Title: The arm Cortex M4, tools and debugging
Author: Craig Duffy 19/10/21
Module: Embedded Systems Programming
Awards: BSc Computer Science
Prerequisites: Some Linux and C skills, plus basic computer architecture

This worksheet will start programming the disco board doing simple things with LEDS and buttons. In order to understand the code we will first look at a little bit more of the CortexM4 architecture.

In order to do the practical work on this worksheet you will need to fork a copy of this repo - https://gitlab.uwe.ac.uk/c-duffy/esp-lab-3. You then should make sure that I can see it as it may be used to determine your mark.

Device drivers

Once you start getting interested in computers and programming you will start to hear a lot about device drivers. They are bit of code and configuration information for controlling a device. Generally, they are associated with an operating system, such as Windows 10 or Linux. They differ from the core operating system code as they provide the interface to the myriad of differing devices that the computer may connect to. So the device drivers will change over time, as new devices come and go, and they won't be in every instance of the operating system – there is no point in having the code if you don't have the device.

In this worksheet we are going to write a couple of very simple device drivers – the first simply has 1 bit output capabilities, it flashes an LED, the second is 1 bit input, it monitors a button. Strangely there are even simpler device drivers than these, ones which, although useful, essentially do nothing! If you look in the startup_stm32l475xx.c file you will notice the **default_handler** routine, this does nothing whatsoever! These sorts of routines are crucial to computers and their safe running.

In order to set up our device drivers we need a few conditions, we need to set up some time source (clocks), initialise the resource necessary for the device, and have some routines to manage the device (flash the LED, read the buttons). This is the same pattern for more complicated devices, although they are likely to have a more sophisticated resource set up and most require a de-initialisation phase to release their resources.

Clocks

The use of clocks and timers is central to computers and device control. Without a sense of time knowing what order events occurred is impossible, setting up alarms and time limits would be impossible. For example a network device driver would have no way of knowing whether a data packet was delayed unless it had some idea of the passage of time.

On the Cortex M4 there are a number of time sources - 4 to be precise. There are internal high and low speed clock sources – HSI and LSI, as well as external high and low speed sources – HSE and LSE. We have already used and initialised these clocks in our first example piece of code without realising it. When we linked our **main.c** with the file **startup_stm32l475xx.o** we called a routine called **SystemInit()**. This routine is part of CMSIS and is found in **Drivers/CMSIS/ Device/ST/STM32L4xx/Source** in the file **system_stm32l4xx.c**. This routine is responsible for setting up the system's clocks and

setting the all-important SYSTICK value. As the CPU and board can connect to a large number of different devices and protocols – BLE, WIFI, USB – it needs quite a number of different clock sources. The Disco board runs at 80MHz.

The Cortex range has been designed to save power as much as possible and each of the separate peripheral buses can be separately clocked and initialised and the CortexM4 exemplifies this. There are 3 peripheral buses on the SMT32 – APB1, APB2, AHB and AHB2. The APB buses deal with peripherals – **A**dvance **P**eripheral **B**us, the AHB – **A**dvanced **H**igh Speed **B**us – deals with memory devices and DMA. The GPIO pins we are dealing with are connected via the AHB2 bus and the standard library gives us commands to enable those specific peripherals on that bus.

```
__HAL_RCC_GPIOB_CLK_ENABLE();
```

There are **RCC** clock commands to enable and disable all of the buses and their peripherals – **RCC** stands for **R**eset and **C**lock **C**ontrol. In order to use these library features we need to include the library definitions and code - stm32l4xx_hal_rcc.[c/h]. This is done by changing a line in the Inc/stm32l4xx_hal_conf.h file - this file allows us to select which of the library files we will include. So editing the comments (/* */) on the following line

```
#define HAL_RCC_MODULE_ENABLED
```

means that the following instruction is given

```
#ifdef HAL_RCC_MODULE_ENABLED
        #include "stm32l4xx_hal_rcc.h"
#endif /* HAL_RCC_MODULE_ENABLED */
```

and also we need to make sure that the stm32l4xx_hal_rcc.o file is in the list of objects to be built
You will notice under the HAL_OBJS variable in the Makefile the rcc and gpio object files are included.

GPIO

The STM4 has 114 GPIO pins that can be used for a variety of purposes. Sometimes they will be used to support a specific peripheral, for example a serial port or Network Interface Chip. So in the case of a NIC, it may require a pin to detect whether the cable is plugged in, and one of the GPIO pins can be used for this, but in order to do so it would need to be configured to the specific requirements in the NIC device driver code. Our driver is much simpler in that our device is a simple LED which is, of course, an output device. Firstly we need to know which pins the LEDs on the board are connected to. Using the DISCO UM2153 manual we can find out what we need to know – you can either find them in the schematic or in the section Input/Output. From this we can find out that Pin 90 is connected to the green LED.

