Lab 2

Using the Push Buttons

In this lab, we will learn using the push buttons of the LaunchPad board via a basic technique called polling.

2.1 Reading the Push Buttons

The LaunchPad board is equipped with three push buttons, listed below:

```
S1: Button connected to Port 1.1
S2: Button connected to Port 1.2
S3: Button connected to the reset (RST) pin
```

The buttons S1 and S2 are used for general-purpose tasks and we'll use them in this lab. The button S3 is connected to the reset pin and holds the chip in reset state for as long as it's pushed. The reset state is a state in which the chip doesn't respond to any event. The reset pin can be re-purposed as an interrupt pin and, therefore, can be used as a general-purpose pin to some extent.

The buttons S1 and S2 are wired in the active low configuration. That is, when they're pushed, they

read as zero. Confirm this by looking at the schematics in the LaunchPad User's Guide `(slau627a)` on pages 29-34 and take a screenshot of the schematics portion showing the buttons.

Port Configuration

The I/O ports are configured using the port configuration registers. These are presented in the FR6xx Family User's Guide `(slau367o)` in the chapter "Digital I/O" on page 363. We summarize the information that relates to this lab below in Table 2.1

Table 2.1: Configuration Registers for Port 1

P1DIR	Sets pin direction (0: input) (1: output)
P1IN	Reads input pins (0: input is low) (1: input is high)
P1REN	Enables built-in resistors (can be set as pull-up or pull-down) (0: disable resistor) (1: enable resistor)
P1OUT	Writes to output (for output pins) $\rightarrow$ (0: write low) (1: write high) Configures built-in resistors (for input pins) $\rightarrow$ (0: pull-down) (1: pull-up)

A port (e.g. Port 1) has 8 bits and, accordingly, each of the variables shown in the table is 8-bit. We have already used P1DIR, P1IN and P1OUT in the previous lab.

The configuration variable P1REN allows enabling a built-in resistor (inside the chip) for each I/O pin. The built-in resistor is optionally used only when the I/O pin is configured as input. We use P1REN to enable/disable the internal resistor. Furthermore, the resistor can be configured as pull-up (to Vcc) or pull-down (to ground). This is done via P1OUT. Note that when the I/O pin is configured as input, P1OUT is not used for data. Therefore, it's used to configure the resistor as pull-up or pull-down.

In this lab, the I/O pins where the buttons are connected will be configured as input, the built-in resistors will be enabled, and they will be set to pull-up. We can justify this by looking at the buttons' schematics. The button's pin is connected to ground when the button is pushed. Therefore, we'll have the pin pulled-up to Vcc so it reads high when the button is released.

Masking Operation

To read a button's status, we need to inspect the button's corresponding bit inside P1IN. Below are the two masking AND operations that allow reading a bit inside a variable. In this example, we're reading BIT3 in the data.

```
  Data: ---- 0---                    Data: ---- 1---
  BIT3: 0000 1000   AND              BIT3: 0000 1000   AND
Result: 0000 0000                  Result: 0000 1000
```

The masking operation ANDs the data with the mask BIT3. On the left side, the bit is 0 and the result of the AND operation is zero. On the right side, the bit is 1 and the result of the AND operations is nonzero. **Please note that the result is not equal to 1, since the bit 1 is not on the rightmost position.** For the case on the right side, we can check if the result is nonzero, or if the result is equal to BIT3. Accordingly, the piece of code below shows how we check the bit's value.

```
// Check if bit 3 is zero
if( (Data & BIT3) == 0 ) ...

// Check if bit 3 is one (two ways)
if( (Data & BIT3) != 0 ) ...
if( (Data & BIT3) == BIT3 ) ...
```

We note that if we check two bits simultaneously in the masking operation, a nonzero result means one of three cases: 01, 10, or 11. However, when we test one bit in the masking operation (like the example above), a non-zero result means the bit is 1.

