Carleton UNIVERSITY

Department of Electronics

ELEC 4601: Laboratory 3

Interrupts and Low-Power Features in an Introduction to Arm Microprocessors

Introduction

In this lab, you will learn how to implement a low-level interrupt handler in an Arm-based microprocessor. Arm is a leading supplier of microprocessor technology, licensing reduced instruction set computing (RISC) architectures to companies that design and manufacture their own products (e.g., Qualcomm, Apple, STMicroelectronics). With (generally) a lower transistor count than x86 microprocessors, the Arm architecture is ideal for low-power applications. In this lab, we will use the state-of-the-art Keil µVision integrated development environment (IDE) to program the Arm microprocessor on an STMicroelectronics development board. **Interrupt requests from a joystick on an mbed application shield will be handled by our code to change the colors of an RGB LED—also on the shield.**

Hardware & Software

There are three tools used in this lab:

1. STM32 Nucleo F401RE development board (Arm Cortex-M4)
2. Arm mbed application shield
3. Keil µVision IDE

STM32 Nucleo F401RE development board

The development board used in this lab is part of the STM32 family of microcontrollers by STMicroelectronics. It is useful for building prototypes and learning new concepts related to 32-bit Arm-based microprocessors. The F401RE features an Arm Cortex-M4 32-bit RISC CPU (+ 96 kB of SRAM, 512 kB of flash memory, a debugging interface, and various peripherals). The pinout (header) of this board, shown in Fig. 1, allows the programmer to access each pin of the microcontroller individually, or through an additional development shield (mbed).

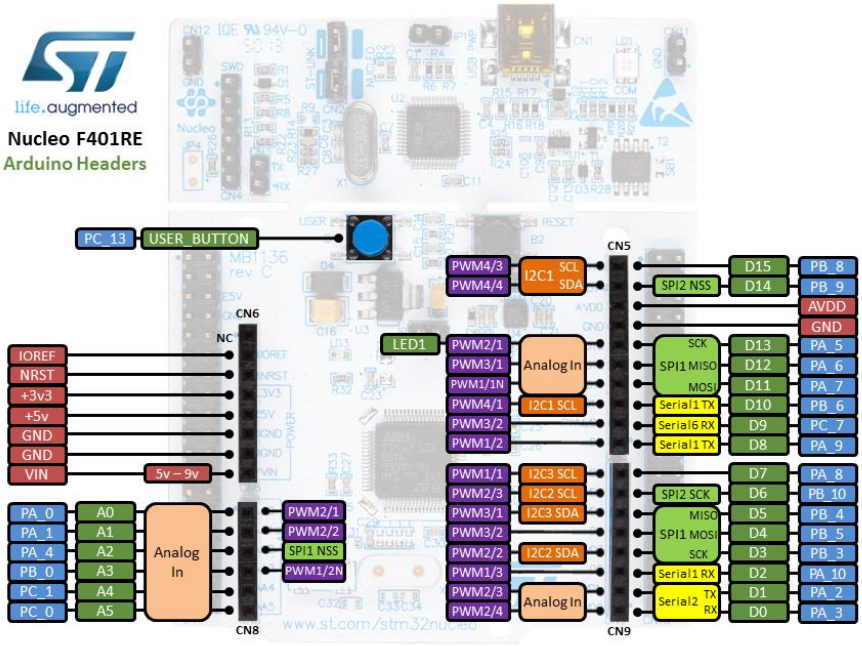

Figure 1 – Nucleo F401RE pin assignments

Arm mbed application shield

The mbed application shield shown in Fig. 2 connects to the Arduino-style header pins of the F401RE to add eight useful I/O tools to its functionality. This lab uses the **5-way joystick and the RGB LED**. The pin-mapping you will need to know when programming these tools is shown in Tab. 1.

