# ESS: Lab 2

Controlling Brightness

Task 1:

Pulse Width Modulation

Lab2 will be based on your previous code.

Pulse Width Modulation (PWM) is a technique to alter the average voltage or power to an external device. Simply put, we generate periodic pulses, and vary the ratio of the on/off time. If the ratio is 50%, then half the time the LED is on and half the time the LED is off. For an LED, as long as the period is below about 30 msec, then the human eye cannot see any flickering and just perceives it as a partially illuminated LED, rather than a rapidly flashing LED. Examples of PWM timing diagrams are shown below:
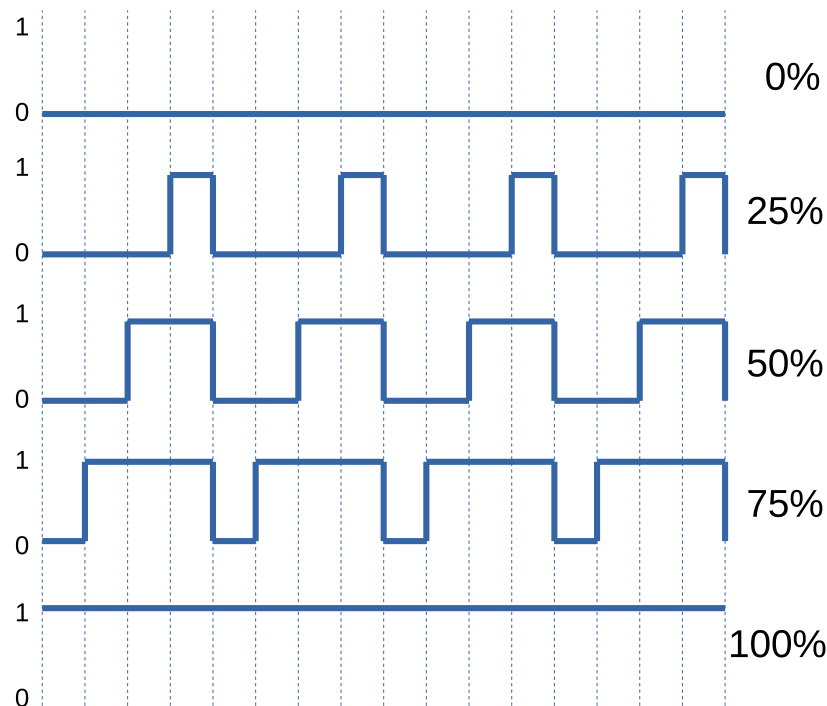
(a) Using the previously written software delay, confirm that you can control the brightness of a single LED, e.g. the green LED.

Set the total period to 10 ms, with `on_time` variable ranging between 0 and 10, and the `off_time` variable given by `off_time = 10 - on_time;`.

(b) Modify your loop to smoothly fade from completely off to completely on every few seconds to show gradual fading.

(c) Modify your loop to fade from the red LED to the green LED smoothly i.e. while the green is fading down, the red should be fading up in brightness.

Solution:

The approach is to use a complementary drive - each LED is on when the other is off.

```c
#define TOTAL_TIME 10    // Period time (10 ms)
#define CYCLE_TIME 20    // Periods per cycle (arbitrary)

// Delay for a multiple of 1 ms
void delay_msec(uint32_t delay);

void delay_msec(uint32_t delay)
{
        // Internal loop counter
        uint32_t k;
        // Dummy variable to prevent compiler optimization
        volatile uint32_t internal_fake;
        while (delay-- > 0)
        {
                for (k = 0; k < 1000; k++)
                {
                        internal_fake--;
                }
        }
}

int main(void) {
uint8_t on_time = 0;             // 0..10 ms
uint8_t loop_counter = 0;        // 0..19
while (1) {
                led_on(&led_green);
                led_off(&led_red);
                delay_msec(on_time);             // 0..10 ms
                led_off(&led_green);
                led_on(&led_red);
                delay_msec(TOTAL_TIME-on_time);  // 10..0 ms
                if ((loop_counter++) >= CYCLE_TIME)
                {  // Cycle completed, start next
                        loop_counter = 0;
                        on_time++;  // Increase on time by 1 ms
                        if (on_time > TOTAL_TIME)
                        {  // Green 100%, red 0% -> restart
                                on_time = 0;     // 0 ms
                        }
                }
        }
}
```

```
        }
```

