Workshop 1: GPIO Fundamentals

Getting Started

For this part of the ELEC0010 module (Embedded Systems and Microcontrollers), we will be using the Direct Register Access (DRA) technique, since it enables us to manipulate control and data registers bit-by-bit, as well as handling interrupts at their most basic level.

In this specific workshop, you will learn how to use the DRA technique to control general purpose input output (GPIO) registers. We will also introduce concepts around clocking and demonstrate how we can configure the MSP432's clock registers to achieve a specific clock frequency.

NOTE: A number of exercises can be found at the end of each task. Completing these exercises will prepare you towards the summative assessments.

Contents

1 Task 1 – GPIO Basics **2**
1.1 Turn an LED on/off 2
1.1.1 EXERCISE 1 3
1.2 Blink an LED 4
1.2.1 EXERCISE 2 4
1.2.2 EXERCISE 3 4

2 Task 2 – Creating your own GPIO Library **5**
2.1 MyGPIOLib: pinMode 5
2.1.1 EXERCISE 4 6
2.2 MyGPIOLib: digitalWrite 6
2.2.1 EXERCISE 5 7
2.2.2 EXERCISE 6 7

3 Task 3 – Reading Inputs **8**
3.1 Read a push button 8
3.1.1 EXERCISE 7 9
3.1.2 EXERCISE 8 9
3.2 MyGPIOLib: inputs 10

4 Task 4 – RGB LED **12**
4.1 Configuring the MSP432's RGB LED 12

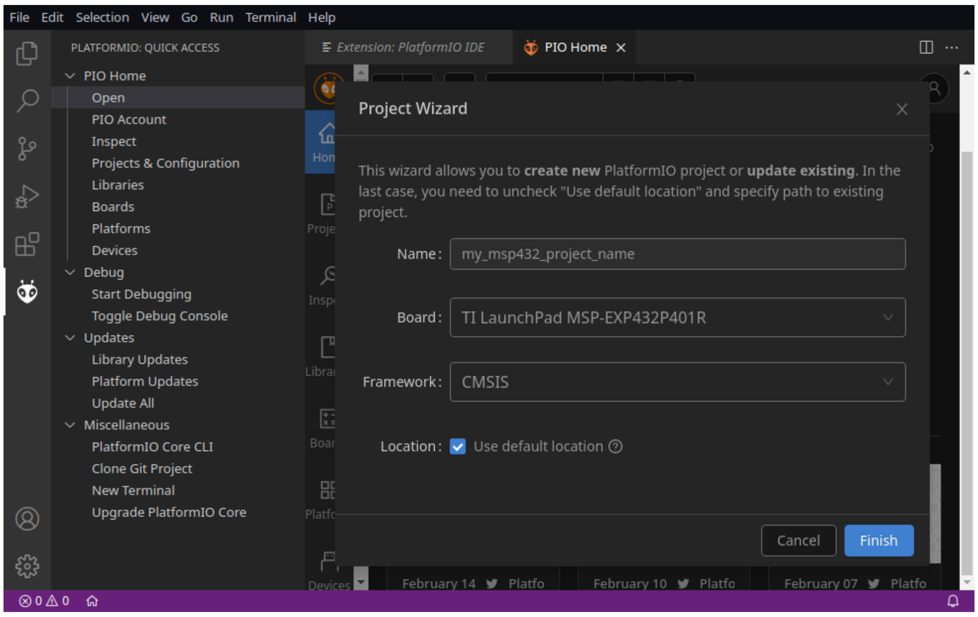
1 Task 1 – GPIO Basics

NOTE: The different programming syntax (for example `WDT_A->CTL` instead of `WDTCTL`) is a characteristic of Arm's Cortex Microcontroller Software Interface Standard (CMSIS), which is supported by the MSP432 because this microcontroller is equipped with an Arm Cortex M4F processor. The MSP432 toolchain with PlatformIO supports both the conventional (i.e. `WDTCTL`) and CMSIS (i.e. `WDT_A->CTL`) programming syntaxes, which can also be used concurrently within the same program. For simplicity, we shall be using the conventional syntax in this workshop.

1.1 Turn an LED on/off

We shall now demonstrate how we can build a program "from scratch". Each task will build on this initial program to generate different GPIO functionalities. **Learning outcome**: Learn how to program your MSP432 to turn an LED on or off.

