ELC018 Real-time Software Engineering

Writing a device driver

The microcontroller on the STM Discovery target board contains an ARM M3 core and a number of peripherals. The target board has six parallel input/output ports (indicated by the letters A to F) each having 16 bits. The board has eight LEDs connected to pins 8 to 15 of port E.

In this lab, the initial code to light the LEDs is given. You will need to alter the code to convert it to C++ device driver format.

Ports on the Discovery board

Port E is mapped to memory location `0x48001000`. The port has a number of registers, and those most relevant to us are shown below. The mode register controls which of the 16 port pins are used for input and which for output. The speed register determines how quickly an output will change, the pull-up/pull down register controls what type of resistor is connected to the port and the input and output registers are the current values being received or supplied. A common way of representing such a set of ports is to use a C++ class.

```
class GPIOs
{
  public:
    volatile uint32_t MODER;        // GPIO port mode register
    ...
    volatile uint32_t OSPEEDR;      // GPIO port output speed register
    volatile uint32_t PUPDR;        // GPIO port pull-up/pull-down register
    volatile uint32_t IDR;          // GPIO port input data register
    volatile uint32_t ODR;          // GPIO port output data register
    ...
};
```

The LEDs are connected to port E. As this port is being used for output, the mode needs to be correctly set. The output speed is set to high (although the operation of the program will not be affected by setting it to low) and pull down resistors are selected (although changing this to pull up will not affect the operation). The locations of individual LEDs in the 16 bits of the output data register are shown in the table below.

Port E (base `0x48001000`)	bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
LED number	LED6	LED8	LED10	LED9	LED7	LED5	LED3	LED4
LED colour	Green	Orange	Red	Blue	Green	Orange	Red	Blue

Enter the code

To start the MDK-ARM editor you will to run µVision 5 (choose **All Programs** from the Windows Orb and look for **Keil uVision 5**). The project for the laboratory is available for download from the Learn server. It is a zip file named **DD.zip** that you will need to extract to a location to which you have write access. You should then choose **Project->Open Project…** from µVision 5 and navigate to the directory where you have downloaded the files and select `DD.uvpro`. Ignore the error message about the project being for an earlier version of the tools. You will notice that the project contains a number of support files and this is the target level software that supports that Discovery board. The project is also already configured so that the program is correctly mapped to the memory available on the board.

The only code you will need to consider is that under the **User** group, which you will find by opening the project in the left hand pane of the editor environment. You will notice that there are three files, `main.cpp`, `DD.cpp` and `DD.h`. The `main.cpp` source file has the following code in its `main()` function.

```
// Drive the on-board LEDs
int main(void)
{
  // STM32F3 Discovery Board initialization
  board_startup();

  // initialize port E for output
  GPIO_E->MODER = (uint32_t) 0x55555555;  // output
  GPIO_E->OSPEEDR = (uint32_t) 0xFFFFFFFF;  // high speed
  GPIO_E->PUPDR = (uint32_t) 0xAAAAAAAA;  // pull down resistors

  // turn all LEDs off
  GPIO_E->ODR = (uint16_t) 0x0000;

  while (1) {

    // turn on LED3 to LED7 in turn, with a 100ms gap between each
    GPIO_E->ODR = GPIO_E->ODR | (uint16_t) LED3;
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR | (uint16_t) LED4;
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR | (uint16_t) LED5;
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR | (uint16_t) LED6;
    HAL_Delay(1000); // 1000ms delay

    // turn off LED3 to LED7 in reverse order, with a 100ms gap between each
    GPIO_E->ODR = GPIO_E->ODR & ~((uint16_t) LED6);
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR & ~((uint16_t) LED5);
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR & ~((uint16_t) LED4);
    HAL_Delay(1000); // 1000ms delay
    GPIO_E->ODR = GPIO_E->ODR & ~((uint16_t) LED3);
    HAL_Delay(1000); // 1000ms delay
  }
}
```

The `DD.cpp` file contains nothing (apart from including DD.h) and DD.h has the definition of the GPIO C++ class as well as the mappings of LEDs to specific bits on port E.

Building and downloading the code to the Discovery board

From the **Project** menu choose **Build target** and the **Build Output** pane (bottom left) will display messages about assembling and linking. There should be no error messages.

To download to the target, choose **Start/Stop Debug Session** (or press **CTRL-F5**). If a warning message appears about the code size limit, just select OK. If there is a warning about the target not being found, try **Project->Options for target**, select the **Debug** tab, click on **Settings** next to **ST-Link Debugger** and make sure **SW** is selected in the **Port** pull-down menu.

Before running the code, make sure the ribbon cable is disconnected from the washing machine. Unplug it at the washing machine end rather than from the target board.

Debug options

The program can be run from the debugger by selecting **Debug->Run** or pressing **F5**. To stop execution, select **Debug->Stop**. If you wish to re-run the program, this may be possible by selecting **Debug->Reset CPU** although whether this is successful will depend on the state of the target board. Pressing **CTRL-F5** to stop the debug session and then **CTRL-F5** to restart is more likely to be successful.

Rather than just running code, you may wish to single step through lines of code. Pressing **F10** repeatedly is normally the best way to do this. To view the values of local variables, you should open the **Call stack + Locals** window by selecting **View->Call Stack Window**. To view global variables, choose **View->Watch Windows->Watch 1**, but you will need to type in the name of the variable whose value you wish to see.

You may also wish view port values. You can choose **View->System Viewer->GPIO->GPIOE** to see the values being sent to the port connected to the LEDs. When the window opens, expand the **ODR** register so that you see the values of the individual bits.

Writing a device driver

The code as written does not correspond to the separation into layers as discussed in the lecture. As a first step to achieving this, the device drivers (that make up the device layer) need to be placed in a file separate from the application layer. In particular, the device layer will now be placed in the files `DD.cpp` and `DD.h`. The application layer will be placed in in the file `main.cpp`.

A device driver needs to encapsulate the low-level functionality of the interaction with the LED hardware. Once the device driver has been written, the following `main()` function should be able to work with the device driver to light the first two LEDs.

```
// Drive the on-board LEDs
int main()
{
  // STM32F3 Discovery Board initialization
  board_startup();

  // initialize port E for output
  GPIO_E->MODER = (uint32_t) 0x55555555;  // output
  GPIO_E->OSPEEDR = (uint32_t) 0xFFFFFFFF;  // high speed
  GPIO_E->PUPDR = (uint32_t) 0xAAAAAAAA;  // pull down resistors

  // define LED objects, all in their initial off state
  LEDs Red(LED3);
  LEDs Blue(LED4);
  LEDs Orange(LED5);
  LEDs Green(LED6);

  // light the first red LED
  Red.Set();
  HAL_Delay(1000); // 1000ms delay

  // light the first blue LED
  Blue.Set();
  HAL_Delay(1000); // 1000ms delay

  // turn off the first blue LED
  Blue.Reset();
  HAL_Delay(1000); // 1000ms delay

  // turn off the first red LED
  Red.Reset();
  HAL_Delay(1000); // 1000ms delay
}
```

To implement the device driver for the LEDs you should write a new C++ class in `DD.h` (call it `LEDs`) and provide member functions for this class in `DD.cpp`. The `LEDs` class should have the following member functions and members.

1. A constructor function to define the bit location of the LED on Port E and switch off the LED.
2. A function `Set()` that turns on the LED .
3. A function `Reset()` that turns off the LED .
4. A function `Get()` that returns the status of the LED (`on` or `off`).
5. A member that holds the bit location of the LED on port E.
6. A member that records whether the LED is `on` or `off`.