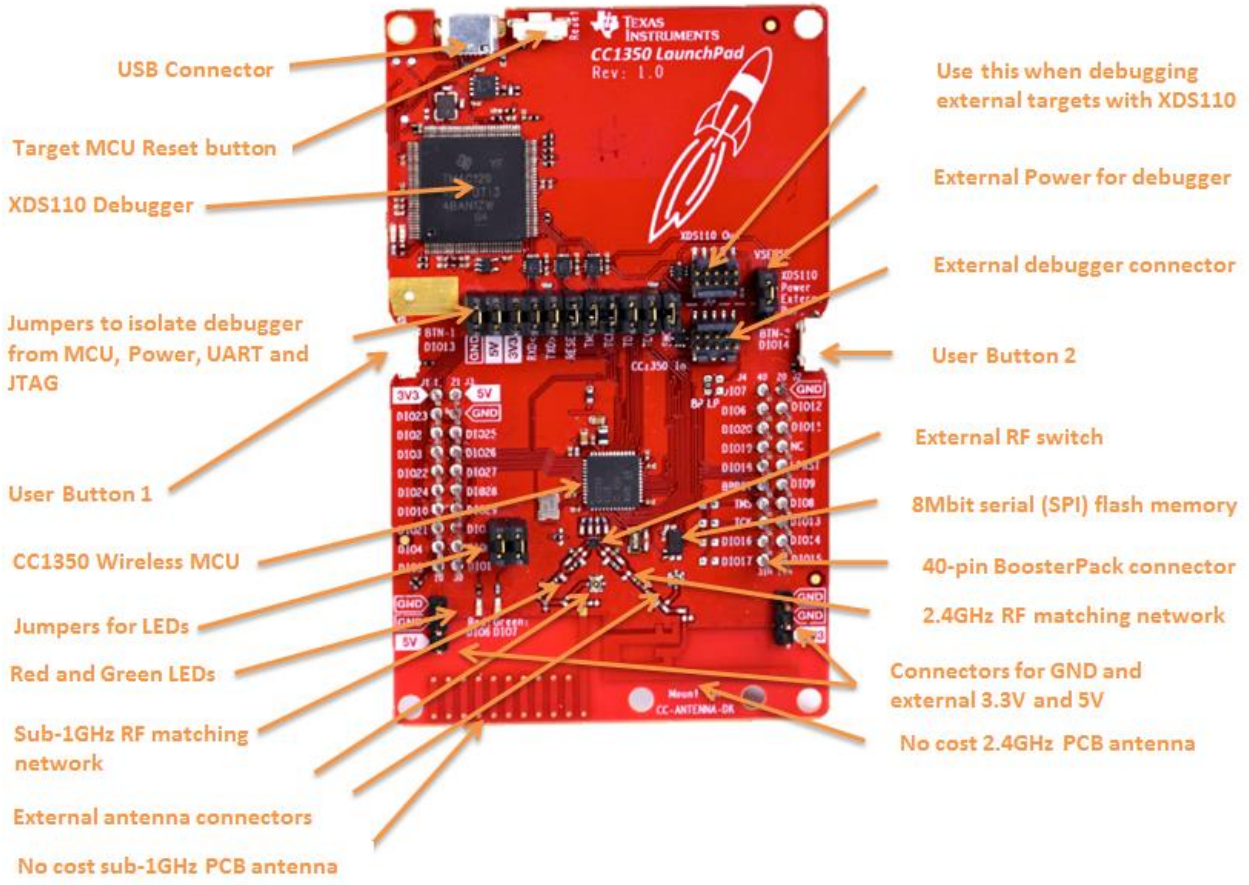
ECEN 361 Lab: Input and Interrupts

Introduction:

One of the key attributes of an embedded system is the ability to accept input. This can come in one of two ways; synchronous and asynchronous. A synchronous input happens at a known time. An asynchronous input comes at any time, we don't know when it might happen. In this lab you'll learn how to accept input, both using polling as well as interrupt driven input.

