Andrew Benner
EECE 458 - Final Project
Prof. Summerville
12/17/21

Executive Summary

The goal of this project is to emit a color from the rgb led by combining three different colors that the user will choose. Color choices will be made by moving the FRDM board along the X,Y, and Z planes. This is accomplished by making use of the **accelerometer (i2c0_irq.c)**, the button (sw1.c), the rgb led (rgbled_pwm.c), and the timer drivers (Systick.c, cop_wdt.c).

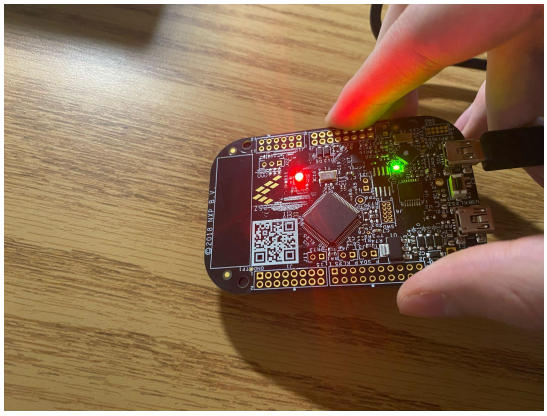
The approach to this project is a **new finite state machine** that breaks the color choices into separate states. The first state is called “ST_IDLE”, where the board waits for a button press to begin the program. The next state is called “ST_Z”. ST_Z allows the user to choose between two colors by moving the FRDM board along the Z plane and pressing the button to choose one of the two possible colors. The Z plane is broken into two ranges of 0-180 degrees and 180-360 degrees. These two ranges emit two different colors which the user can choose from. Upon pressing the button in ST_Z, the color choice gets saved and the finite state machine moves to “ST_X”. ST_X behaves exactly like ST_Z, with the sole exception of different color choices being demonstrated by movement along the X plane. A button press will save the color choice and take the finite state machine to “ST_Y”, which allows the user to choose from one of two colors depending on the range of degrees in the Y plane. Finally, a button press from ST_Y will take the finite state machine to “ST_FINAL”, where a function named “color_calculator” is run and will calculate the duty cycle of the red,green and blue led. The final color is emitted and another button press takes it back to ST_IDLE.

The approach to this problem worked very well. Moving the FRDM board along the X,Y, and Z planes is a fun and efficient way to choose different colors to create the final color. The finite state machine is relatively simple, but does everything it needs to do. There were no limitations in this approach, but some final color combinations were a little too close to white than desired.

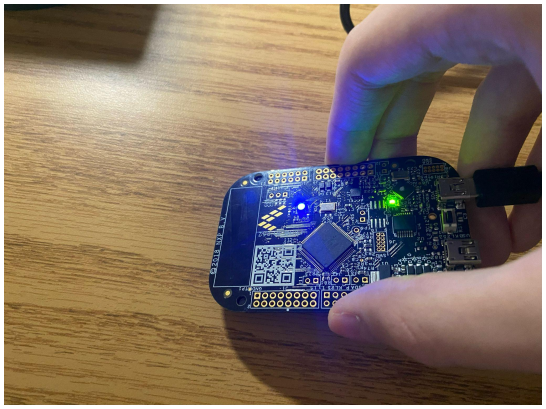
Listed below are pictures that demonstrate every state of the finite state machine.

ST_IDLE: The board is waiting for a button press to begin the program

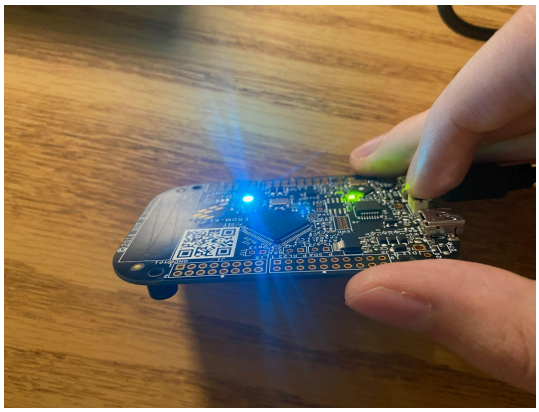
ST_Z: The user chooses red along the Z plane



