ESS: Lab 1

Problem Statement

A project has arisen from a logistics company. The company ships a wide variety of products, ranging from fresh fruit and vegetables to paintings and scientific equipment across the globe. If an item arrives damaged then the company has to pay compensation, via their insurance company. The number of claims made in the past two years has increased dramatically, resulting in a consequent increase in their insurance premiums. This is affecting the profitability of their business and their relationships with their clients. To address this state of affairs, the CTO of the company has come up with the idea of a 'smart-pallet', a sensor-rich platform that will be embedded within a conventional pallet and provide diagnostics on the journey that a particular consigment has been on.

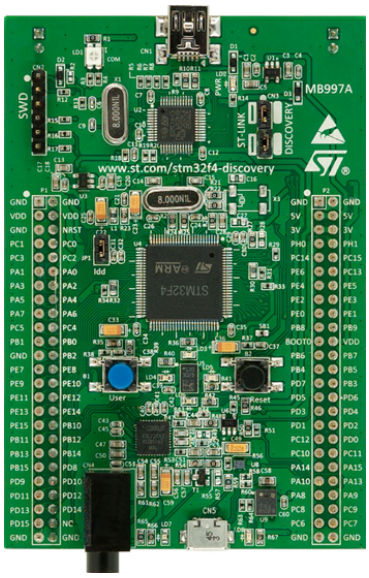
The key high level requirements are:

- R1: Measure vibration
- R2: Measure impact
- R3: Measure pallet tilt angle
- R4: Measure pallet storage temperature
- R5: Measure pallet humidity
- R6: Simple user display of pallet status (ok/damaged)
- R7: Logging capability
- R8: Wireless upload capability for IoT integration
- R9: Battery powered, long-life essential
- R10: Extremely low cost (U$3 in quantity)
- R11: Rugged integration to IP67 standards

As the software team, you are required to make a rapid prototype to demonstrate the feasibility of the approach and impress the client. The hardware team are building a first hardware prototype, but it will not be ready for two weeks. In order to work in parallel, you will start coding with the intention of being able to readily shift onto the hardware platform when it becomes available.

Development Kit

To make a rapid start on our application, we are going to use the STM32F4 Discovery Board. This is an evaluation kit/development board made by ST Microelectronics, featuring an ARM M4 Cortex processor. The ARM M4 Cortex processor is essentially an ARM M3 Cortex processor with the addition of a FPU (floating point unit) and associated DSP (digital signal processing) instructions. It is a very powerful processor, and for the end application, is way over spec. However, at £11, it is an inexpensive way of doing rapid prototyping.



- STM32F407VGT6 microcontroller featuring 32-bit ARM Cortex-M4F core, 1 MB Flash, 192 KB RAM in an LQFP100 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
- LIS302DL or LIS3DSH ST MEMS 3-axis accelerometer
- MP45DT02, ST MEMS audio sensor, omni-directional digital microphone
- CS43L22, audio DAC with integrated class D speaker driver
- Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
- Two push buttons (user and reset)
- Extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing

