

## 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

<b>P1DIR</b>	Sets pin direction (0: input) (1: output)
<b>P1IN</b>	Reads input pins (0: input is low) (1: input is high)
<b>P1REN</b>	Enables built-in resistors (can be set as pull-up or pull-down) (0: disable resistor) (1: enable resistor)
<b>P1OUT</b>	Writes to output (for output pins) → (0: write low) (1: write high) Configures built-in resistors (for input pins) → (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---
BIT3: 0000 1000   AND
Result: 0000 0000
```

```
Data: ---- 1---
BIT3: 0000 1000   AND
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.

```
// 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 a button interrupt and to put the microcontroller in low-power mode while waiting for the button push. We'll explore this approach in a future 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 Using Two Push Buttons

Modify the code of the previous part by adding the same functionality on button S2 and the green LED. Therefore, the red LED should light up as long as S1 is held down and the green LED should light up as long as S2 is held down. The code should also work when both buttons are held down, and also when transitioning from one button to the other without lifting both of them.

For this part, perform the following:

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

## 2.3 Electric Generator Load Control (v1)

In this part, implement an industrial application in which two machine operators turn on their machines using the push buttons. Only one machine can operate at a time since the electric generator can't support the machines running simultaneously. It is your software's responsibility to ensure that the machines don't run simultaneously.

This is how the system works. Operator 1 requests to run his machine by holding button S1 down and, if his request is granted, the red LED turns on to indicate that the machine is running. Similarly, Operator 2 requests to run her machine by holding button S2 down and, if her request is granted, the green LED turns on to indicate that the machine is running. The operators hold their buttons down for as long as they wish to run their machines. The software should implement a first-come first-serve policy and an operator can run his machine for as long as he wishes.

Implement the following policy regarding simultaneous requests. If an operator is running his machine, it's okay for the other operator to make a request, but the current machine that is running continues to run. When the current machine stops, the other machine runs if its request is still pending. Basically, this policy means that the operators are allowed to make simultaneous requests and it's the software's responsibility to ensure that the machines don't run at the same time.

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 Electric Generator Load Control (v2)

Modify the code of the previous part by implementing a stricter policy on simultaneous requests. The idea here is that the operators should never make simultaneous requests. If a machine is running and the other operator makes a request, the running machine stops immediately and no machine can turn on until both operators release their requests (i.e. both buttons are released).

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. Evaluate this lab's codes regarding power-efficiency if the device is battery operated. Is reading the button via polling power efficient?