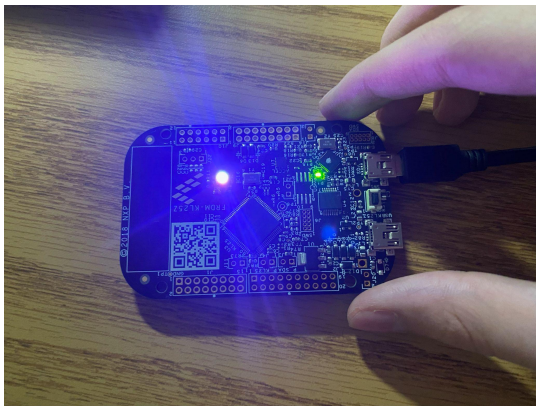
ST_X: The user chooses blue along the X plane



ST_Y: The user chooses cyan along the Y plane



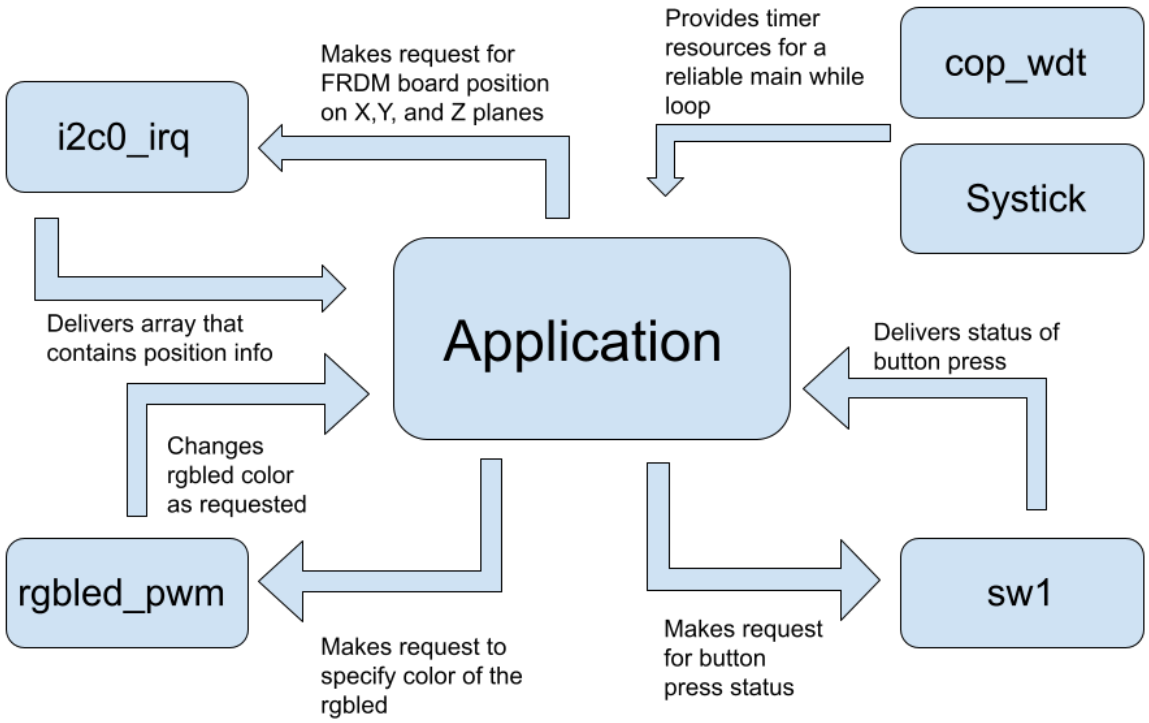
ST_FINAL: The final color is displayed (a shade of purple)



A video that demonstrates the project in more detail has been emailed to Prof. Summerville. That video contains a brief explanation of the project and multiple demos of different color combinations.