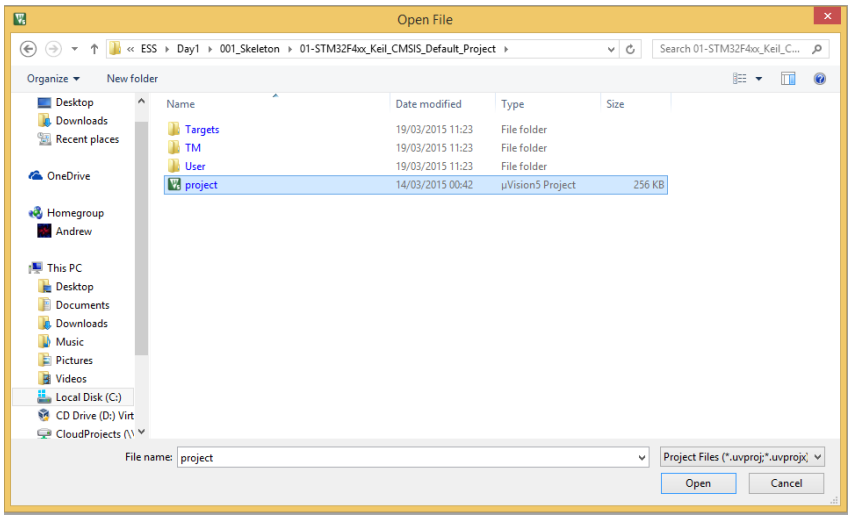
Keil MDK 5

To write the C code for the microcontroller, we are going to use Keil (ARM) MDK 5. This consists of a C-compiler, a debugger and an IDE. It is supported by ARM, so is widely used in industry (along with competitors like IAR). Unfortunately, it (along with IAR) only runs in Windows, so we have to run it in a Virtual Machine on the Macs[1]. We will be using Keil MDK during the week to build and test our application.

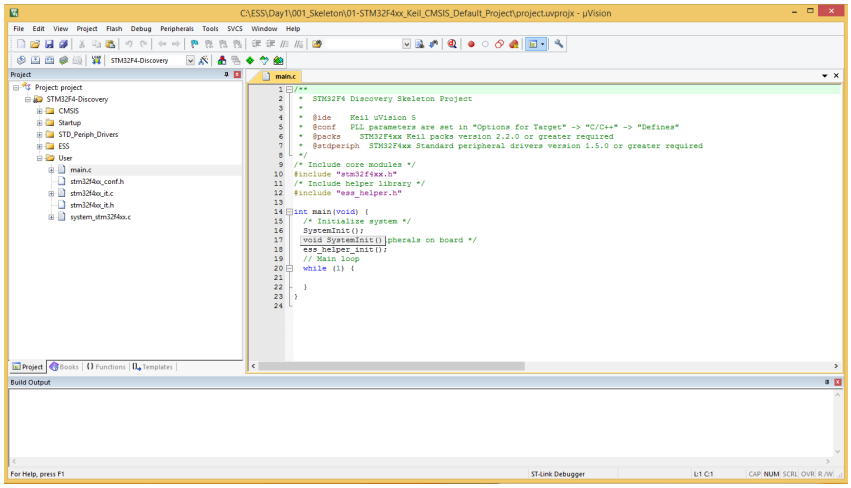
Task 1:

Initial bringup

A skeleton application ESS/labs/day1/001_skeleton has been created. This is a project which has been setup with the correct libraries and some basic initialization. Navigate to ESS/labs/day1/001_skeleton/01_STM32F4xx_Keil_CMSIS_Default_Project and double-click the μVision project called project.uvprojx.



This should open Keil uVision IDE, which should look like this:



[1]You are welcome to install Keil MDK on a Windows laptop - we are using the license free version - ask for help if you are, as it is pretty quirky

You can now try and build the project (use the second button from the left, or press F7)

If all goes well, your project should build without any errors, as shown in the build window in the bottom of the window. If there are any errors, ask for help to get the initial project up and running.

```
Build Output
Build target 'STM32F4-Discovery'
assembling startup_stm32f40_41xxx.s...
linking...
Program Size: Code=2344 RO-data=424 RW-data=36 ZI-data=1028
FromELF: creating hex file...
".\Targets\STM32F4_Discovery\project.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
```

You can now flash (download) the code to the discovery board, by clicking the 'load' button:

It should program the board and end up with the following output:

```
Project | Books | {} Functions | {}. Templates |
Build Output
Build target 'STM32F4-Discovery'
assembling startup_stm32f40_41xxx.s...
linking...
Program Size: Code=2344 RO-data=424 RW-data=36 ZI-data=1028
FromELF: creating hex file...
".\Targets\STM32F4_Discovery\project.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
Load "C:\\ESS\\Day1\\002_LED\\01-STM32F4xx_Keil_CMSIS_Default_Project\\Targets\\
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 14:16:44
```

Again, if it throws up an error (like not being able to find the programmer, which could be due to the Virtual Machine not being able to see the USB device), just ask for help!

