General Purpose Input/Output

Overview:

In this first lab we will walk through the basics of writing to digital output pins on the MSP432P4. While we will provide you with the information you will need to know for this lab, further documentation for the MSP432P401 board is available. The following resources may be useful and are listed for your reference. They will be referenced at the appropriate points throughout this lab.

- MSP432P401R pinout diagram: https://components101.com/sites/default/files/component_pin/MSP432-Pinout.png

- MSP432P401R GPIO driver library API settings and specification: http://dev.ti.com/tirex/content/simplelink_msp432_sdk_1_20_00_45/docs/driverlib/msp432p4xx/html/driverlib_html/group__gpio__api.html

Note, for this and all other labs the launchpad's watchdog timer **must** be disabled. Be sure to always include the Example 1 function given below as the first statement in your main() function.

```
1 // Stop watchdog timer
2 WDT_A_hold(WDT_A_BASE);
```

Example 1: Stopping the watchdog timer.

Part 1: Timed Binary Counter

The MSP432P401R launchpad has lots of pins available for general purpose input and output (GPIO) functions. Every physical pin on the board is identified by two numbers: port number and pin number. The common format to represent these port and pin number pairs that you will find in Texas Instrument's documentation is PX.Y or PX_Y for the output identified by port X and pin Y. We will also adopt this notation for this and future labs. Pins will also be identified by the same port/pin number pair in the code we will write. Figure 1, shown below, identifies the pins' names and locations available on the MSP432P401R. Many of the pins' names are also printed on the launchpad itself.

For example, if we want to create a program to toggle LED1. Using Figure 1 we find that LED1 corresponds to P1.0 (aka port 1, pin 0) as seen in the bottom left of the figure.