Detailed Design

System Diagram:



The i2c0_irq driver relays information regarding the position of the FRDM board in various planes to the main application. The position information is held in an array and the application can ask for it specifically (X plane position is held in index 1, Y plane position in index 2, and Z plane position in index 3). The sw1 driver gives the application the ability to detect button clicks from the user. This is vital to the program because the user needs button clicks in order to save color choices and in order to continue to progress through the finite state machine. The rgbled_pwm driver grants the application access to the led. This is also essential because the user cannot make color choices or see the final color created if the application could not access and program the led. Finally the Systick and cop_wdt drivers work together to keep the main loop running smoothly and consistently by “feeding the watchdog”.

Appendices

Final_Project.c:

```
// Author: Andrew Benner
/* This program uses the accelerometer to allow the user to choose different colors on the board
per plane. All 3 colors are added at the end to create a new color. */

#include "rgbled_pwm.h"
#include "i2c0_irq.h"
#include "copwdt.h"
#include "sw1.h"
#include "Systick.h"
#include <stdint.h>
#include <stdbool.h>

#define DEBOUNCE_TIME_MS 5
#define LONG_PRESS_TIME_MS 1500
#define ACCEL_I2C_ADDR 0x1d
#define I2C_BUFSIZE 7
volatile uint8_t i2c_data[I2C_BUFSIZE];

typedef enum{NONE,SHORTP,LONGP}press_t;
press_t switch_check_fsm();

void accelerometer_fsm();
void color_calculator(_Bool Z_Color, _Bool X_Color, _Bool Y_Color);

uint16_t R_Duty = 0x0000;
uint16_t G_Duty = 0x0000;
uint16_t B_Duty = 0x0000;

void main()
{
	asm("CPSID i");
	configure_rgbled();
	configure_sw1();
	configure_i2c0();
	configure_copwdt();
	configure_systick();
	asm("CPSIE i");

	_Bool Z_Color;
	_Bool X_Color;
	_Bool Y_Color;

	while(1)
	{
```

```
asm("WFI"); //Wait for SysTick IRQ
if(!systick_has_fired())
{
	continue;
}

accelerometer_fsm();
press_t pressed = switch_check_fsm();

static enum {ST_IDLE, ST_Z, ST_X, ST_Y, ST_FINAL} state = ST_IDLE;
switch(state)
{
case ST_IDLE:
	set_rgbled_color_to(0,0,0);
	if(pressed == SHORTP)
	{
		state = ST_Z;
	}
	break;
case ST_Z:
	if(((i2c_data[3])<<8) <    0x8000)
	{
		set_rgbled_color_to((i2c_data[3])<<8,0,0);
		if(pressed == SHORTP)
		{
			Z_Color = true; //red
			state = ST_X;
		}
	}
	else
	{
		set_rgbled_color_to(0,(i2c_data[3])<<8,0);
		if(pressed == SHORTP)
		{
			Z_Color = false; //green
			state = ST_X;
		}
	}
	break;
case ST_X:
	if(((i2c_data[1])<<8) <    0x8000)
	{
		set_rgbled_color_to(0,0,(i2c_data[1])<<8);
		if(pressed == SHORTP)
		{
			X_Color = true; //blue
			state = ST_Y;
		}
	}
```

```
                else
                {
                        set_rgbled_color_to(((i2c_data[1])<<8)/2,((i2c_data[1])<<8)/2,0);
                        if(pressed == SHORTP)
                        {
                                X_Color = false; //yellow
                                state = ST_Y;
                        }
                }
                break;
        case ST_Y:
                if(((i2c_data[2])<<8) <    0x8000)
                {
                        set_rgbled_color_to(((i2c_data[2])<<8)/2,0,((i2c_data[2])<<8)/2);
                        if(pressed == SHORTP)
                        {
                                Y_Color = true; //purple
                                state = ST_FINAL;
                        }
                }
                else
                {
                        set_rgbled_color_to(0,((i2c_data[2])<<8)/2,((i2c_data[2])<<8)/2);
                        if(pressed == SHORTP)
                        {
                                Y_Color = false; //cyan
                                state = ST_FINAL;
                        }
                }
                break;
        case ST_FINAL:
                color_calculator(Z_Color, X_Color, Y_Color);
                set_rgbled_color_to(R_Duty, G_Duty, B_Duty);
                if(pressed == SHORTP)
                {
                        state = ST_IDLE;
                }
                break;
        }

        feed_the_watchdog();

    }
    return;
}

void accelerometer_fsm()
{
```

```