Your board is now running the code that has been written, and should be doing... nothing. The four user LEDs in the middle of the board should all be off. We will now try and write something useful in the next steps.

First Steps: Basic display

Task 2:

Making something happen

Our first task will be to tackle **R6:** 'Simple display of pallet status'. We are going to use the four on-board LEDs as a display. The advantage of starting with this requirement is that customers are always impressed by flashing lights, as it gives a visual indication that the hardware is actually working. Consulting the datasheet, we can see that the LEDs are connected to Port D 12 (PD12), PD13, PD14 and PD15. They need to be driven high (written with a 1) to turn them on or driven low (written with a 0) to turn them off.

By digging very deep in the datasheet for this particular microcontroller, we find that memory location `0x40020C14` corresponds to the output register for Port D[2]. In C, we can write to a register at a known address by treating it as a pointer and then dereferencing it.

Change your main code to the following:

```
int main(void) {
        /* Initialize system */
        SystemInit();
        /* Initialize peripherals on board */
        ess_helper_init();
        // Set all the LEDs to on.
        *(uint32_t*)0x40020C14 = 0xF000;
        while (1) {

        }
}
```

In the new line `*(uint32_t*)0x40020C14 = 0xF000;`, we are making a pointer from the address (`uint32_t *`), dereferencing it (`*`) and then setting the top bits (PD12 to PD15) to 1. Build and program this into your device. What happens?

Solution:

All LEDs should turn on and stay on.

(a) Modify your code to turn on only one LED at a time to determine which color LED is connected to each port pin.

Solution:

- PD12: LED_GREEN
- PD13: LED_ORANGE
- PD14: LED_RED
- PD15: LED_BLUE

[2]Port D has a bunch of other registers that can set the mode, the direction of the pins, the input state, the speed of the pins etc. More on this later.

Task 3:

Writing a simple LED driver

We know that we can control individual LEDs, but the mechanism we have for doing so is not very programmer friendly. We want to build a Hardware Abstraction Layer (HAL) to shield us from the details of which register to address and which bit to set/clear.

(a) List some reasons why using a HAL is a good idea

> **Solution:**
>
> - A HAL leads to decoupling between the software and the hardware being used.
> - It is relatively straightforward to port to a different platform by modifying code in one place.
> - A properly written HAL can make testing easier.

(b) At a minimum, we need to do the following:

- Initialize an LED.
- Set an LED (turn it on).
- Clear an LED (turn it off).

As an API, we start with the following to turn on and off the green LED.

```
// Initialize the green LED.
void led_green_init(void);
// Turn the green LED on
void led_green_on(void);
// Turn the green LED off
void led_green_off(void);
```

To make life easy, start by writing these prototypes directly in the main.c file above the function definition for `main(void)`. What should the default behaviour be for your LEDs when they are initialized? What is your justification for this? Should it be the same for all peripherals?

> **Solution:**
> A sensible default action is to turn the LED off. This is extremely important for actuators like motors which if initialized in the wrong state could cause injuries or death. There are some cases in which we want to initialize something as being on as a default action, such as powering up a slave device.

(c) Write your code and check that everything works i.e you can turn on the Green LED using the following code:

```
        // Main loop
        led_green_init();
        led_green_on();
        while (1) {

        }
```

Solution:

```
// ESS: Day 1_003
// Initialize the green LED.
void led_green_init(void)
{
        *(uint32_t*)0x40020C14 = 0x0000;
}
// Turn the green LED on
void led_green_on(void)
{
        *(uint32_t*)0x40020C14 = 0x1000;
}

// Turn the green LED off
void led_green_off(void)
{
        *(uint32_t*)0x40020C14 = 0x0000;
}
int main(void) {
        /* Initialize system */
        SystemInit();
        /* Initialize peripherals on board */
        ess_helper_init();
        // Main loop
        led_green_init();
        led_green_on();
        while (1) {

        }
```