Task 2:

Refactoring

At the moment, the PWM code we have is not very modular. What we need to build is a four channel PWM driver, where each channel is responsible for controlling an LED. Follow the similar sequence of steps as the previous lab to create pwm_driver.h and pwm_driver.c. This is an example of what the API in pwm_driver.h could look like:

```c
/**
 *        Four Channel PWM driver
 */
#ifndef PWM_DRIVER_H
#define PWM_DRIVER_H

#include <stdint.h>
// include led_driver to get the definition of LED_t
#include "led_driver.h"

// Set the maximum pwm value to 100%
#define PWM_MAX 100

// initialize pwm driver. This assumes that led_init() has been previously called for
// each channel to populate the LED_t struct properly
void pwm_driver_init(LED_t * ch0, LED_t *ch1, LED_t * ch2, LED_t * ch3);

// Set brightness value for a particular channel (channel between 0 and 3).
// Value ranges between 0 (0%) and 100 (100%).
void pwm_driver_set(uint8_t channel, uint8_t value);

/**
 * Update PWM output for each channel
 * This function is called periodically.
 * It checks the status of each pwm channel and decides whether to turn it on or off.
 *
 * @warning This assumes that pwm_driver_init() has been called for each channel before this function is called.
 *
 * No return
 */
void pwm_driver_update(void);

#endif
```

This is an example skeleton of what the implementation could look like (some code removed, comments remain):

```c
#include "pwm_driver.h"

// Encapsulate the state of the PWM driver
struct pwm_state
{
        uint32_t counter;
        uint32_t ch0_compare;
        uint32_t ch1_compare;
        uint32_t ch2_compare;
        uint32_t ch3_compare;
        LED_t * ch0;
        LED_t * ch1;
        LED_t * ch2;
        LED_t * ch3;
};
```

```
// we declare the state as static so it only has file scope and cannot be accessed externally
// this makes the assumption that we only have one pwm_driver instance in the system
static struct pwm_state state;

// initialize the pwm driver
void pwm_driver_init(LED_t * ch0, LED_t *ch1, LED_t * ch2, LED_t * ch3)
{
        // Start counter at zero
        <your code here>
        // Set all channels to off (i.e. compare value of 0)
        <your code here>
        // store the pointers for each led driver
        state.ch0 = ch0;
        state.ch1 = ch1;
        state.ch2 = ch2;
        state.ch3 = ch3;
}


void pwm_driver_set(uint8_t channel, uint8_t value)
{
        // bounds check
        <your code here>
        // update the compare register depending on which channel is selected
        <your code here>
}

void pwm_driver_update(void)
{
        // Update Ch0: if compare value is greater than counter, turn on, else turn off
        <your code here>
        // Update Ch1
        <your code here>
        // Update Ch2
        <your code here>
        // Update Ch3
        <your code here>
        // update counter value
        if (state.counter++ > PWM_MAX)
        {
                state.counter = 0;
        }

}
```

And your main.c would look something like this (note that I created a finer delay, `delay_usec()` to update the LEDs at a rate of 10kHz to avoid flickering:

```
#include "stm32f4xx.h"
/* Include helper library */
#include "ess_helper.h"
// led driver
#include "led_driver.h"
// loop delay
#include "loop_delay.h"
// pwm driver
#include "pwm_driver.h"
// Use a define for the address of the PORTD output register
#define PORTD   ((volatile uint32_t*)0x40020C14)
// main loop
int main(void) {
        // create the led ADTs
        LED_t led_green;
        LED_t led_orange;
        LED_t led_blue;
        LED_t led_red;
        /* Initialize system */
        HAL_Init();
        /* Initialize peripherals on board */
```

```
ess_helper_init();
// Set up the leds
led_init(&led_green,PORTD,12);
led_init(&led_orange,PORTD,13);
led_init(&led_red,PORTD,14);
led_init(&led_blue,PORTD,15);
// set up the pwm driver
pwm_driver_init(&led_green,&led_red,&led_orange,&led_blue);
// set brightness values
pwm_driver_set(0,100);
pwm_driver_set(1,50);
pwm_driver_set(2,25);
pwm_driver_set(3,0);
while (1) {
        delay_usec(100);
        pwm_driver_update();
}
}
```

(a) Fill in the missing code to make it all work (if you are stuck, ask for help, or read the next question on debugging).