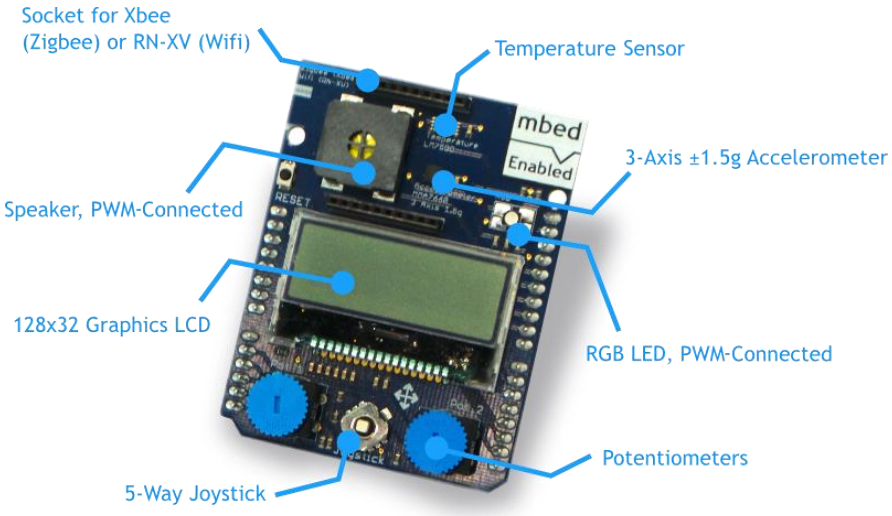

Figure 2 – Arm mbed application shield (labeled)

Table 1 – Pin mapping of the joystick and the RGB LED

Pin	mbed API
JOY_UP	PA_4
JOY_DOWN	PB_0
JOY_RIGHT	PC_0
JOY_CENTER	PB_5
JOY_LEFT	PC_1
RED_LED	PB_4
GREEN_LED	PC_7
BLUE_LED	PA_9

Keil µVision IDE

The µVision IDE combines project management, a run-time environment, build facilities, source code editing, and program debugging. The integrated debugging tool allows you to test, verify, and optimize the application code. As such, µVision is the only software tool you will need to program the STM32 microcontroller; it will be used throughout the remaining three labs of this course. We strongly *suggest* you take the time to become familiar with the layout, as well as some basic C code, if you are not already.

Procedure

In this lab, we will program the 5-way joystick on the mbed application shield to work as an interrupt source on the F401RE microcontroller. We will also program the RGB LED on the mbed shield to toggle in response to an interrupt request in the following manner:

- Left press → Toggle red light
- Right press → Toggle green light
- Up press → Toggle blue light
- Center press → Toggle white light (Red + Green + Blue)

A code template has been provided for you on **cuLearn**, along with useful PowerPoint notes on interrupts, low-power features, and digital inputs and outputs as well as the Reference Manual of STM32F401. You will need to finish the project, test it, and perform an analysis (more on this later) before checking out with the TA. The following steps will guide you along the process:

1. Download and open the project "interrupt_low_level.uvprojx" in **Keil µVision**

2. In the **Project** window, expand the folders until you see the following source files:
   - main.c *(The main program that runs)*
   - leds.c *(Configures the mbed RGB LED. Called inside **main.c**)*
   - switches.c *(Configures the mbed 5-way joystick. Called inside **main.c**)*
   - interrupts.c *(Configures interrupts on the F401RE. Called inside **main.c**)*

3. Familiarize yourself with the C code included in these source files. Read the brief comments provided in the code. You can always get more information about any unfamiliar variable, function, macro, library, etc. by right clicking on its name and selecting the *"Go To Definition of" option.* For example, find the following line of code in the ***init_RGB*** function in the ***leds.c*** source file:

   GPIOB->MODER &= ~MODER(RED_LED);

   The purpose of this line of code is to configure the microcontroller's pin that is connected to the Red LED as output. Here, ***MODER(x)*** is a macro (A macro is piece of code which has a name and wherever this name is used in the code, it is replaced by it value). Now, go to the definition of this macro. You will be directed to ***utils.h*** header file in which that macro is defined as: *#define MODER(x) (0x03 << x*2)*. This macro has a simple task to left-shift the hex value "0x03" by twice the value that the macro receives as its operand. In the sample line of code above, ***RED_LED*** has been defined as the pin number of ***PB_4*** in the microcontroller. ***GPIOB->MODER*** gets access to the ***MODER*** element in the ***GPIOB*** structure. At this point you may need to refer to the provided reference manual (Section 8.4) to read more about Port Mode Register (***GPIOx_MODER***) as well as other GPIO registers that you will need to complete the codes.

