ELECTRONIC SYSTEMS ENGINEERING **ENEL 351**
FACULTY OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF REGINA

Lab 2

GPIO on the STM32F10X

Purpose:

This lab will expand the practice of general purpose I/O (GPIO) on the STM32F10X microcontroller. In addition, we will look at how to configure the clock source for the microcontroller.

Equipment/Software/References:

STM32F10X Development Board
USB Cable
PC with Keil uVision5
STM32F1XX Reference Manual
STM32F1XX Data Sheet
Development Board Schematic
ENEL 383/384 parts kit, if applicable

Background:

General Purpose Input/Output (GPIO) refers to the use of logic level pins on the microcontroller device to connect to user input and output devices. It is often referred to as parallel I/O since multiple inputs or outputs appear in common registers inside the device. Reading a group of switches may be as simple as reading the value contained in one device register.

In this lab, we will connect external switches and LEDs to the development board as well as use the onboard switch and green LED. We will also revisit the configuration of the clock generation system.

IO pins are grouped together as IO PORTS. On the STM32F10X, the ports are generally 16 bits wide. Each port connects to a PERIPHERAL REGISTER located in the memory map of the device. Voltage levels that appear at the pins correspond to bit values in the registers. The STM32F10X is designed to operate at 3.3V, this means that the highest output voltage will be approximately 3.3V and the lowest will be 0V. Inputs must also fall within this range. In order to accommodate connections to legacy or existing 5V devices, a number of pins on the STM32F10X are also able to accept inputs up to 5V. As a designer, you must ensure that the allowable input voltage on any pin is never exceeded

The Clock Generation system provides the stable oscillator signals necessary for the micro controller to function. The STM32F10X is versatile in that it can use an internal oscillator, or an external oscillator, or the output of a frequency multiplier circuit, the PLL. The STM32F103 can operate at a maximum clock 72 Mhz obtained from the PLL.

Procedure:

<u>Connecting External Components</u>

1) Construct the circuits as shown in the breadboard schematic PDF. Connect the LEDS to PA7 through PA10 on your NUCLEO board. Connect the switches to to any 4 available inputs on your Nucleo board.

<u>Finding the registers:</u>

1) Refer to the STM32F10X Reference Manual and locate the Device Memory Map. Find the memory locations that correspond to the boundary locations for GPIO Port A, GPIO Port B, GPIO Port C, and the Reset and Clock control, RCC. These addresses represent the beginning of the memory area in which the relevant device registers will appear.

2) Locate the complete register maps for each of the peripherals listed in 1). The register maps will tell you which memory addresses are assigned to specific configuration, status, and data registers for the peripheral.

<u>Modify Lab 1:</u>

1) Modify your program from Lab 1 such that the LED will flash approximately once every second when the USER switch is pressed. This will require you to use the delay() function provided in the library files.

2) Test and troubleshoot until you have it working.

<u>Write some code to interact with your other components:</u>

1) Read the “Things you should know” section of the lab for some background about how the libraries for the STM32F100 work to create structures in our programming environment.

2) Using the library files from lab 1 as a template, create two (2) new sets of library files for your own use. Each library will have consist of a header file (.h) and a source file (.c). The first library will contain the functions, function prototypes, and definitions to handle system clock setup and any timing functions you may need in the future. The second library will contain everything you need to initialize and interact with all required GPIO pins connected to the switches and LEDs. Choose library names that make it easy to identify the types of functions that you will find in the libraries.

3) Create a new main.c file for this lab. In the initialization routines, set the microcontroller clock to 24 MHz.

4) In the main body of your program, create a loop will use the state of the four push-buttons to control the operation of the four LEDs. You could implement different display patterns on the LEDs, change the rate at which you flash them, or use specific LEDs to indicate the state of individual switches.

5) Download and debug your program.

Deliverables:

1) Create a short unlisted YOUTUBE video which displays the operation of your program(s). Submit the YOUTUBE link via the Lab 2 Video Submission assignment link in URCourses.

2) Store your libraries and source file on a network drive. You may be asked to explain or modify their operation during the next lab.

Things you should know:

Every peripheral system in the STM32F10X has a clock source which must be enabled for the peripheral to operate. This includes the GPIO ports

Each pin on a GPIO port can operate in either Input Mode or Output Mode. Within each Mode there are different configurations in which each pin can operate. We use the registers GPIOx_CRL and CPIOx_CRH to set the Mode and Configuration for each pin on a port. By default, pins are set as Floating Inputs.

The Keil library file **stm32f10x.h** defines names, pointers, and structures for all the peripheral registers in your STM32F100. If we include this header file in our project, we can use this information to aid us in writing our programs. However, we need to go back through a number of other header files to decode some of the things that are happening.

From **core_cm3.h**

```
#define     __IO    volatile           /*!< Defines 'read / write' permissions            */
```

From **stdint.h**:

```
typedef unsigned          int uint32_t;
```

From **stm32f10x.h**:

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;

#define PERIPH_BASE           ((uint32_t)0x40000000) /*!< Peripheral base address in the alias region */
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
#define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
#define GPIOA                 ((GPIO_TypeDef *) GPIOA_BASE)
```

So…..GPIOA is a pointer to a structure and CRH is a member of that structure which means that GPIOA->CRH is our pointer to a volatile unsigned int (a 32 bit number) at location 0x40010804.

Now we can perform operations on that 32 number such as assigning it a value directly:

```
GPIOA->CRH = 0x44433334;
```

or setting and clearing individual bits using logical operations

```
GPIOA->CRH |= GPIO_CRH_MODE12 | GPIO_CRH_MODE11 ;
GPIOA->CRH &= ~GPIO_CRH_CNF12 & ~GPIO_CRH_CNF11 ;
```

We can also use this information to read the value in a register into a variable:

```
uint32_t        porta_data;

porta_data = (GPIOA->IDR);
```

or we can use the value in the register along with a bitmask to control program flow;

```
if ((GPIOA->IDR & GPIO_IDR_IDR0) == 0)

        {}
else
        {}
```

And let's not forget about the bit shift operators >> and <<. These operators will shift a number to the right >> or left << by the specified number of bit positions.

```
/*
* Name:         read_DIP
*Returns:       value on 4 position dip switch connected to PA6,PA7,PC10,PC11
*/

uint16_t read_DIP(void)
{
        uint16_t sw_val;

// Note how the >> operator helps us create a four bit value in the 4 least significant bit positions of
// our return value. This four bit value indicates the state of all 4 switches

        sw_val = (((GPIOA->IDR &( GPIO_IDR_IDR6 | GPIO_IDR_IDR7))>>6) \
                | ((GPIOC->IDR &( GPIO_IDR_IDR10 | GPIO_IDR_IDR11))>>8)) \
                & 0x000F;

        return (sw_val);

}
```