	static enum {ST_START, ST_WAIT_WHOAMI, ST_WHOAMI_SUCCESS, ST_WHOAMI_FAIL,
ST_CONFIG_WAIT, ST_READ_ACCEL_DATA} state=ST_START;
	switch(state)
	{
	case ST_START:
		i2c0_read_bytes(ACCEL_I2C_ADDR <<1, 0x0d, 1, i2c_data);
		state=ST_WAIT_WHOAMI;
		break;
	case ST_WAIT_WHOAMI:
		if(i2c0_last_transaction_complete())
		{
			if(i2c0_last_transaction_had_error())
			{
				state=ST_START;
			}
			if(i2c_data[0] == 0x1A)
			{
				state=ST_WHOAMI_SUCCESS;
			}
			else
			{
				state=ST_WHOAMI_FAIL;
			}
		}
		break;
	case ST_WHOAMI_SUCCESS:
		i2c0_write_byte(ACCEL_I2C_ADDR <<1, 0x2a, 0x1f);
		state=ST_CONFIG_WAIT;
		break;
	case ST_WHOAMI_FAIL:
		break;
	case ST_CONFIG_WAIT:
		if(i2c0_last_transaction_complete())
		{
			if(i2c0_last_transaction_had_error())
			{
				state=ST_WHOAMI_SUCCESS;
			}
			else
			{
				state=ST_READ_ACCEL_DATA;
				i2c0_read_bytes(ACCEL_I2C_ADDR <<1, 0x0, 4, i2c_data);
			}
		}
		break;
	case ST_READ_ACCEL_DATA:
		if(i2c0_last_transaction_complete())
		{
			i2c0_read_bytes(ACCEL_I2C_ADDR <<1, 0x0, 4, i2c_data);
```

```
                }
                break;
        }

}

void color_calculator(_Bool Z_Color, _Bool X_Color, _Bool Y_Color)
{
        if((Z_Color == false)&&(X_Color == false)&&(Y_Color == false))
        {
                R_Duty = 0x2aaa;
                G_Duty = 0xaaaa;
                B_Duty = 0x2aaa;
        }
        else if((Z_Color == true)&&(X_Color == true)&&(Y_Color == true))
        {
                R_Duty = 0x8000;
                G_Duty = 0;
                B_Duty = 0x8000;
        }
        else if((Z_Color == true)&&(X_Color == false)&&(Y_Color == false))
        {
                R_Duty = 0x8000;
                G_Duty = 0x5555;
                B_Duty = 0x2aaa;
        }
        else if((Z_Color == false)&&(X_Color == true)&&(Y_Color == false))
        {
                R_Duty = 0;
                G_Duty = 0x8000;
                B_Duty = 0x8000;
        }
        else if((Z_Color == false)&&(X_Color == false)&&(Y_Color == true))
        {
                R_Duty = 0x5555;
                G_Duty = 0x8000;
                B_Duty = 0x2aaa;
        }
        else if((Z_Color == true)&&(X_Color == true)&&(Y_Color == false))
        {
                R_Duty = 0x5555;
                G_Duty = 0x2aaa;
                B_Duty = 0x8000;
        }
        else if((Z_Color == true)&&(X_Color == false)&&(Y_Color == true))
        {
                R_Duty = 0x8000;
                G_Duty = 0x5555;
                B_Duty = 0x2aaa;
```

```
        }
        else if((Z_Color == false)&&(X_Color == true)&&(Y_Color == true))
        {
                R_Duty = 0x2aaa;
                G_Duty = 0x5555;
                B_Duty = 0x8000;
        }
}

press_t switch_check_fsm()
{
        static uint16_t switch_time=0;
        static enum {ST_NO_PRESS, ST_DEBOUNCE_PRESS, ST_PRESS,
                ST_LONG_PRESS, ST_DEBOUNCE_RELEASE} sw_state=ST_NO_PRESS;

        switch_time++;

        switch(sw_state){
         default:
         case ST_NO_PRESS:
                if( sw1_is_pressed() )
                {
                        switch_time = 0;
                        sw_state = ST_DEBOUNCE_PRESS;
                }
                break;
         case ST_DEBOUNCE_PRESS:
                if( sw1_is_not_pressed() )
                        sw_state = ST_NO_PRESS;
                else if( switch_time >= DEBOUNCE_TIME_MS )
                        sw_state = ST_PRESS;
                break;
         case ST_PRESS:
                if( sw1_is_not_pressed() )
                {
                        switch_time=0;
                        sw_state = ST_DEBOUNCE_RELEASE;
                        return SHORTP;
                }
                else if( switch_time >= LONG_PRESS_TIME_MS )
                        sw_state = ST_LONG_PRESS;
                break;
         case ST_LONG_PRESS :
                if( sw1_is_not_pressed() )
                {
                        switch_time=0;
                        sw_state = ST_DEBOUNCE_RELEASE;
                        return LONGP;
                }
```

```
                break;
        case ST_DEBOUNCE_RELEASE:
                if( switch_time >= DEBOUNCE_TIME_MS )
                        sw_state = ST_NO_PRESS;
                break;
        }
        return NONE;
}
```