(d) Write the following code and download to the board:

```
        // Main loop
        led_green_init();
        while (1) {
          led_green_on();
          led_green_off();
        }
```

What happens? Why? [hint: think about `volatile`]

Solution:

The LED remains off. Why is it not flashing? Is it running too fast for the eye to see? No. The compiler has looked at the instructions in the while loop and decided they are pointless.

```
*(uint32_t*)0x40020C14 = 0x1000;
*(uint32_t*)0x40020C14 = 0x0000;
```

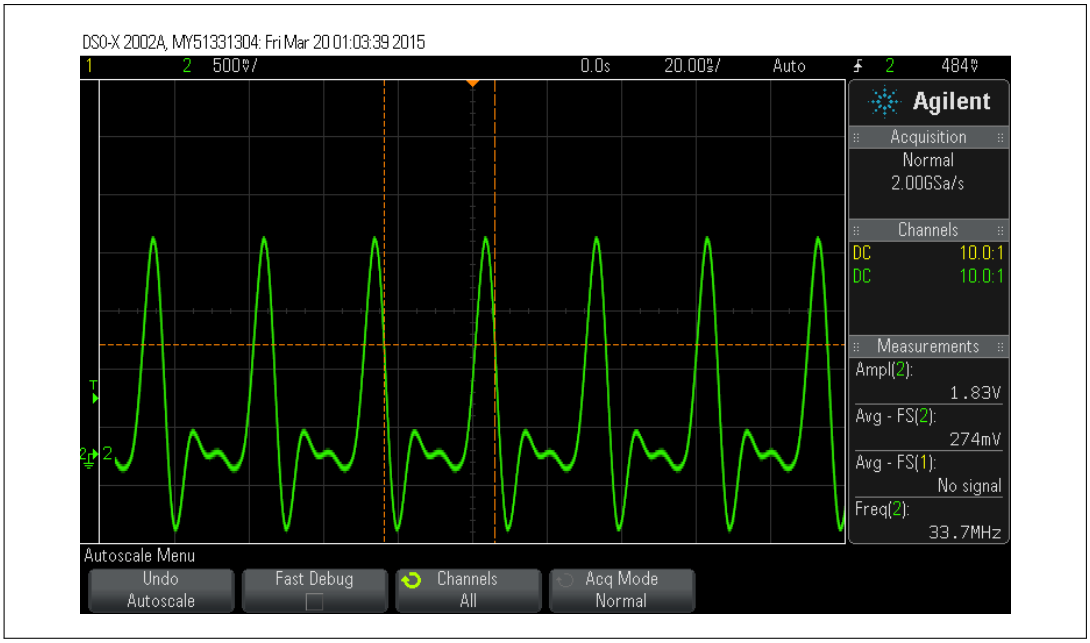
The solution to this is to use the `volatile` keyword to tell the compiler that this is actually something external to the programming model, so should not be optimized away. If the code is changed to:

```
// Use a define to make it cleaner
#define PORTD   ((volatile uint32_t*)0x40020C14)

// Initialize the green LED.
void led_green_init(void)
{
        *PORTD = 0x0000;
}
// Turn the green LED on
void led_green_on(void)
{
        *PORTD = 0x1000;
}

// Turn the green LED off
void led_green_off(void)
{
        *PORTD = 0x0000;
}
```

However, when we run this, the LED still remains off! This is because it is now running too fast to actually turn the LED on. If we connect up an oscilloscope to PD12, we can indeed see that this is the case and it is turning on and off the pin at 33 MHz. This waveform is far from square because it has insufficient current drive capacity to overcome the capacitance of the pin and associated circuitry.