Solution:

Here is one possible implementation:

```
#include "pwm_driver.h"

struct pwm_state
{
        uint32_t counter;
        uint32_t ch0_compare;
        uint32_t ch1_compare;
        uint32_t ch2_compare;
        uint32_t ch3_compare;
        LED_t * ch0;
        LED_t * ch1;
        LED_t * ch2;
        LED_t * ch3;
};

// we declare as static so it only has file scope and cannot be accessed externally
static struct pwm_state state;

void pwm_driver_init(LED_t * ch0, LED_t *ch1, LED_t * ch2, LED_t * ch3)
{
        // Start counter at zero
        state.counter = 0;
        // Set all channels to off (i.e. compare value of 0)
        state.ch0_compare = 0;
        state.ch1_compare = 0;
        state.ch2_compare = 0;
        state.ch3_compare = 0;
        // store the pointers for each led driver
        state.ch0 = ch0;
        state.ch1 = ch1;
        state.ch2 = ch2;
        state.ch3 = ch3;
}


void pwm_driver_set(uint8_t channel, uint8_t value)
{
        // bounds check
        if (value > PWM_MAX)
        {
                value = PWM_MAX;
```

```
                }
                // update the compare register
                switch (channel)
                {
                        case(0):
                                state.ch0_compare = value;
                                break;
                        case(1):
                                state.ch1_compare = value;
                                break;
                        case(2):
                                state.ch2_compare = value;
                                break;
                        case(3):
                                state.ch3_compare = value;
                                break;
                }
}

void pwm_driver_update(void)
{
        // Update Ch0
        if (state.ch0_compare > state.counter)
        {
                led_on(state.ch0);
        }
        else
        {
                led_off(state.ch0);
        }
        // Update Ch1
        if (state.ch1_compare > state.counter)
        {
                led_on(state.ch1);
        }
        else
        {
                led_off(state.ch1);
        }
        // Update Ch2
        if (state.ch2_compare > state.counter)
        {
                led_on(state.ch2);
        }
        else
        {
                led_off(state.ch2);
        }
        // Update Ch3
        if (state.ch3_compare > state.counter)
        {
                led_on(state.ch3);
        }
        else
        {
                led_off(state.ch3);
        }
        // update counter value
        if (state.counter++ > PWM_MAX)
        {
                state.counter = 0;
        }

}
```

(b) Modify your routine to fade between all four LEDs in a repetitive pattern to demonstrate it working.

(c) ⋆ The method of passing a pointer to the LED_t struct works, but it could be made more elegant. Suggest some further refactoring to reduce the coupling between the PWM driver and the LED_driver.

> **Solution:**
>
> A struct containing the LED struct itself and the function pointers could be passed to the PWM driver that encapsulates all the state associated with one LED channel. This kind of layout would make it simple to switch to something like a motor driver which might have different conditions or an LED driver that was active low instead of active high. This is a form of dynamic dispatching, more commonly seen in OOP. In C we would set these to be `const` function pointers at compile time, as modifiable function pointers are frowned on by MISRA.
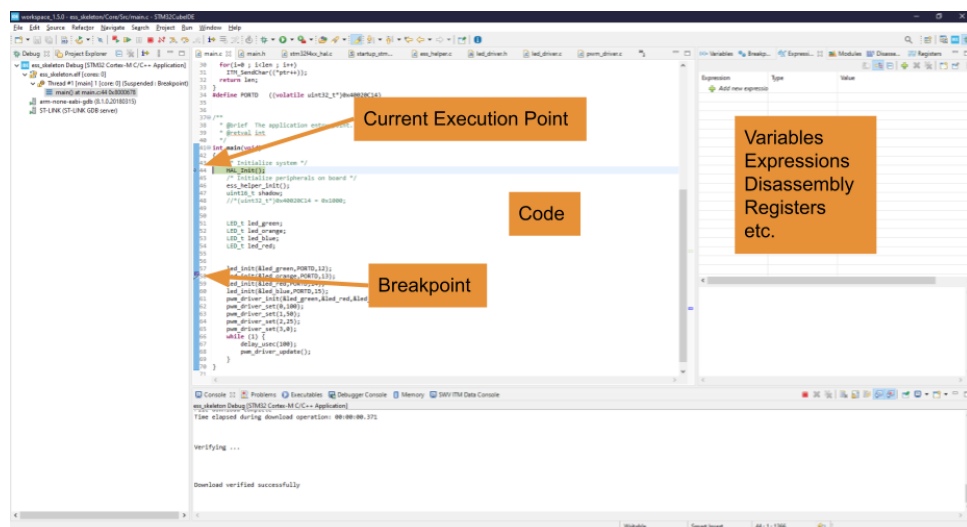>
> ```c
> struct pwm_control
> {
>   LED_t led;
>   void (*led_init)(LED_t * led, volatile uint32_t * port, uint32_t pin);
>   void (*led_on)(LED_t * led);
>   void (*led_off)(LED_t * led);
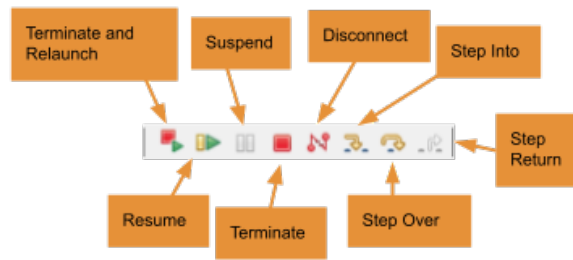> }
> ```