The GPIO pins are organised into a series of ports on the Cortex and each port has 16 pins. The banks are labelled GPIOA, GPIOB and so on up to GPIOG. So to find a specific pin you need to know which port it is in and which pin in that port. So the green LED2 is the 14th pin in port B – and is referred to as PB14. The standard library code has a series of routines which allow easy configuration of the various ports and their pins, this is

stm32l4xx_hal_gpio.[c/h]. This will require including a change in the Inc/stm32l4xx_hal_conf.h file to activate the stm32l4xx_hal_gpio.h file which will give us access to all the addresses of the pins and lots of settings that can be applied to them. This is achieved by editing the file Inc/stm32l4xx_hal_conf.h in the Lab3 Directory, the line to include the gpio header is uncommented.

```
#define HAL_GPIO_MODULE_ENABLED
```

Each port has a series of registers to control/configure the port, to read and write data to and from the pins. Access to the registers is made easier by the creation of a series of C structures that allow high level language access to them. For example in stm32l475xx.h the struct GPIO_TypeDef is created.

```
typedef struct
{
  __IO uint32_t MODER;       /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;      /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;     /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;       /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;         /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;         /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint32_t BSRR;        /*!< GPIO port bit set/reset  register,     Address offset: 0x18      */
  __IO uint32_t LCKR;        /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];      /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
  __IO uint32_t BRR;         /*!< GPIO Bit Reset register,               Address offset: 0x28      */
  __IO uint32_t ASCR;        /*!< GPIO analog switch control register,   Address offset: 0x2C     */

} GPIO_TypeDef
```

This allows us to access all of the registers for each port – the control registers CRL/CRH; 16 bit access data read write registers IDR/ODR, individual bit set and read registers BSRR and BRR and, finally a lock register to allow controlled individual access to the registers.

In the library there are a number of routines to allow us to initialise, read/write and lock the ports.

```
/* Initialization and de-initialization functions
*****************************/
void               HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef
*GPIO_Init);
void               HAL_GPIO_DeInit(GPIO_TypeDef  *GPIOx, uint32_t GPIO_Pin);


/* IO operation functions
*************************************************/
GPIO_PinState      HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void               HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
GPIO_PinState PinState);
void               HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t
GPIO_Pin);
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void               HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin);
void               HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
```

GPIO port routines

One of the first routines we will use is GPIO_Init(), but we will need to make sure that it knows which pins we wish to configure and exactly how we intend to configure them.

```
#define LED2_PIN                           GPIO_PIN_14
#define LED2_GPIO_PORT                     GPIOB
#define LED2_GPIO_CLK_ENABLE()
__HAL_RCC_GPIOB_CLK_ENABLE()
#define LED2_GPIO_CLK_DISABLE()
__HAL_RCC_GPIOB_CLK_DISABLE()
```

This shows that we are configuring pin14 (or green LED) we are giving it a clock speed on the bus, and a mode – Output Push Pull. We then call HAL_ GPIO_Init(). This code is placed in the LED2_Init() function in main.c

Once the port has been configured we can use the LED2_On() function to turn on the LED and LED2_Off() to turn off. We have to put a delay into the program otherwise it would happen so fast you wouldn't notice it. Both of these functions call the

```
HAL_GPIO_WritePin(LED2_GPIO_PORT, LED2_PIN, GPIO_PIN_RESET);
```

function either with GPIO_PIN_RESET (to turn off) or GPIO_PIN_SET (to turn on). There is also a LED2_Toggle() function which flips the state between on and off.

Exercises

1) Build, load and run the example program. Change it so that a) the delay is changed and b) that LED2_Toggle() is used instead of LED2_On and LED2_Off.

2) The Disco User Manual show on page 29 show us that the other LED (LED1) is on GPIO_PORTA and is GPIO_PIN_5. Using this information write some new functions – LED1_Init, LED1_On and LED1_Off which can flash the other LED – LED1.

Extra Credit

Change the code so that the program alternates between LED1 and LED2.

Create a data structure that can hold the LED details so that they can be passed to a generic set of LED functions – LED_Init(), LED_On() etc.

Getting Input

Getting input from the board is, as we shall see, much more complicated despite only being a 1 bit input. The Disco board has two switches, one (B2) that can be programmed, but the other one (B1) is a dedicated one used for resetting the system.