i2c0_irq.c:

```
// Author: Andrew Benner (provided by Prof. Summerville)
// Implementation of functionality of driver for the accelerometer on FRDM-KL25Z

#include "i2c0_irq.h"
#include <stdbool.h>
#include <stdint.h>
#include "MKL25Z4.h"

static volatile struct{
	union{
		uint8_t volatile *pdata;
		uint8_t data;
	};
	uint8_t count;
	uint8_t reg;
	uint8_t device;
	_Bool error:1, done:1, busy:1, write:1;
} isr_data;

static void i2c0_do_start(uint8_t);
static void i2c0_do_stop();
static void i2c0_do_repeat_start(uint8_t);
static void i2c0_put_byte( uint8_t );
static uint8_t i2c0_get_byte( );
static void i2c0_nak_after_byte_received();
static _Bool i2c0_ack_was_received();
static void i2c0_ack_after_byte_received();
static _Bool i2c0_is_busy();

_Bool i2c0_write_byte( uint8_t device, uint8_t reg, uint8_t data)
{
	if( i2c0_is_busy() || isr_data.busy)
		return false;
	isr_data.data=data;
	isr_data.reg=reg;
	isr_data.device=device;
	isr_data.busy=true;
	isr_data.error=false;
	isr_data.done=false;
	isr_data.write=true;
	i2c0_do_start( device | 0x00 );
	return true;
}

_Bool i2c0_read_bytes( uint8_t device, uint8_t reg, uint8_t count, uint8_t *data)
{
	if( i2c0_is_busy() || isr_data.busy)
```

```
                return false;
        isr_data.pdata=data;
        isr_data.count=count;
        isr_data.reg=reg;
        isr_data.device=device;
        isr_data.busy=true;
        isr_data.error=false;
        isr_data.done=false;
        isr_data.write=false;
        i2c0_do_start( device | 0x00 );
        return true;
}
_Bool i2c0_last_transaction_complete()
{
        return isr_data.done;
}
_Bool i2c0_last_transaction_had_error()
{
        return isr_data.error;
}
void configure_i2c0()
{
        SIM->SCGC4 |= SIM_SCGC4_I2C0_MASK;
        SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;
        //PORTE->PCR[24] = (PORTE->PCR[24] &! PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);
        //PORTE->PCR[25] = (PORTE->PCR[25] &! PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);
        PORTE->PCR[24] = (PORTE->PCR[24] &~ PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);
        PORTE->PCR[25] = (PORTE->PCR[25] &~ PORT_PCR_MUX_MASK) | PORT_PCR_MUX(5);
        I2C0->F =  I2C_F_ICR(0x12) | I2C_F_MULT(0);
        I2C0->C1 |= I2C_C1_IICEN_MASK | I2C_C1_IICIE_MASK;

        NVIC_SetPriority( I2C0_IRQn, 3);
        NVIC_ClearPendingIRQ(I2C0_IRQn);
        NVIC_EnableIRQ(I2C0_IRQn);

}

static void i2c0_do_start(uint8_t device_addr)
{
        I2C0->C1 |= ( I2C_C1_TX_MASK | I2C_C1_MST_MASK);
        I2C0->D = device_addr;
}
static void i2c0_do_stop()
{
        I2C0->C1 &= ~I2C_C1_MST_MASK;
}
static void i2c0_do_repeat_start(uint8_t device_addr)
{
        I2C0->C1 |= (I2C_C1_TX_MASK | I2C_C1_RSTA_MASK);
```

```
	I2C0->D = device_addr;
}