Task 3:

Debugging

One of the advantages of the STM32F4 development board is that it comes with a full featured debugger, something that a few years ago would cost a few thousand pounds. Debug mode can be entered by pressing the debug button ![debug button]. The general process on STM32CubeIDE is to build, download and then enter debug mode. Note that if you change any code, you need to exit debug mode, rebuild, download and enter debug mode again.

(a) Enter debug mode. The screen should change and look similar to this:

These are the debug controls:

(c) ⋆ The method of passing a pointer to the LED_t struct works, but it could be made more elegant. Suggest some further refactoring to reduce the coupling between the PWM driver and the LED_driver.

> **Solution:**
>
> A struct containing the LED struct itself and the function pointers could be passed to the PWM driver that encapsulates all the state associated with one LED channel. This kind of layout would make it simple to switch to something like a motor driver which might have different conditions or an LED driver that was active low instead of active high. This is a form of dynamic dispatching, more commonly seen in OOP. In C we would set these to be `const` function pointers at compile time, as modifiable function pointers are frowned on by MISRA.
>
> ```c
> struct pwm_control
> {
>   LED_t led;
>   void (*led_init)(LED_t * led, volatile uint32_t * port, uint32_t pin);
>   void (*led_on)(LED_t * led);
>   void (*led_off)(LED_t * led);
> }
> ```

Task 3:

Debugging

One of the advantages of the STM32F4 development board is that it comes with a full featured debugger, something that a few years ago would cost a few thousand pounds. Debug mode can be entered by pressing the debug button ![debug button]. The general process on STM32CubeIDE is to build, download and then enter debug mode. Note that if you change any code, you need to exit debug mode, rebuild, download and enter debug mode again.

(a) Enter debug mode. The screen should change and look similar to this:

These are the debug controls:

1. Add a breakpoint to a line that you want to stop execution at (for example, `pwm_driver_update();` in main.c). When the breakpoint is hit, the program stops executing and waits for you to start it again. To add a breakpoint, double click in the left hand gutter which will add a stop point to the line in question.

2. Click the 'Resume' button, which will make it run and then stop.

3. Note that all the registers on the left hand side will alter, and the disassembly listing should move to the line in question.

4. Try single stepping through instructions, by clicking 'Step Into'. This shows the flow of the instructions. If you get too deep, click 'Step Return' or 'Step Over' to move a level back up. As you step through the instructions, observe the LEDs on the board - are they changing according to the instruction?

5. On a line that you are stopped on, hover over a variable e.g. `state.counter` - it should pop up with a window showing the current value of the variable.

(b) Watching LEDs is one way of debugging, but it is also possible to see (watch) expressions and registers.

To watch expressions, click 'Add new expression' in the Expressions window. Type in expressions (e.g. state.counter) and then press 'Enter'.

As you run through your code, watch how the value in here changes. Feel free to add anything else to the watch window that you are interested in.

(c) To see the value of a particular address in the memory, for example, PORTD. Turn to "Memory" window. Click "Add Memory Monitor", and type in 0x40020C14. You can see the value stored in this memory address as well as other adjacent addresses.