Hardware:



In this lab you'll use User Button 1 and/or User Button 2 to get input. In addition you'll need the following parts:

1 LED
1 330 Ohm (220 to 470 ohm will do) resistor
1 Switch
1 breadboard

You'll also need access to a Logic Analyzer. There are four LogicPort Logic Analyzers in the lab, or you can purchase your own. They are $12 at amazon. Here is the link:

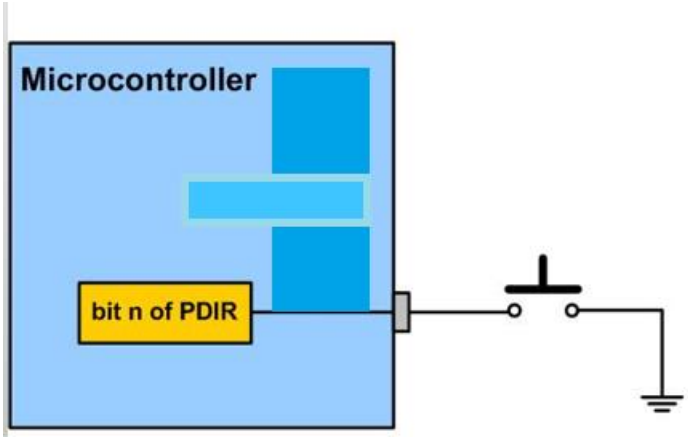
https://www.amazon.com/HiLetgo-Analyzer-Ferrite-Channel-Arduino/dp/B077LSG5P2/ref=sr_1_3?ie=UTF8&qid=1535386495&sr=8-3&keywords=logic+analyzer

See the document **Walk through for Using the LogicPort Logic Analyzer** for help on how to use the Logic Analyzer.

Using Polling to Get Input

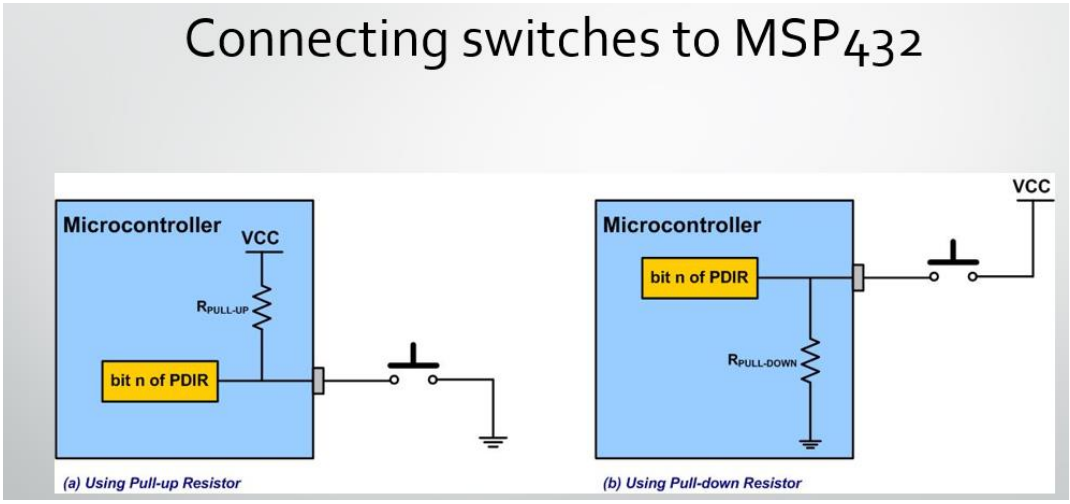
The easiest way to get input is simply to set up the hardware and then simply ask the hardware if an input has occurred. We call this polling, and you can use a loop to accomplish it.

In this section of the lab you'll also learn how to set up the MSP432P401R's hardware using low level commands. So let's get started. First, the hardware. You can just connect the switch to a port on the MSP432. However, there is a problem in that the state of the switch when it is not pressed is a bit sketchy. Here is the circuit:



So the key question is, what is the voltage level the switch "sees" when the switch is open? The answer is, well it depends on the output state of the GPIO line, which is generally defined as floating, which can mean anywhere between 3.3 and 0 volts. What you really want is a way to force the voltage level to a 1 (3.3V) or a 0 (0 V) when the switch is open. We do this normally with a pull up (to 5 volts) or a pull down (to 0 volts) circuit.

The circuits look like this:



The circuit on the left works like this. When the switch is open the there is no current flowing through the $R_{PULL-UP}$ and therefore the voltage at the input pin is VCC. When the switch is closed current flows through $R_{PULL-UP}$ and the voltage at the input pin is 0 volts. The circuit on the right works similarly, however, when the switch is open the voltage is 0, and when it is closed it is forced to VCC.

Fortunately the MSP432P401R already has these in Resistors in the hardware, you'll just need to configure them. So here are the steps to create a polling loop (you'll use one of the on-board LEDs to indicate that the switch has been pressed.)