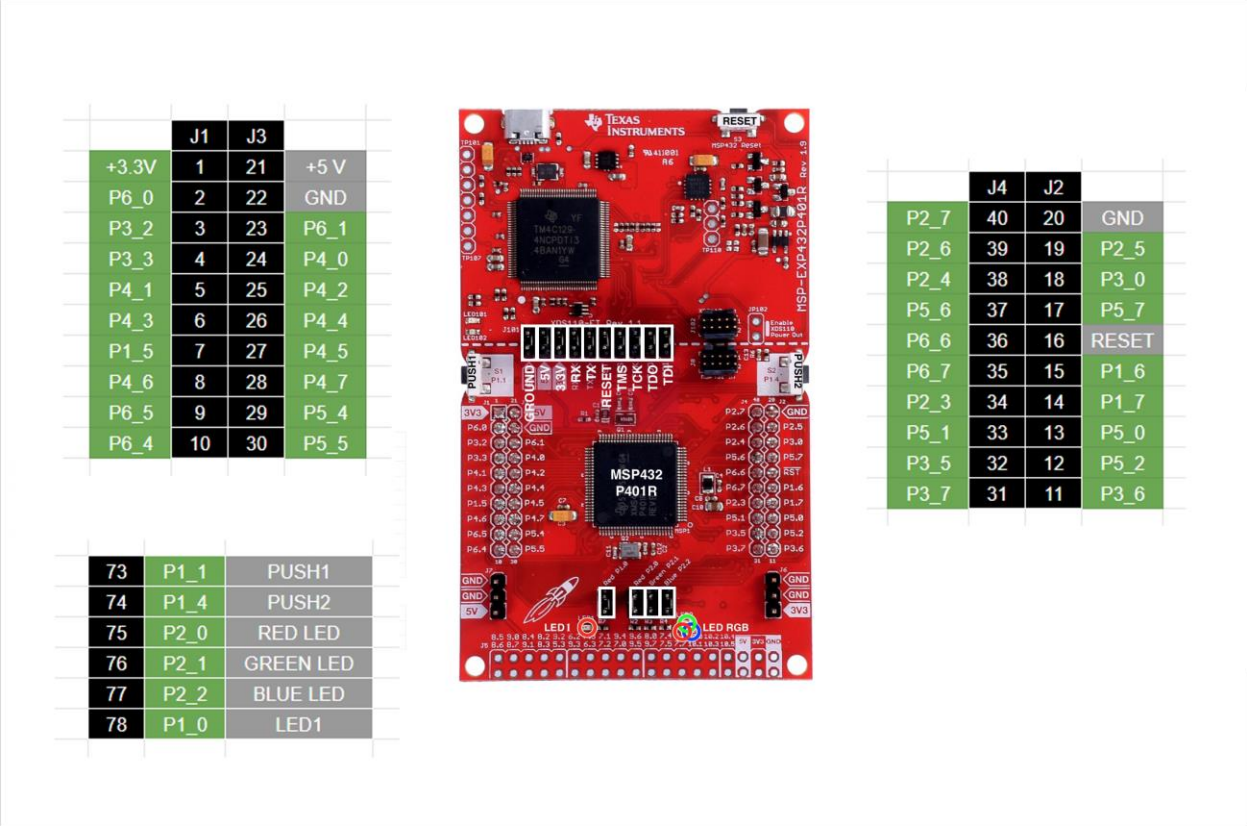

Figure 1: MSP432P401R Launchpad Pinout Diagram [Original Source]

In order to use a pin for digital output two things need to be done first. First, we need to include the correct libraries. Secondly, we must set the pin as an output. Once the setup steps are complete we can write to the digital output pins. There are three functions available to write digital outputs to pins:

1. **GPIO_setOutputHighOnPin()** : This function sets the output of the given port and pins to a logical one, or high voltage. Multiple pins can be set at the same time if they belong to the same port. This is accomplished by performing a bitwise OR on the pin numbers.
2. **GPIO_setOutputLowOnPin()** : This function is identical to the previous one except that it sets the output to a logical zero, or low voltage. Again, multiple pins belonging to the same port can also be specified by performing a bitwise OR on the pin numbers.
3. **GPIO_toggleOutputOnPin()** : This function will flip the specified pin(s) output from a logical zero to a logical one or vice versa. Multiple pins belonging to the same port can also be specified by performing a bitwise OR on the pin numbers.

The full syntax for these functions are presented below in example 2.

```
1   #include <driverlib.h>
2
3   // Configure the pin as an output
4   GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
5
6   // Set the output to LOW on P1.0
7   GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0);
8
9   // Set the output to HIGH on P1.0
10  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);
11
12  // Flip output on P1.0
13  GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
```

Example 2: Configuration options for using an output pin.

Other ports and pins can be specified using similar constants to the ones seen in Example 2 on lines 5, 8, and 11 by replacing the numbers with the correct port and pin numbers. For example, if you wanted to use P2.6 for output instead of P1.0 in the previous example you would use the constants "`GPIO_PORT_P2`" and "`GPIO_PIN6`". The functions and constant definitions are defined and explained in the GPIO driver library API provided by Texas Instruments as mentioned earlier in the overview section. The link is also available in the overview section above.

For this lab we will use the RGB LED to count in binary from 0 to 7 repeatedly. The RGB LED consists of three small LED lights, one red, one green, and one blue. Each of these colored LEDs will represent a bit in the binary number that we will use to count. The lights should count up approximately every second, similar to the blinkled example. The blinkled example accomplished the necessary delay by having a for loop count down to zero from a large number.

To get started, create a new sketch in Energia. After each increment through the loop function, the RGB LED should be updated in reference to table 1. Your code should count from 0 to 7 and then repeat indefinitely. Use the code from example 3 as an initial outline. You will only need to implement your logic in the RGB_output() function. This function is explicitly boxed in example 3 for clarity.

Important Practical Notes

Do not use Copy-and-Paste from a pdf into Energia! This can insert hidden control characters into your code which will cause your compilation to fail for no apparent reason. This can be difficult to debug because the control characters are invisible. You should just type in the code by hand. You can use Copy-and-Paste from another Energia skecth, if desired, but not from a pdf!

Count	LED B (P2.2)	LED G (P2.1)	LED R (P2.0)	Output Color	Example
0	0	0	0	Black	
1	0	0	1	Red	
2	0	1	0	Green	
3	0	1	1	Yellow	
4	1	0	0	Blue	
5	1	0	1	Magenta	
6	1	1	0	Cyan	
7	1	1	1	White	

Table 1. RGB LED binary count sequence and corresponding output colors

```
#include <driverlib.h>

uint8_t count = 0;

void RGB_output(uint8_t count){
  // Function to output to RGB LED based on count value between 0 and 7


}

void setup() {

  // Stop watchdog timer
  WDT_A_hold(WDT_A_BASE);

  GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2);
}

void loop() {
  // put your main code here, to run repeatedly:

  RGB_output(count);

  count++;
  if (count > 7){
    count = 0;
  }

  // Delay
  delay(1000);

}
```

Example 3: Main Program for Lab Exercise

Part 2: Binary Counter Using Switch Input

In this part, you will modify your 3-bit binary counter to increment each time a switch (P1.1) is pressed. You will display the count on the RGB LED using the RGB_output() function developed in part 1.

