A simple device driver

1 Introduction

This lab is concerned with device driver development for the `LPC_2378_STK` development board.

Developing a device driver can be a complicated business. Even for a device as simple as a LED.

Consider the steps required to develop a driver for the `USB_LINK_LED` on this board.

1. *Identify the hardware features of the microcontroller that are used to control the device*
   A LED is an *output* device. It is a simple device that is either *on* or *off*. We can guess that it is associated with one of the GPIO (General Purpose I/O) pins of the `LPC_2378`. But which one? We could read the schematic. Download and open the schematic in a pdf viewer (e.g. acrobat reader or evince) and zoom in until you can read the details (about 300% for me). Find the `USB_LINK` pin on the schematic of the microcontroller. Notice that it is associated with P0.13. That means pin 13 in GPIO port 0. Actually, the pin is associated with several functions. We'll assume that the GPIO function is selected for this pin when we develop our device driver.

2. *Understand how to control the relevant hardware features from software*
   So now we need to know how to program the GPIO pins. We need to read the user manual for the `LPC_2378`. Notice by looking at page 16 that the `LPC_2378` communicates with its peripheral devices (I/O devices) by *mapping* them into its own *memory space*. The devices are said to be *memory-mapped*. This is convenient for the programmer, who can control devices simply by assigning values to the device addresses as though they were ordinary variables. Note that some other processors, e.g. the Intel range, require the programmer to control I/O devices by working with a dedicated set of *I/O ports* using special read and write instructions. These devices are said to be *I/O mapped*. We won't consider this approach in this module. There's more relevant information about programming the GPIO ports in Chapter 10 of the LP23xx user manual, see p.173 in particular.

3. The user manual tells us that the direction register for fast GPIO port 0 (`FIO0DIR`) is at memory address `0x3FFFC000`. This register is used to

determine, for each pin, whether it's an *input* pin (value 0) or an *output* pin (value 1). To control an LED we need an output pin. The manual also mentions a register (`FIO0PIN`) that controls the value of the pins. Its address is `0x3FFFC014`.

4. So, in order to configure pin 13 as an output pin, we need to set bit 13 of the direction register to the value 1. Once the pin is configured as an output pin, we can control the state of the pin by setting (assigning the value 1 to) or clearing (assigning the value 0 to) bit 13 in the `FIO0PIN` register. We should modify both of these registers without disturbing the value of the other bits in them. We need to understand how to manipulate individual bits in order to do this. Read the section on 'Common Bit Manipulation Techniques' in this Wikipedia article for more information. Once, we've gathered enough information from the documentation, we might try the following program to control the LED:

```
#include <stdbool.h>
#include <stdint.h>

typedef uint32_t volatile *pDeviceRegister_t;

#define FIO0DIR ((*(pDeviceRegister_t)0x3FFFC000))
#define FIO0PIN ((*(pDeviceRegister_t)0x3FFFC014))

#define LINK_LED_MASK ((1UL << 13))

void main(void) {
  FIO0DIR |= LINK_LED_MASK;     /* make P0.13 an output pin */
  while (true) {
    FIO0PIN |= LINK_LED_MASK;   /* set pin high (1)         */
    FIO0PIN &= ~LINK_LED_MASK;  /* set pin low  (0)         */
  }
}
```

5. In fact, there are so many device registers associated with the LPC2378 that the provider of the compiler has written a header file containing definitions for all of them, so that we don't have to write our own. The file `iolpc2378.h` defines `FIO0DIR` and `FIO0PIN`, along with all the other device registers. We can use these names in our programs by *including* `iolpc2378.h`.

2 In the lab

1. Download the file workspace.zip into a suitable directory either on a pen drive or in your University workspace. I suggest you call the directory `EN572/labs/lab02`. Unzip `workspace.zip`.

2. Start up EWARM and load the workspace `workspace/workspace.eww`.

3. Connect a `LPC-2378-STK` board to a USB port on your computer.

4. Download and debug the project `lab02`. Run it and observe its behaviour.

5. Carefully read the file `leds.c` and make sure that you understand how it works. It's a bit different from the program discussed in the introduction. Ask your lab tutor to explain anything that isn't clear to you.

6. Modify the `main` program so that it has exactly the same behaviour but uses the `ledToggle()` function instead of `ledSetState()`. Keep the size of your code as compact as you can.

7. Write your own function to toggle the link LED (call it `ledToggleXor()`). It should have the same behaviour as `ledToggle()` but it should be implemented using `FIO0PIN` and the exclusive or operator, `^`.

8. Modify the `leds.c` file to provide a similar driver for the `USB_CONNECT_LED`. Add suitable function prototypes to `leds.h`. Use your new driver in `main.c` to flash the `USB_CONNECT_LED` at the same rate as the `USB_LINK_LED`. Notice that you'll also need to add some code to `ledsInit()`.

9. Modify `main.c` so that your program flashes the `USB_CONNECT_LED` at half the rate of the `USB_LINK_LED`.

10. Define an enumeration type in `leds.h` called `ledIdentifier_t`. The type should contain two constants: `USB_LINK_LED` and `USB_CONNECT_LED`. You can base your code on the definition of `ledState_t` that is already present in `leds.h`. Add a function prototype to your interface to allow the user to select which LED to set by passing a led identifier as a parameter. For example,

    ```
        ledSetState(USB_LINK_LED, LED_ON)
    ```

    should turn on the link LED and

    ```
    ledSetState(USB_CONNECT_LED, LED_ON)
    ```

    should turn on the connect LED.

11. Implement your new function in `leds.c` and test it. Add a similar function `ledGetState` to `leds.c` to allow the user to get the current state of a specified led.

12. Modify the file `delay.c` so that the word `volatile` is removed from the declarations of `dly` and `dly100`. Download and debug the modified program. How has the behaviour of the program changed? If at all. Try changing the compiler settings in

    ```
        Project->Options->C/C++ Compiler->Optimizations
    ```

so that optimization is set to the high level. Download and debug the program again. How does the program behave now? Can you explain what's going on?. After you've thought about this, read Use `volatile` judiciously to help your understanding. Why is the approach to creating a delay that is taken by the function `dly100us()` so poor?

3 Other activities

1. Read Symbolic constants for a discussion of the use of `#define`, `enum` and `const` for defining constant values is C programs.
2. Develop your understanding of the IAR EWARM IDE by working through tutorials 1 and 2. Start up EWARM. Click on the tutorials icon on the introductory page, follow the links to tutorials 1 and 2.
3. See if you can find out roughly what proportion of the number of lines of code in a traditional OS (e.g. Windows, Unix, Mac OS) is contributed by the device driver code.
4. Based on your experience in this lab, explain why you are glad that your computer comes with an operating system that provides drivers for all of its devices.
5. Check that you understand the following:
   - memory-mapped I/O
   - use of `#define` and `#include`
   - how to give a name to a specific memory address
   - the use of `typedef` and `volatile`
   - the use of bit operators: `|` `&` `^` `~` and `<<`
   - the use of pointers and enumeration types
   - the use of `EWARM` to edit, compile, link, download and execute programs