1) configure P1.1 as simple I/O in P1SEL1:P1SEL0 registers,
2) make P1.1 input pin in P1DIR register for push-button switch S1,
3) configure P1REN register to enable the pull resistor,
4) configure P1OUT register to select the pull-up resistor,
5) configure P2.0 as simple I/O in P2SEL1:P2SEL0 registers,
6) make P2.0 output pin in P2DIR register for red LED,
7) read switch from P1.1,
8) if P1.1(switch) is high, set P2.0 (red LED)
9) else clear P2.0 (red LED)
10) Repeat steps 7 to 9.

And here is the code:

```
int main(void) {
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;  /* stop watchdog
timer  */
    P1->SEL1 &= ~2;         /* configure P1.1 as simple I/O */
    P1->SEL0 &= ~2;
    P1->DIR &= ~2;          /* P1.1 set as input */
    P1->REN |= 2;           /* P1.1 pull resistor enabled */
    P1->OUT |= 2;           /* Pull up/down is selected by P1->OUT */

    P2->SEL1 &= ~1;         /* configure P2.0 as simple I/O */
    P2->SEL0 &= ~1;
    P2->DIR |= 1;           /* P2.0 set as output pin */

    while (1) {
        if (P1->IN & 2)     /* use switch 1 to control red LED */
            P2->OUT &= ~1;  /* turn off P2.0 red LED when SW is not
pressed */
        else
            P2->OUT |= 1;   /* turn on P2.0 red LED when SW is pressed
*/
    }

}
```

This code works well, unfortunately our code will be stuck in this while forever loop looking for input. It can't do anything else. One solution to this problem is to use interrupts to capture this input.

Using Interrupts

As discussed in class, interrupts are a way for the hardware to control software execution. You will decide beforehand what code to run when an interrupt occurs. So let's look at an example. Here is some code from the

gpio_input_interrupt_MSP_EXP432P401R_nortos_ccs

project. This is the gpio_input_interrrupt.c file.

```
/******************************************************************************
 * MSP432 GPIO - Input Interrupt
 *
 * Description: This example demonstrates a very simple use case of the
 * DriverLib GPIO APIs. P1.1 (which has a switch connected to it) is configured
 * as an input with interrupts enabled and P1.0 (which has an LED connected)
 * is configured as an output. When the switch is pressed, the LED output
 * is toggled.
 *
 *                MSP432P401
 *             ------------------
 *         /|\|                  |
 *          | |                  |
 *          --|RST         P1.0  |---> P1.0 LED
 *            |                  |
 *            |            P1.1  |<--Toggle Switch
 *            |                  |
 *            |                  |
 *
 ******************************************************************************/
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>

int main(void)
{
    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();

    /* Configuring P1.0 as output and P1.1 (switch) as input */
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);

    /* Configuring P1.1 as an input and enabling interrupts */
    MAP_GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
    MAP_GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);
    MAP_Interrupt_enableInterrupt(INT_PORT1);

    /* Enabling SRAM Bank Retention */
    MAP_SysCtl_enableSRAMBankRetention(SYSCTL_SRAM_BANK1);

    /* Enabling MASTER interrupts */
```

```
    MAP_Interrupt_enableMaster();

    /* Going to LPM3 */
    while (1)
    {
        MAP_PCM_gotoLPM3();
    }
}

/* GPIO ISR */
void PORT1_IRQHandler(void)
{
    uint32_t status;

    status = MAP_GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, status);

    /* Toggling the output on the LED */
    if(status & GPIO_PIN1)
    {
        MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
    }

}
```

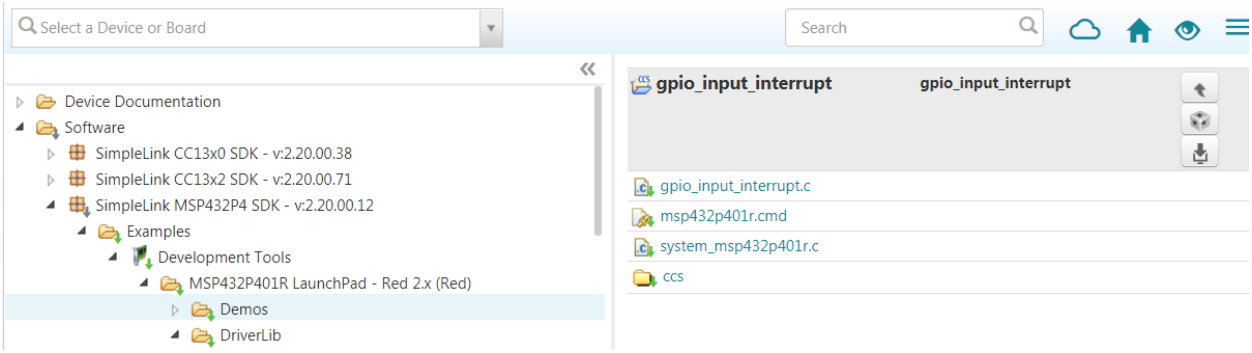
It is important to remember that this code uses a hardware library to abstract a significant set of the actual hardware setup. However, it does configure an interrupt so that it can toggle the LED when it happens.