The buttons are, like the LEDs, connected to GPIO pins, and like the LEDs the system needs programming to know how to deal with them. As with the LEDs we can use the ST Micro standard library, its data structures and functions to program the switches. We can call HAL_GPIO_Init() with the following modifications to the GPIO_InitStructure. Firstly we need to know which port (PortC) the button is connected too and which pin (13)

```
#define BLUE_BUTTON_PIN                     GPIO_PIN_13
#define BLUE_BUTTON_GPIO_PORT               GPIOC
#define BLUE_BUTTON_GPIO_CLK_ENABLE()
__HAL_RCC_GPIOC_CLK_ENABLE()
#define BLUE_BUTTON_GPIO_CLK_DISABLE()
__HAL_RCC_GPIOC_CLK_DISABLE()
```

Then we need to pass that data over to the library calling the HAL_GPIO_Init function. The mode setting tells the system what type of input is expected.

```
void Blue_PB_Init()
{
  GPIO_InitTypeDef gpio_init_structure;

  /* Enable the BUTTON clock */
  BLUE_BUTTON_GPIO_CLK_ENABLE();

  /* Configure Button pin as input */
  gpio_init_structure.Pin = BLUE_BUTTON_PIN;
  gpio_init_structure.Mode = GPIO_MODE_INPUT;
  gpio_init_structure.Pull = GPIO_PULLUP;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(BLUE_BUTTON_GPIO_PORT, &gpio_init_structure);
}
```

The buttons can be accessed using the same library functions as were used for the LEDs, but obviously the buttons are read and the LEDs written to. So the function to read the pin/button is:

```
HAL_GPIO_ReadPin(BLUE_BUTTON_GPIO_PORT, BLUE_BUTTON_PIN);
```

This will return the state of pin 13 – GPIO_PIN_SET (1) means pressed GPIO_PIN_RESET (0) means released.

In the Src directory you will find a button.c – this can be used but you will need to make a backup of your existing main.c – *mv main.c oldmain.c* and then rename button.c to main.c *mv button.c main.c*.

Exercises.

Pass exercise 1. Write a simple program that reads the **B2** button and turns on and off the green LED each time it is pressed.

```
greenledval = 1 - greenledval;
```

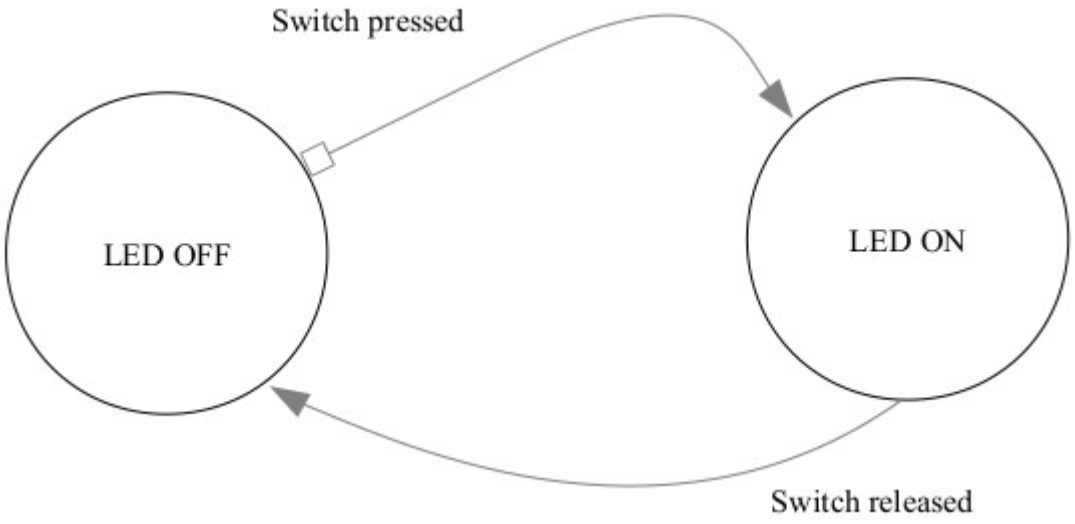
Experiment with these in your code.

Pass exercise 2. Now write a program which toggles between the 2 LEDs when a button is pressed.

De-bouncing switch input

You will notice that your switch/LED program works some what erratically. Sometimes it will miss switch presses other times it will respond well. This is due to 2 things. Firstly the CPU, even on a slowish board such as this, is running at a fast rate and our fingers are slow, so this mismatch can produce odd results. Secondly, the CPU is working in a digital world where things are either 1 or 0, the switch is a mechanical device which has physical properties which are hard to represent digitally. Finally the switches will degenerate as they get older and battered by more and more frustrated students.

Our code so far doesn't take this into account. Represented as a finite state machine it is as follows:



This translates into the following code:

```
while (1)
      {
 but_state=Blue_PB_GetState();
 if (but_state == GPIO_PIN_RESET)
   LED2_On();
 else
   LED2_Off();
 //HAL_Delay(1000);  //delay for 1000 milliseconds - namely 1 second
      }
```

As we have seen this code doesn't really work all the time. What we need to do is to de-bounce the input, this means that we need to write in an extra stage which filters the input to check to see if we have confidence that it is really a key press/release. This will require an extra state both for the key pressed and the key released using a variable that will express our confidence that the trigger event (key press/release) has really happened. The size of this variable will depend upon:

a) the reliability of the switches (pretty good in our case) and
b) how reliable we wish the code to be.

In general it is best to try various values to see their effectiveness. You might also be surprised how they can differ from board to board.

Credit exercise. Design a finite state machine and code to de-bounce the switch input and demonstrate it to your lab tutor.