Turning on the LED with the Button

Let's write a code that turns on the red LED when button S1 is pushed. As long as the button is held down, the red LED should remain on. When the button is released, the red LED should turn off. Below is the code, fill the missing parts. Throughout the code, we should use the symbolic constant masks BIT0, BIT1, etc or redefined names e.g. redLED, rather than hex values so our code is easily readable.

```
// Turning on the red LED while button S1 is pushed

#include <msp430fr6989.h>
#define redLED BIT0             // Red LED at P1.0
#define greenLED BIT7           // Green LED at P9.7
#define BUT1 BIT1               // Button S1 at P1.1

void main(void) {
  WDTCTL = WDTPW | WDTHOLD;     // Stop the Watchdog timer
  PM5CTL0 &= ~LOCKLPM5;         // Enable the GPIO pins

  // Configure and initialize LEDs
  P1DIR |= redLED;              // Direct pin as output
  P9DIR |= greenLED;            // Direct pin as output
  P1OUT &= ~redLED;             // Turn LED Off
  P9OUT &= ~greenLED;           // Turn LED Off

  // Configure buttons
  P1DIR &= ~BUT1;               // Direct pin as input
```

```
  P1REN ...                     // Enable built-in resistor
  P1OUT ...                     // Set resistor as pull-up

  // Polling the button in an infinite loop
  for(;;) {
      // Rewrite the pseudocode below into C code
      if ( button S1 is pushed )
          Turn red LED on
          else Turn red LED off
  }

}
```

This code polls the button infinitely and, therefore, keeps the CPU locked to this operation. A downside is that the CPU can't do another task. Secondly, if the device is battery-operated, polling the button infinitely could drain the battery. A more advanced approach is to setup the button via interrupt and to put the microcontroller in low-power mode while waiting for the button push. We'll explore this approach in a later lab.

For this part, perform the following and include the answers in your report.

- Include a screenshot of the schematics portion showing the buttons' wiring. Explain why the buttons are active low and why the pull-up resistor is used.
- Write the missing parts of the code and demo it to the TA
- Submit the code in your report

2.2 Two Push Buttons: Occupancy Monitor

Modify the code of the previous part to implement the following occupancy monitor application. There are two parking spots that are equipped with sensing circuits. The sensing circuits output will be emulated with the push buttons. A button that is pushed indicates that the respective parking spot is currently occupied; it's considered occupied as long as the button remains pushed. The two sensing circuits correspond to buttons S1 and S2. When S1 indicates its spot is occupied, the red LED should turn on, and when S2 indicates its spot is occupied, the green LED should turn on. The two sensing operations are independent and are updated concurrently. Therefore, it's possible that the two LEDs are lit simultaneously, which happens when both buttons are pushed.

For this part, perform the following:

- Write the code and demo it to the TA.
- Submit the code in your report.

2.3 Two Push Buttons: Electric Generator Load Control (v1)

In this part, we will implement an industrial application where two machine operators make requests to run their respective machines by using the push buttons. It is imperative that both machines don't run at the same time because this would overload the electric generator powering them, which would lead to a hazardous situation.

The first operator requests to run his machine by pushing S1. If his machine runs, the red LED turns on for as long as the machine is running. Similarly, the second operator requests to run her machine by pushing S2, and, if her machine runs the green LED turns on for as long as the machine is running. In real-life, the operator has a locking button that stays pushed until it is actively released. On our board, keep the button pushed to indicate that a machine is running.

Implement the software based on the following criteria. If no machine is running, an operator can start their machine immediately. Once an operator is running their machine, the other operator's request is ignored as long as the first operator keeps their machine running. In addition, implement the following policy: a request made while the other machine is running will be honored, but only once the other button has been released. That is, if operator 1 is running his machine and operator 2 pushed the button and kept it pushed, the machine of operator 2 runs as soon as operator 1 stops his machine.

For this part, perform the following:

- Write the code and demo it to the TA.
- Stress test the code by checking all possible scenarios.
- Submit the code in your report.

2.4 Two Push Buttons: Electric Generator Load Control (v2)

In this part, we will modify the previous part by implementing policies that enhance the safety of the plant. The main idea here is that the operators should be aware of the current condition and should not make simultaneous requests. Modify the code to implement the following criteria. If a machine is running and the other operator makes a request (pushes the button), the running machine stops immediately. Furthermore, no machine can be started unless both operators have released their buttons. This implies that a request made while the other button was pushed will not be honored (even if the earlier button was released). Both buttons must be released before any request is processed.

Perform the following:

- Write the code and demo it to the TA.
- Stress test the code by checking all possible scenarios.
- Submit the code in your report.

Student Q&A

Submit the answers to the questions in your report.

1. When a pin is configured as input, P1IN is used for data, which leaves P1OUT available for other use? In such case, what is P1OUT used for?
2. A programmer wrote this line of code to check if bit 3 is equal to 1: `if((Data & BIT3)==1)`. Explain why this if-statement is incorrect.
3. Comment on the codes' power-efficiency if the device is battery operated. Is reading the button via polling power efficient?