(e) The solution we have at the moment is pretty hardcoded, which can make future modifications more challenging. Instead of using constants like `0x1000` to set a particular LED, a more elegant way is to construct the constant, by passing in the pin number. This can be stored locally in a variable `led_green_pin`. Then, when we call `led_green_on()` and `led_green_off()` these routines will work out the constant needed. [hint, use shifts e.g. `mask = 0x01 << bits_position;`]

```
// this stores the led pin we are going to use
// we declare as static to restrict the scope to this file
static uint32_t led_green_pin;
// Initialize the green LED.
// Supply a pin number between 0 and 15
void led_green_init(uint32_t pin);
// Turn the green LED on
void led_green_on(void);
// Turn the green LED off
void led_green_off(void);
```

Task 4:

Generic LED driver

The LED driver we have at the moment works but is pretty ugly. For example, writing LED drivers for each colour LED would lead to a lot of duplication, violating the DRY (don't repeat yourself) software engineering principle. A better way would be to decouple the logic from the actual pin. We could then supply the name of the LED to be turned on/off to a standardized API. One of the best ways of doing this is to use an abstract datatype (ADT) to carry around the state/configuration of each LED. In C, we can make an ADT by making a `struct` containing all the state we need.

```
// Use a define for the address of the PORTD output register
```

```
#define PORTD   ((volatile uint32_t*)0x40020C14)
// Generic struct to encapsulate LED state.
struct LEDstruct
{
        volatile uint32_t * port; // the LED register
        uint32_t pin;  // 0..15
};
```

It will get pretty painful writing `struct LEDstruct` all the time, so we can create our own 'type' like so:

```
// Use a define for the address of the PORTD output register
#define PORTD   ((volatile uint32_t*)0x40020C14)
// Generic struct to encapsulate LED state.
struct LEDstruct
{
        volatile uint32_t * port; // the LED register
        uint32_t pin;  // 0..15
};
// Typedef for an LED struct - this is a shorthand
// that defines a new type for encapsulating LED state
typedef struct LEDstruct LED_t;
```

Our API now looks like the following (again, for now, we just are writing in main.c).

```
// Function Declarations
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);
```

(a) Write the code to turn on and off any LED. What was challenging about this?

Solution:

The challenge here is that we are modifying individual bits, but affecting the shared (global) register. There are two ways of handling this - either reuse the port register itself or shadow it by creating a shadowed variable. We will take the first approach, but in many cases, the second approach leads to cleaner, more decoupled code, especially if something else is sharing that port register.

```
// Use a define for the address of the PORTD output register
#define PORTD   ((volatile uint32_t*)0x40020C14)
// Generic struct to encapsulate LED state.
struct LEDstruct
{
        volatile uint32_t * port; // the LED register
        uint32_t pin;  // 0..15
};
// Typedef for an LED struct - this is a shorthand
// that defines a new type for encapsulating LED state
```

```
typedef struct LEDstruct LED_t;

// Function Declarations
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);

// Function Definitions

// Initialize an LED, start in off-state
void led_init(LED_t * led, volatile uint32_t * port, uint32_t pin)
{
        led->port = port;
        led->pin = pin;
        // and turn it off
        led_off(led);
}
// Turn the green LED on
void led_on(LED_t * led)
{
        *led->port |= 0x01 << (led->pin);
}

// Turn the green LED off
void led_off(LED_t * led)
{

        *led->port &= ~ (0x01 << (led->pin));
}



int main(void) {
        // create the green led ADT
        LED_t led_green;
        LED_t led_orange;
        LED_t led_blue;
        LED_t led_red;
        /* Initialize system */
        SystemInit();
        /* 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 their values
        led_on(&led_green);
        led_off(&led_green);
        led_on(&led_red);
        led_on(&led_blue);
        led_on(&led_orange);
        while (1) {

        }
```

(b) Write a simple delay function by using a `while` loop. Your function should have an API like this, giving about a 1000 msec (1 second) delay for an input of 1000.

```
void delay_msec(uint32_t delay);
```

Solution:

The solution is just to use a while/for loop which just spins the processor and does nothing. Again, the only trick is that we have to fight against the all too clever compiler and insert a dummy operation in the middle of the loop, just so the whole loop does not get optimized away.

```
// Delay for a multiple of 1 msec
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 <20000;k++)
                {
                        internal_fake--;
                }
        }
}

int main(void) {
        // create the green led ADT
        LED_t led_green;
        LED_t led_orange;
        LED_t led_blue;
        LED_t led_red;
        /* Initialize system */
        SystemInit();
        /* 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 their values
        led_on(&led_green);
        led_off(&led_green);
        led_on(&led_red);
        led_on(&led_blue);
        led_on(&led_orange);
        while (1) {
                led_on(&led_green);
                delay_msec(1000);
                led_off(&led_green);
                delay_msec(1000);
        }
}
```

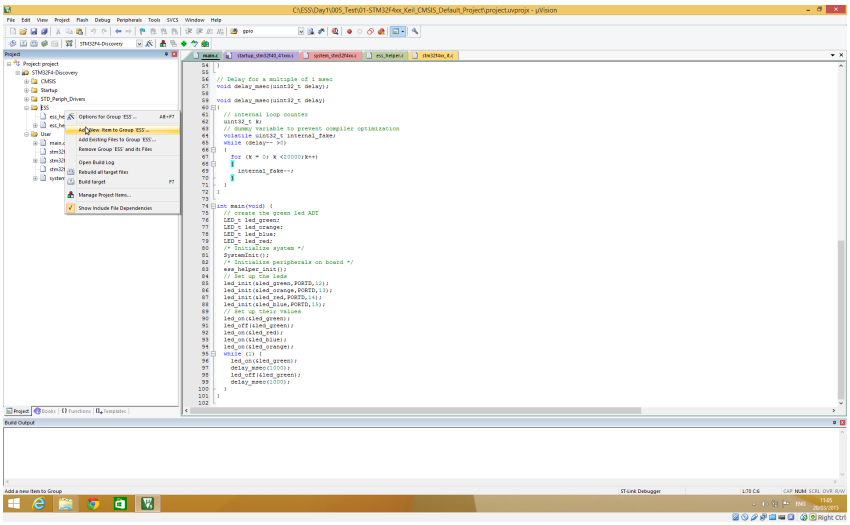
Task 5:

Refactoring

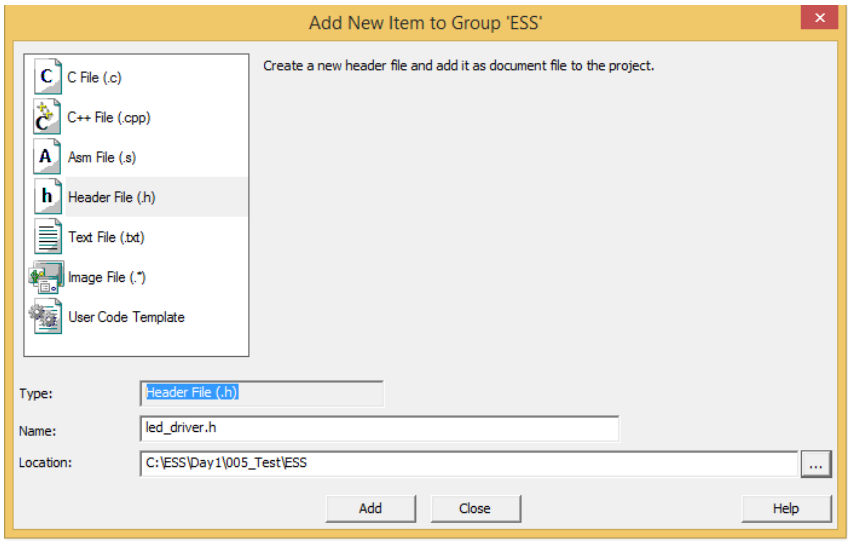
Now that we have written the LED driver, we are going to try and make it cleaner, by moving all the logic for the driver into its own files.