Lab Results:

Submit the following on the project assignment on i-learn.

1) Create a new project in the Code Composer IDE, an empty project with main.c. Add the code for the simple switch loop with the LED. Build the code, and then download the code. Use the logic analyzer on the LED output signal, and submit a screen shot of the Logic Analyzer capturing a transition based on a button press.

2) Download the gpio_input_interrupt_MSP_EXP432P401R_nortos_ccs example project from here:



Compile and Run the Code. Use the Logic Analyzer to capture a transition based on a button press.

3) Use your parts to create your own switch and LED interface. Here is how to connect the switch:

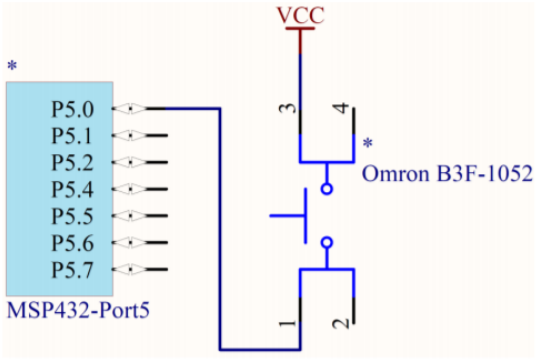


Figure 8.3 . Interface connecting the switch to P5.0 using internal pulldown. (CircuitMaker).

And here is some code to test the switch:

Warning: *Limit the current into and out of port pins to be less than 6 mA. One very bad way to build the switch interface is to place one side of the switch to +3.3V and the other side to ground, causing a 3.3V to ground short whenever the switch is pressed.*

```
Hint: Sample code to interface one switch with
internal pull ups using P5.0

uint8_t sensor;
int Program8_1(void){
  Clock_Init48MHz();  // makes bus clock 48 MHz
  P5->SEL0 &= ~0x01;  // configure P5.0 GPIO
  P5->SEL1 &= ~0x01;
  P5->DIR &= ~0x01;   // make P5.0 in
  P5->REN |= 0x01;    // enable pull resistor on P5.0
  P5->OUT &= ~0x01;   // P5.0 pull-down
  while(1){
    sensor = P5->IN&0x01; // read switch
  }
}
```

And here is how to connect the LED:

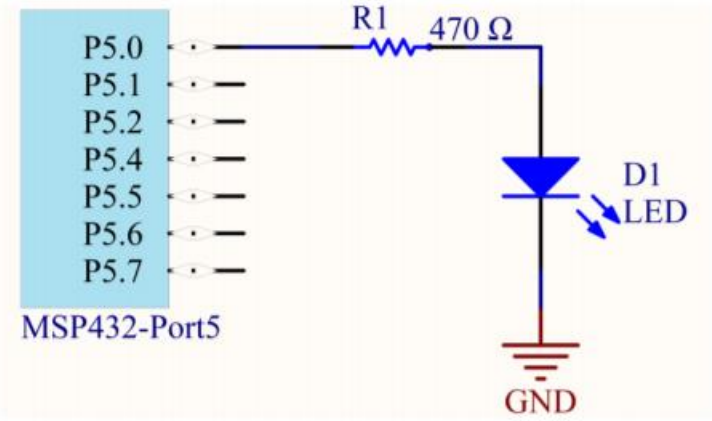


Figure 8.4. Example interface connecting the LED output to P5.4. (CircuitMaker)

And some code to test the LED connection:

```
Hint: Use this program to test the LED interface
void LED_Init(void){
  P5->SEL0 &= ~0x01;  // configure P5.0 GPIO
  P5->SEL1 &= ~0x01;
  P5->DIR |= 0x01;    // make P5.0 output
}
void LED_On(void){
  P5->OUT |= 0x01;  // turn on
}
void LED_Off(void){
  P5->OUT &= ~0x01; // turn off
}
void LED_Toggle(void){
  P5->OUT ^= 0x01;  // change
}
int Program8_3(void){
  Clock_Init48MHz();  // makes bus clock 48 MHz
  LED_Init();         // activate output for LED
  while(1){
    LED_On();
    LED_Off();
  }
```

Use these to create an external version of the internal version you created in step 1) above. Capture a Logic Analyzer trace of the functionality.