resumed. In order to print the coordinates of the touch point, We used library function `LCD_PrintString` and `LCD_PrintChar`:

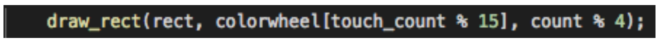


```
coord(&newPos);
LCD_PrintInteger(newPos.x);
LCD_PrintChar(' ');
Ly CD_PrintInteger(newPos.y);
LCD_PrintChar('\n');

if (abs(newPos.x - pos.x) > 10 || abs((newPos.y - pos.y) > 10)) {
    for (int j = 0; j < 2000000; j++);
    coord(&newPos);
    while (abs(newPos.x - pos.x) <= 20 && abs((newPos.y - pos.y) <= 20)) {
    coord(&newPos);
    }
}
```

Step 4: Change the color of the cube when the LCD is pressed and print the RGB code on the screen:

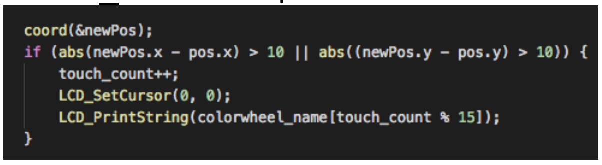
The modifications we made based on the previous step are:

1) We added before the `main` function a modified version of the `Color4` array in SSD2119.c and an array of strings of RGB codes corresponding to each color:
2) Instead of passing `BLUE` only as parameter to `draw_rect`, we pass a color in the `colorwheel` array:



```
draw_rect(rect, colorwheel[touch_count % 15], count % 4);
```

3) When a valid touch happens, instead of waiting for another touch, we update the `touch_count` and print the new color to the screen:

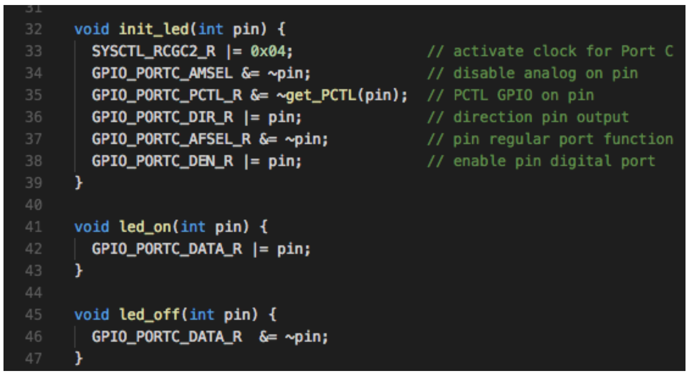


```
coord(&newPos);
if (abs(newPos.x - pos.x) > 10 || abs((newPos.y - pos.y) > 10)) {
    touch_count++;
    LCD_SetCursor(0, 0);
    LCD_PrintString(colorwheel_name[touch_count % 15]);
}
```

Step 5: Update the traffic light control FSM in lab 2 by replacing two buttons with two virtual buttons on LCD

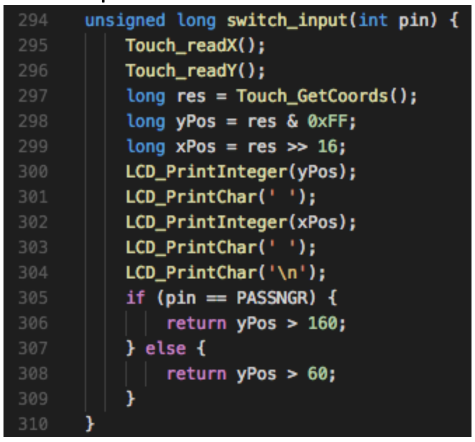
The modifications we made based on Lab 2 Section B are:

1) Instead of using GPIO Port A for interfacing with LEDs, we used Port C to avoid conflicts with LCD functionalities using Port A. Therefore, we updated logic for LED initialization and LED on/off control:



```c
void init_led(int pin) {
  SYSCTL_RCGC2_R |= 0x04;             // activate clock for Port C
  GPIO_PORTC_AMSEL &= ~pin;           // disable analog on pin
  GPIO_PORTC_PCTL_R &= ~get_PCTL(pin); // PCTL GPIO on pin
  GPIO_PORTC_DIR_R |= pin;            // direction pin output
  GPIO_PORTC_AFSEL_R &= ~pin;         // pin regular port function
  GPIO_PORTC_DEN_R |= pin;            // enable pin digital port
}

void led_on(int pin) {
  GPIO_PORTC_DATA_R |= pin;
}

void led_off(int pin) {
  GPIO_PORTC_DATA_R  &= ~pin;
}
```

2) The `switch_input` function is updated to use library function `Touch_readX`, `Touch_readY` and `Touch_Coords` to get the current touch value and, based on the switch `main` is asking for, returns whether the coordinate satisfies the valid press constraint:



```c
unsigned long switch_input(int pin) {
    Touch_readX();
    Touch_readY();
    long res = Touch_GetCoords();
    long yPos = res & 0xFF;
    long xPos = res >> 16;
    LCD_PrintInteger(yPos);
    LCD_PrintChar(' ');
    LCD_PrintInteger(xPos);
    LCD_PrintChar(' ');
    LCD_PrintChar('\n');
    if (pin == PASSNGR) {
        return yPos > 160;
    } else {
        return yPos > 60;
    }
}
```

Results

Section A & B:
We successfully implemented the functionalities described in the spec, but we observed that the temperature readings we obtained from ADC changes much faster than we anticipated. When we press the left button, the LCD skips several colors in the middle and goes directly to light blue. The primary challenge we faced while implementing these two sections is figuring out the set-ups for driver files of UART and ADC modules.

Section C:
We implemented the functionality described in the spec. We didn't face much challenge in this section.

Section D:
We implemented all the functionalities described in the spec. The major obstacle we faced is making the cube rotate and fill the color. We tried multiple algorithms and the one we finally used is the best we could do. The only disadvantage of our algorithm is that there are extra pixels unable to be removed by the black circles.

Conclusion

I learned the following skills in this lab:

1) Using ADC module, internal temperature sensor, and UART module on the board.
2) Using DMA module on the board.
3) Setting up driver files for the LCD.
4) Implementing graphics algorithm for LCD.

The concepts and techniques introduced in this lab is crucial for future labs and projects. DMA is an efficient way of acquiring data without interrupting the microprocessor, and doing computer graphics on LCD prepares us for our final project. The UART communication protocol is also useful for transferring data from the board to the computer so that we can save and query the data later.

Reference

1. TM4C123 datasheet
2. EB-LM4F120-L35 datasheet
3. http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C10_FiniteStateMachines.htm
4. http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C14_ADCdataAcquisition.htm
5. https://en.wikipedia.org/wiki/Rotation_matrix
6. https://sites.google.com/site/luiselectronicprojects/tutorials/tiva-tutorials/tiva-dma/understanding-the-tiva- dma
7. Source code in .zip