4. In the following order, finish the empty methods in **leds.c**:
   i. toggle_r
   ii. toggle_g
   iii. toggle_b
   iv. toggle_all
   v. The last bit of init_RGB (under the comment //Set outputs high)

   The first three methods in this list only require one line of C code each; however, an understanding of how the microcontroller interfaces with the mbed application shield, and how the individual pins and registers are accessed in code, is required. The most important thing to know is that the shield connects

to the microcontroller's general-purpose input/output (**GPIO**) pins. These **GPIO** pins MUST be configured (already mostly done for **leds.c**) and set according to the program requirements. For this part you may need to refer to the reference manual (Port Output Data Register (***GPIOx_ODR***) section) to complete the code.

5. Complete **switches.c** to configure the joystick pins as inputs (under the comment //Set pins to input mode). Remember that ***GPIOx_MODER*** register is used to configure the direction of the pins.

6. Finish configuring the 5-way joystick as an interrupt request line in **interrupts.c**:
   i. Under the comment //Set priority
   ii. Under the comment //Clear pending interrupts
   iii. Under the comment //Enable interrupts

   In the ARM microcontroller you are using in this lab, the i^{th} pin (i= 0, …, 15) of all ports (A-E, H) are connected to the i^{th} line of the external interrupt. However, there are only 7 interrupt handlers for GPIO pins in this microcontroller. The table bellow shows the interrupts and how they are mapped to handlers.

Table 2 – STM32F401 interrupts handlers mapping.

IRQ	Handler Name	Description
EXTI0_IRQn	EXTI0_IRQHandler	Handler for pins connected to line 0
EXTI1_IRQn	EXTI1_IRQHandler	Handler for pins connected to line 1
EXTI2_IRQn	EXTI2_IRQHandler	Handler for pins connected to line 2
EXTI3_IRQn	EXTI3_IRQHandler	Handler for pins connected to line 3
EXTI4_IRQn	EXTI4_IRQHandler	Handler for pins connected to line 4
EXTI9_5_IRQn	EXTI9_5_IRQHandler	Handler for pins connected to lines 5 to 9
EXTI15_10_IRQn	EXTI15_10_IRQHandler	Handler for pins connected to lines 10 to 15

   Note: to set the priority of an interrupt on the Nested Vector Interrupt Controller (NVIC), the command is *NVIC_SetPriority(IRQ, priority number).* For this lab, set the priority to 0. You also need the following commands to clear pending interrupts and enable them: *NVIC_ClearPendingIRQ(IRQ)* and *NVIC_EnableIRQ(IRQ)*.

7. There are four interrupt handlers at the top of **main.c**, labeled "EXTIx_IRQHandler," where "x" refers to the interrupt line of the **GPIO**. EXTI0_IRQHandler has already been completed for you. You will need to fill in the code for the other three handlers. Like the previous lab, the handler (or ISR) simply sets a flag that will be updated back in the main loop

8. Build your code by clicking **Project → Build Target (F7)**

9. Load the built code onto the microcontroller by clicking **Flash → Download (F8)**

10. Check if your code works. If it does, demo it to the TAs

11. Open the debugger by clicking **Debug → Start/Stop Debug Session (Ctrl+F5)**

12. Find the ISR number for each of the four interrupt handlers in the program. The debugger has all the step and breakpoint tools you will need for this, and all the important registers and flags are shown on the left side of the window. Think about what you will need to do to stop the program in each of the

interrupt handlers so that you can gather the important information. Screenshots from registers list and disassembly codes are recommended for the future reference.

13. Track the main stack pointer (MSP) throughout the interrupt handlers (pick two) and the main program. Expand the disassembly code to give you a better idea on when/why the MSP changes—you will need to understand this for the lab report. Screenshots are recommended.

14. Track when the program is in thread mode or handler mode. Screenshots are recommended.

15. Investigate the functionality of *__wfi()* function in the main.c

Note: Before end of the lab make sure you

- Demo your work to TAs
- Have four ISR numbers
- Have MSP values for the main part of the program and two of the interrupt handlers
- Know when the program is in the thread mode or the handler mode
- Know the purpose of *__wfi()* function
- Have a copy of the disassembly code of *__wfi()* and *toggle_g* functions

Lab report

Please complete and submit the Google Form posted on cuLearn. The deadline is one week after your lab session.