static void i2c0_put_byte( uint8_t data)
{
	I2C0->D = data;
}
static uint8_t i2c0_get_byte( )
{
	I2C0->C1 &= ~I2C_C1_TX_MASK;
	return I2C0->D;
}
static void i2c0_nak_after_byte_received()
{
	I2C0->C1 |= I2C_C1_TXAK_MASK;
}
static _Bool i2c0_ack_was_received()
{
	return !(I2C0->S & I2C_S_RXAK_MASK);
}
static void i2c0_ack_after_byte_received()
{
	I2C0->C1 &= ~I2C_C1_TXAK_MASK;
}
static _Bool i2c0_is_busy()
{
	return I2C0->S & I2C_S_BUSY_MASK;
}

void I2C0_IRQHandler()
{

	static enum { REG, RESTART, READ, WRITE1, WRITE2} state=REG;
	static int8_t i;

	I2C0->S |= I2C_S_IICIF_MASK;

	switch(state)
	{
		default:
		case REG:
			if( !i2c0_ack_was_received() )
				goto IRQ_ERROR;
			i2c0_put_byte(isr_data.reg);
			if( isr_data.write )
				state=WRITE1;
			else
				state=RESTART;
			break;
		case RESTART:
```

```
                    if( !i2c0_ack_was_received() )
                            goto IRQ_ERROR;
                    i2c0_do_repeat_start(isr_data.device | 0x01);
                    state=READ;
                    i=isr_data.count +1;
                    i2c0_ack_after_byte_received();
                    break;
            case READ:
                    if( i==isr_data.count+1 && !i2c0_ack_was_received() )
                            goto IRQ_ERROR;
                    if(i--)
                    {
                            if( i == 0 )
                                    i2c0_do_stop();
                            if( i == 1 )
                                    i2c0_nak_after_byte_received();
                            if( i == isr_data.count )
                                    i2c0_get_byte();
                            else
                                    *isr_data.pdata++=i2c0_get_byte();
                    }
                    if( i == 0 )
                    {
                            state=REG;
                            isr_data.busy=false;
                            isr_data.done=true;
                            isr_data.error=false;
                    }
                    break;
            case WRITE1:
                    if( !i2c0_ack_was_received() )
                            goto IRQ_ERROR;
                    i2c0_put_byte(isr_data.data);
                    state=WRITE2;
                    break;
            case WRITE2:
                    if( !i2c0_ack_was_received() )
                            goto IRQ_ERROR;
                    i2c0_do_stop();
                    state=REG;
                    isr_data.busy=false;
                    isr_data.done=true;
                    isr_data.error=false;
                    break;
    }
    return;

IRQ_ERROR:
    isr_data.busy=false;
```

```
            isr_data.done=true;
            isr_data.error=true;
            i2c0_do_stop();
            return;
}
```