(d) ⋆⋆ [For the brave] STM32CubeIDE also has an interesting feature that displays the value of a variable in a graphical format (i.e. as a logic analyser). This takes a little bit of setup. First open the "SWV Data Trace Timeline Graph" through 'Window' → 'Show View' → 'SWV' → 'SWV Data Trace Timeline Graph'. Open the options dialog like so:

The data trace will look like this (do not forget to click 'Start Trace' before resuming your program):
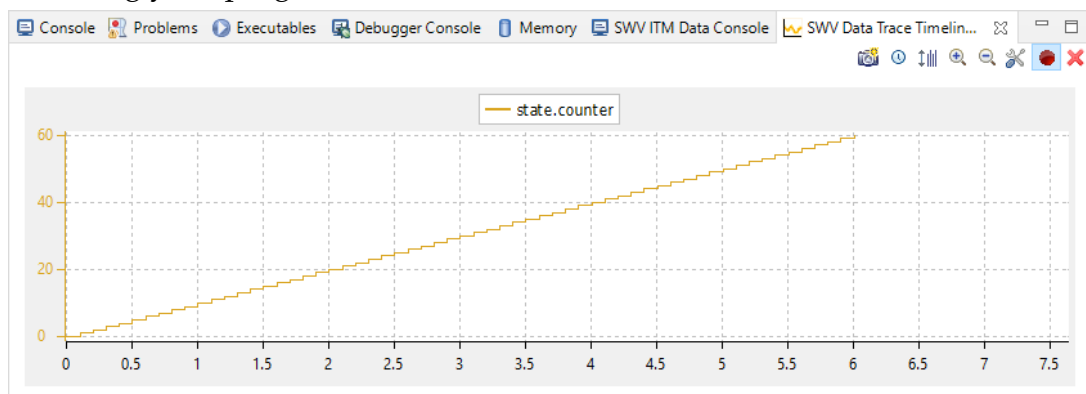
Task 4:

Hardware Timer

Using a software delay to control the timing of execution is particularly bad, as adding any other functions (such as checking a button or performing a calculation) will require all the delays to be recalculated. A better approach is to exploit one of the available timer peripherals and let this take care of timing in the background. We can check when the timer has expired and use this to yield more precise delays, regardless of any other instructions that have been executed.

The STM32F4 has a number of timer peripherals. We are going to use Timer 4, a

general purpose 16 bit timer. Rather than writing to memory addresses directly, like we did for the LED, we are going to use the supplied header files (stm32f4xx.h) which have all the registers defined.

In main.c, add the following define:

```
#include "stm32f4xx_hal_tim.h"
```

Add these functions to start and wait for the timer respectively:

```
// Initialize Timer 4

TIM_HandleTypeDef TIM_Handle; //define a handle
void TMR4_Init(void) {
    /* Enable clock for TIM4 */
    __TIM4_CLK_ENABLE();
    TIM_Handle.Instance = TIM4;    //Same timer whose clocks we enable
    // timer_tick_frequency = 16000000 / (0 + 1) = 16000000
    TIM_Handle.Init.Prescaler = 0;
    /* Count up */
    TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    /*
    Set timer period when it must reset
    First you have to know max value for timer
    In our case it is 16 bit = 65535
    Frequency = timer_tick_frequency / (TIM_Period + 1)
    If you get Period larger than max timer value (in our case 65535),
    you have to choose larger prescaler and slow down timer tick frequency
    */
    TIM_Handle.Init.Period = XXXX;// <your value here>
    TIM_Handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    TIM_Handle.Init.RepetitionCounter = 0;
    TIM_Handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    /* Initialize TIM4 */
    HAL_TIM_Base_Init(&TIM_Handle);
    /* Start count on TIM4 */
    HAL_TIM_Base_Start_IT(&TIM_Handle);
}

// Loops until the timer has expired
void TMR4_WaitForExpiry(void)
{
    // Check the flag. When the timer is expired, the flag is SET.
    while(__HAL_TIM_GET_FLAG(&TIM_Handle, TIM_FLAG_UPDATE) == RESET){}
    // Reset flag for next expiry
    __HAL_TIM_CLEAR_FLAG(&TIM_Handle, TIM_FLAG_UPDATE);
}
```

(a) Calculate a timer period register that will give a timer frequency of 10 kHz.

> **Solution:**
>
> We want 10 kHz from an 16 MHz clock. This means that we need to divide by a factor of 1600. We set `TIM_BaseStruct.TIM_Period = 1599;` to achieve this.