(a) The first part is to *refactor* the code to make it clean. We are going to move all the LED code into two files: led_driver.h for the function declarations (API) and led_driver.c for the function definition (implementation). Then instead of having a main.c that is littered with code for the led_driver, we will delegate this to its own module.

- **Step 1:** On the left hand side, in the project layout, expand the 'ESS' group. Right-click and in the drop-down menu, click "Add new item to group ESS..."



Then create the led_driver.h file - note the name and that we have used a path one level up from the main project directory[3].



[3]This step can be quite tricky to get right, and the compiler/linker might complain it can't find the file - just ask for help

Populate the led_driver.h file so it looks something like this:

```
// file: led_driver.h

/**
 *        LED Driver
 *
 *        @author          Andrew Markham
 *        @version          v1.0
 *        @ide                 Keil uVision
 *        @license         GNU GPL v3
 */
#ifndef LED_DRIVER_H
#define LED_DRIVER_H

#include <stdint.h>

// Generic struct to encapsulate LED state.
struct LEDstruct
{
        volatile uint32_t * port; // the LED register
        uint32_t pin;  // 0..15
};
// Typedef for an LED struct - this is a shorthand
// that defines a new type for encapsulating LED state
typedef struct LEDstruct LED_t;

// Function Declarations
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);

#endif
```

- **Step 2:** Create the led_driver.c file in the same way as above. Copy your implementation to this file, in the sections shown below.

```
// file: led_driver.c

 #include "led_driver.h"

// Initialize an LED, start in off-state
void led_init(LED_t * led, volatile uint32_t * port, uint32_t pin)
{
        // your implementation
}
// Turn the LED on
void led_on(LED_t * led)
{
        // your implementation
}

// Turn the LED off
void led_off(LED_t * led)
{
        // your implementation
}
```

- **Step 3:** You now need to tell the main.c file that you are using an API defined in led_driver.h, using the following include statement:

```
/* Include core modules */
#include "stm32f4xx.h"
/* Include helper library */
#include "ess_helper.h"
// led driver
#include "led_driver.h"
```

- **Step 4:** Compile and run - if you get no errors, your code should be doing exactly as it did before, just a lot cleaner.

Task 6:

Reacting to input

The blue button is on PA0 (the other button is a reset button). It has been configured by `ess_helper_init();` as an input pin. The register we need is `GPIOA->IDR;`, which has been defined for us in the STM32F4 header files.

(a) When the button is pressed, turn on the green led.

(b) Once you have a button press working, see if you can use it to cycle through states, stop/start LEDs flashing etc.

Solution:

The following turns on the green led while the switch is pressed.

```
int main(void) {
      // shadow the PA input data register
      uint16_t shadow;
      // create the green led ADT
      LED_t led_green;
      LED_t led_orange;
      LED_t led_blue;
      LED_t led_red;
      /* Initialize system */
      SystemInit();
      /* 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 their values
      led_on(&led_green);
      led_off(&led_green);
```

```
        led_on(&led_red);
        led_on(&led_blue);
        led_on(&led_orange);
        while (1) {
                shadow = GPIOA->IDR;
                if (shadow & 0x01)
                {
                        led_on(&led_green);
                }
                else
                {
                        led_off(&led_green);
                }
        }
}
```

Task 7:

⋆ **Testing**

Testing the LED driver by flashing LEDs is painful and a labour intensive technique. It also doesn't lend itself well to regression testing. It would be better to test the logic of the LED driver separately, to check that it is doing what we think it should be doing.

The best way of doing this is to pull out the led_driver.h and led_driver.c files and cross-compile them to run natively (i.e. on the desktop). We can then write a test harness around these files to check the operation. This would take a while to setup[4], so we are just going to test on the hardware directly, but not watch the flashing lights.

We need some indication that a test has passed or failed. To do this, we can use the ETM (embedded trace macrocell) to redirect `printf()` via the USB to a trace window. This takes a bit of magic to work out, but a skeleton which prints via debug is available.

[4]however, you are welcome to do it, if you want. Just use gcc or your favourite C compiler.