i2c0_irq.h:

```
// Author: Andrew Benner (provided by Prof. Summerville)
// Function declarations of driver for accelerometer on FRDM-KL25Z

#ifndef I2C0_IRQ_H
#define I2C0_IRQ_H
#include <stdbool.h>
#include <stdint.h>

_Bool i2c0_write_byte( uint8_t device, uint8_t reg, uint8_t data);
_Bool i2c0_read_bytes( uint8_t device, uint8_t reg, uint8_t count, uint8_t *data);
void configure_i2c0();
_Bool i2c0_last_transaction_complete();
_Bool i2c0_last_transaction_had_error();

#endif
```

rgbled_pwm.c:

```
// Author: Andrew Benner
// Implementation of functionality of driver for rgb led with pwm and timer implementation on FRDM-KL25Z

#include <stdint.h>
#include <MKL25Z4.h>
#include "rgbled_pwm.h"

static void set_red_led_duty_cycle(uint16_t duty);
static void set_green_led_duty_cycle(uint16_t duty);
static void set_blue_led_duty_cycle(uint16_t duty);

void configure_rgbled()
{
	//configure red led port
	SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK;
	PORTB->PCR[18] = PORT_PCR_MUX(3);

	//configure red/green led timer
	SIM->SCGC6 |= SIM_SCGC6_TPM2_MASK;
	TPM2->SC = TPM_SC_PS(0) |
		TPM_SC_CMOD(0) |
		TPM_SC_CPWMS(0);
	TPM2->MOD = 0XFFFF;

	//configure red led channel
	TPM2->CONTROLS[0].CnSC = TPM_CnSC_MSB(1)
		| TPM_CnSC_MSA(0)
		| TPM_CnSC_ELSB(0)
		| TPM_CnSC_ELSA(1);

	//configure green led port
	PORTB->PCR[19] = PORT_PCR_MUX(3);

	//configure green led channel
	TPM2->CONTROLS[1].CnSC = TPM_CnSC_MSB(1)
		| TPM_CnSC_MSA(0)
		| TPM_CnSC_ELSB(0)
		| TPM_CnSC_ELSA(1);

	//configure blue led port
	SIM->SCGC5 |= SIM_SCGC5_PORTD_MASK;
	PORTD->PCR[1] = PORT_PCR_MUX(4);
	PTD->PDDR |= (1<<1);

	//configure blue led timer
	SIM->SCGC6 |= SIM_SCGC6_TPM0_MASK;
	TPM0->SC = TPM_SC_PS(0) |
```

```
                TPM_SC_CMOD(0) |
                TPM_SC_CPWMS(0);
        TPM0->MOD = 0XFFFF;

        //configure blue led channel
        TPM0->CONTROLS[1].CnSC = TPM_CnSC_MSB(1)
                | TPM_CnSC_MSA(0)
                | TPM_CnSC_ELSB(0)
                | TPM_CnSC_ELSA(1);

        turn_off_rgbled();

        TPM0->SC = TPM_SC_CMOD(1);
        TPM2->SC = TPM_SC_CMOD(1);
}
void set_rgbled_color_to(uint16_t red, uint16_t green, uint16_t blue)
{
        set_red_led_duty_cycle(red);
        set_green_led_duty_cycle(green);
        set_blue_led_duty_cycle(blue);
}
void turn_off_rgbled()
{
        set_red_led_duty_cycle(0);
        set_green_led_duty_cycle(0);
        set_blue_led_duty_cycle(0);
}

static void set_red_led_duty_cycle(uint16_t duty)
{
        TPM2->CONTROLS[0].CnV = duty;
}

static void set_green_led_duty_cycle(uint16_t duty)
{
        TPM2->CONTROLS[1].CnV = duty;
}

static void set_blue_led_duty_cycle(uint16_t duty)
{
        TPM0->CONTROLS[1].CnV = duty;
}
```