(b) In your `main()` function initialize the timer, and then instead of waiting for a delay, wait for the timer to expire. Show that the operation is identical to the previous software based delay.

```
// initialize hardware timer
TMR4_Init();
while (1) {
        TMR4_WaitForExpiry();
        pwm_driver_update();
}
```

(c) ⋆ Refactor your code to move this hardware based timer delay into a proper library that has the same API to the software delay library so they can be used interchangeably.

Task 5:

Interrupt Driven Timer

Although the hardware timer is more accurate, it still is not a good solution as it uses 100% of the CPU whilst waiting for the timer to expire in the current implementation. If it is doing something else and not checking the timer overflow flag frequently enough, it could also lead to jitter for the PWM output. For LEDs, this is probably not going to be noticeable, but for something like a servo motor, this could be a major issue. A far better approach is to use an interrupt on TMR4 when it overflows to signal to the main loop that it is done.

Add the following include to main.h, which has the definitions for the interrupt handlers.

```
#include "stm32f4xx_it.h"
```

The timer initialization code is modified to also enable the NVIC (interrupt controller) at the same time:

```
// Initialize Timer 4 for interrupts
void TMR4_Init_ISR(void) {
    /* Enable clock for TIM4 */
    __TIM4_CLK_ENABLE();
    TIM_Handle.Instance = TIM4;    //Same timer whose clocks we enable
    // timer_tick_frequency = 16000000 / (0 + 1) = 16000000
    TIM_Handle.Init.Prescaler = 0;
    /* Count up */
    TIM_Handle.Init.CounterMode = TIM_COUNTERMODE_UP;
    /*
    Set timer period when it must reset
    First you have to know max value for timer
    In our case it is 16bit = 65535
    Frequency = timer_tick_frequency / (TIM_Period + 1)
    If you get Period larger than max timer value (in our case 65535),
    you have to choose larger prescaler and slow down timer tick frequency
    */
    TIM_Handle.Init.Period = XXXX;// <your value here>
    TIM_Handle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
```

```
    TIM_Handle.Init.RepetitionCounter = 0;
    TIM_Handle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    /* Initialize TIM4 */
    HAL_TIM_Base_Init(&TIM_Handle);
    /* Start count on TIM4 */
    HAL_TIM_Base_Start_IT(&TIM_Handle);
    /* Enable the TIM4 gloabal Interrupt */
    HAL_NVIC_SetPriority(TIM4_IRQn, 0, 1);
    HAL_NVIC_EnableIRQ(TIM4_IRQn);
}

// This is triggered when the counter overflows
void TIM4_IRQHandler(void)
{
    //In case other interrupts are also running
    if (__HAL_TIM_GET_FLAG(&TIM_Handle, TIM_FLAG_UPDATE) != RESET)
    {
        if (__HAL_TIM_GET_ITSTATUS(&TIM_Handle, TIM_IT_UPDATE) != RESET)
        {
            __HAL_TIM_CLEAR_FLAG(&TIM_Handle, TIM_FLAG_UPDATE);
            /*put your code here */
        }
    }
}
```

(a) In your `main()` function initialize the timer and then enter an infinite loop. Show that the operation of the leds is identical to previous solutions.

(b) Enter debug mode and check that the interrupt handler is indeed running.

(c) Modify your main loop to alter PWM state every time the button is pressed.

Task 6:

⋆**Display**

Ultimately, the purpose of this work is to create a module that can satisfy **R6** (Display of pallet status). Flashing an LED will work, but it is unlikely to impress the client. The marketing team have decided that it would be good if we could display the tilt of the device visually using the four LEDs (up, down, left, right). The more detailed requirements are:

- R6.1 When the board is perfectly horizontal, all LEDs should be off

- R6.2 When the board is tilted, it should indicate the degree of tilt by controlling the brightness of the LEDs

- R6.3 When the board is tilted beyond 45 degrees on any axis, that respective LED should have a brightness of 100%

At the moment, we don't even have the ability to measure the tilt of the board! The acquisition and calculation of the tilt of the device will be handled by **R1** - this task is just concerned with the display itself. However, we can start to build the display independently. This embraces the software engineering concepts of loose coupling, programming to interfaces and modularity.

Write a display driver with the following API:

```c
// Initialize the tilt display using PWM driver
void display_init(void);
// Display a two-dimensional tilt.
// x_tilt varies from -90 deg to +90deg. y_tilt varies from -90 deg to +90deg.
void display_tilt(int8_t x_tilt, int8_t y_tilt);
```