Design Specifications

Each time SW1 (P1.1) is pressed, your program should increment a count variable and display the corresponding color as given in Table 1. No matter how long SW1 is held, the count should only increment ***once*** for each press and release of SW1. You need to write two short polling loops – one loop to wait until the switch is pressed and another one to wait until the switch is released. A couple of short delay loops should be used to avoid reading SW1 just after it has changed state. This is because mechanical switches often 'bounce' on the switch contact when pressed or released. The delay is to avoid reading the switch input when it could possibly be 'bouncing'. You can adjust the length of the delay as needed if a single press and release of the switch results in more than a single increment of the count. Typically, switch bounce only lasts up to about 50 milliseconds, so the delay should be quite short.

You can use the code shown in Example 4 as a starting point for your program. **Remember, do *not* use Copy-and-Paste from a pdf into Energia!** This can insert hidden control characters into your code which will cause your compilation to fail. This can be difficult to debug because the control characters are invisible. You should type in the code by hand or use your existing code from part 1. You can use Copy-and-Paste from another CCS file, if desired, but not from a pdf!

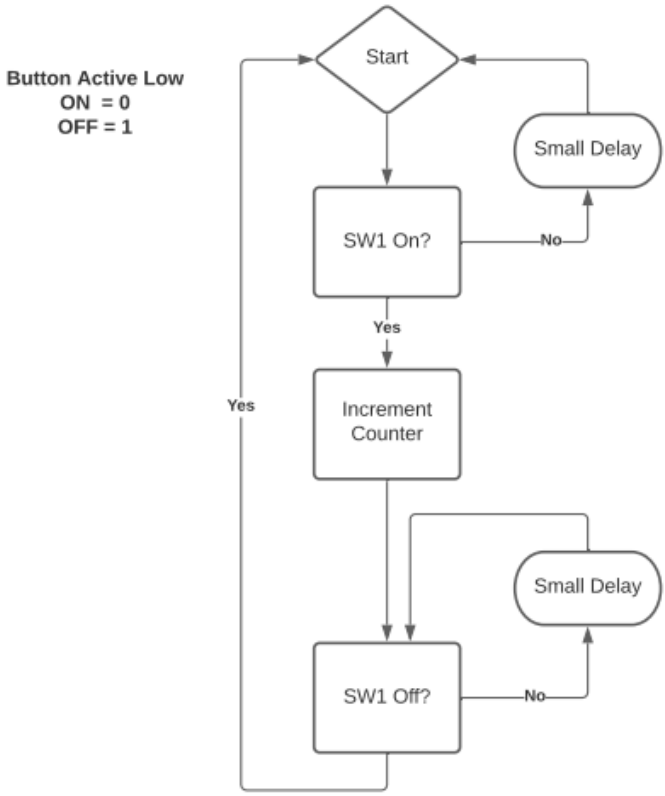

Figure 2: Debounced Switch Logic

```
#include <driverlib.h>

uint8_t count = 0;

void RGB_output(uint8_t count) {
  // Function to output to RGB LED based on count value between 0 and 7

}

void setup () {

    // Stop watchdog timer
    WDT_A_hold(WDT_A_BASE);

    // Set P1.0 to output direction
    GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2);
    GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0 | GPIO_PIN1 | GPIO_PIN2);
    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
}

void loop () {

    /*
     *  Wait here until Sw1 on P1.1 is pressed
     *  This should be a tight polling loop
     *  The processor will execute the loop until SW1 is pressed
     *  Recall that SW1 is active low
     */

    count++;
    if (count > 7){
        count = 0;
    }
    RGB_output(count);

    delay(100);                         // Delay

    /*
     *  Wait here until Sw1 on P1.1 is not pressed
     *  This should be a tight polling loop
     *  The processor will execute the loop until SW1 is not pressed
     */

    delay(100);                         // Delay

}
```

Example 4: Main Program for Part 2

In order to fetch the input value from a GPIO, use the following function (which returns an 8-bit unsigned integer).

```
#include <driverlib.h>

// Fetch GPIO Input
GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1)

```

For more information about this function and other functions in the driver library, refer to the second link in this lab's overview.