rgbled_pwm.h:

```
// Author: Andrew Benner
// Function declarations of driver for rgb led with pwm and timer implementation on FRDM-KL25Z

#ifndef RGBLED_PWM_H
#define RGBLED_PWM_H
#include <stdint.h>

void configure_rgbled();
void set_rgbled_color_to(uint16_t red, uint16_t green, uint16_t blue);
void turn_off_rgbled();

#endif
```

sw1.c:

```
// Author: Andrew Benner
// Implementation of functionality of driver for sw1 on FRDM-KL25Z

#include <MKL25Z4.h>
#include "sw1.h"
#include <stdbool.h>

#define SW1_LOCATION 20

void configure_sw1()
{
	SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;
	PORTA->PCR[SW1_LOCATION] =
		PORT_PCR_PE(1) |
		PORT_PCR_PS(1) |
		PORT_PCR_PFE(0) |
		PORT_PCR_DSE(0) |
		PORT_PCR_SRE(0) |
		PORT_PCR_MUX(1) |
		PORT_PCR_IRQC(0) |
		PORT_PCR_ISF(1);
	PTA->PDDR &= ~(1<<SW1_LOCATION);
}

_Bool sw1_is_pressed()
{
	return !(PTA->PDIR & (1<<SW1_LOCATION));
}


_Bool sw1_is_not_pressed()
{
	return !sw1_is_pressed();
}
```

sw1.h:

```
// Author: Andrew Benner
// Function declarations of driver for sw1 on FRDM-KL25Z

#ifndef SW1_H
#define SW1_H
#include <stdbool.h>

_Bool sw1_is_pressed();
_Bool sw1_is_not_pressed();
void configure_sw1();

#endif
```

Systick.c:

```
// Author: Andrew Benner
// Implementation of functionality of driver for SysTick on FRDM-KL25Z

#include <MKL25Z4.h>
#include "systick.h"
#include <stdbool.h>

static _Bool systick_interrupt_has_occurred=0;

#define SYSTICK_PERIOD ( 1000 ) //Desired Period in ms
#define SYSTICK_TOP ( SYS_CLOCK / SYSTICK_PERIOD )

void configure_systick()
{
	SysTick->LOAD = SYSTICK_TOP;
	SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
		SysTick_CTRL_TICKINT_Msk |
		SysTick_CTRL_ENABLE_Msk;
}

_Bool systick_has_fired()
{
	_Bool retval=systick_interrupt_has_occurred;
	systick_interrupt_has_occurred=0;
	return retval;
}

void SysTick_Handler()
{
	systick_interrupt_has_occurred=1;
}
```

Systick.h:

```
// Author: Andrew Benner
// Function declarations of driver for SysTick on FRDM-KL25Z

#ifndef SYSTICK_H
#define SYSTICK_H
#include <stdbool.h>

void configure_systick();
_Bool systick_has_fired();

#endif
```

cop_wdt.c:

```
// Author: Andrew Benner
// Implementation of functionality of driver for watchdog timer on FRDM-KL25Z

#include <MKL25Z4.h>
#include "copwdt.h"

void configure_copwdt()
{
	SIM->COPC = SIM_COPC_COPT(3) |
		SIM_COPC_COPCLKS(1) |
		SIM_COPC_COPW(0);
}

void feed_the_watchdog()
{
	SIM->SRVCOP = 0X55;
	SIM->SRVCOP = 0XAA;
}
```

cop_wdt.h:

```
// Author: Andrew Benner
// Function declarations of driver for watchdog timer on FRDM-KL25Z

#ifndef COPWDT_H
#define COPWDT_H

void configure_copwdt();
void feed_the_watchdog();

#endif
```