- Let's start by creating a new PlatformIO project. Note that there is a shortcut with home icon at the left of VSCode bottom bar.
  - Click New Project
  - Name Proejct **Ex1-LEDBeginner**
  - Search for **"432"** and select **TI LaunchPad MSP-EXT432P401R** board
  - Select **CMSIS** framework.



Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name: my_msp432_project_name

Board: TI LaunchPad MSP-EXP432P401R

Framework: CMSIS

Location: ✓ Use default location

Cancel Finish

- Create main.c in src directory and copy the starting code:

```c
#include "msp.h"

void main() {
  // Stop watchdog timer
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;
}
```

The registers of our microcontroller are referenced by name. These register names are found in the header file, "msp.h", which is included in Line 1. This file can be opened by right-clicking on "msp.h" in code and selecting "Go To Definition".

As always, the first thing to do in the main function is to stop the watchdog timer. This is carried out in Line 4.

Once we have included the necessary header files and stopped the watchdog timer, the next step is to start setting up the GPIO pins.

First, we must define whether the pin is an input or output. This is set using the P1DIR register. This is an 8-bit register and each bit corresponds to a particular pin of port 1. For example, the first bit of P1DIR corresponds to P1.0 while the second bit corresponds to P1.1 and so forth. Setting a bit in P1DIR to 1 defines the pin as an output, whereas setting the bit to 0 defines the pin as an input.

- On our MSP432, the red LED1 is connected to P1.0. Hence, we wish to set P1.0 to be an output.
  - Look at your board and the MSP432's pin map to understand this LED-to-pin connection.

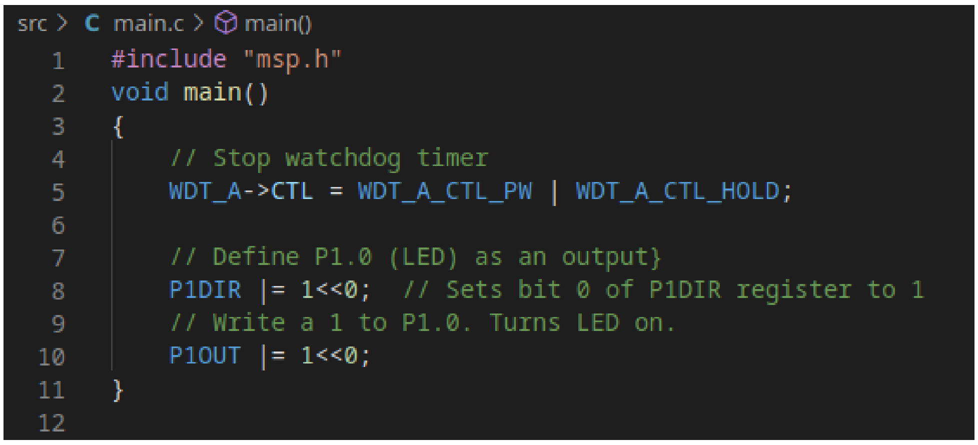
```
// Define P1.0 (LED) as an output}
P1DIR |= 1<<0;  // Sets bit 0 of P1DIR register to 1
```

- The voltage of a pin is defined by the P1OUT register. A value of 0 sets the pin to 0V, whereas a value of 1 sets the pin to Vcc. Here, we wish to turn LED1 on.

```
P1OUT |= 1<<0;  // Write a 1 to P1.0. Turns LED on.
```

NOTE: If you do not understand how the value of 1 is written to bit 0 (for both registers P1DIR and P1OUT), then you need to refer to the lecture notes on bitwise operations.

- Your final code should look like the one in the following screenshot. Compile and upload this code by clicking on →. You should then find that the red LED1 (connected to pin P1.0) will turn on.



```
 1  #include "msp.h"
 2  void main()
 3  {
 4      // Stop watchdog timer
 5      WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;
 6
 7      // Define P1.0 (LED) as an output}
 8      P1DIR |= 1<<0;  // Sets bit 0 of P1DIR register to 1
 9      // Write a 1 to P1.0. Turns LED on.
10      P1OUT |= 1<<0;
11  }
12
```

1.1.1 EXERCISE 1

Modify your program to turn LED1 off.

1.2 Blink an LED

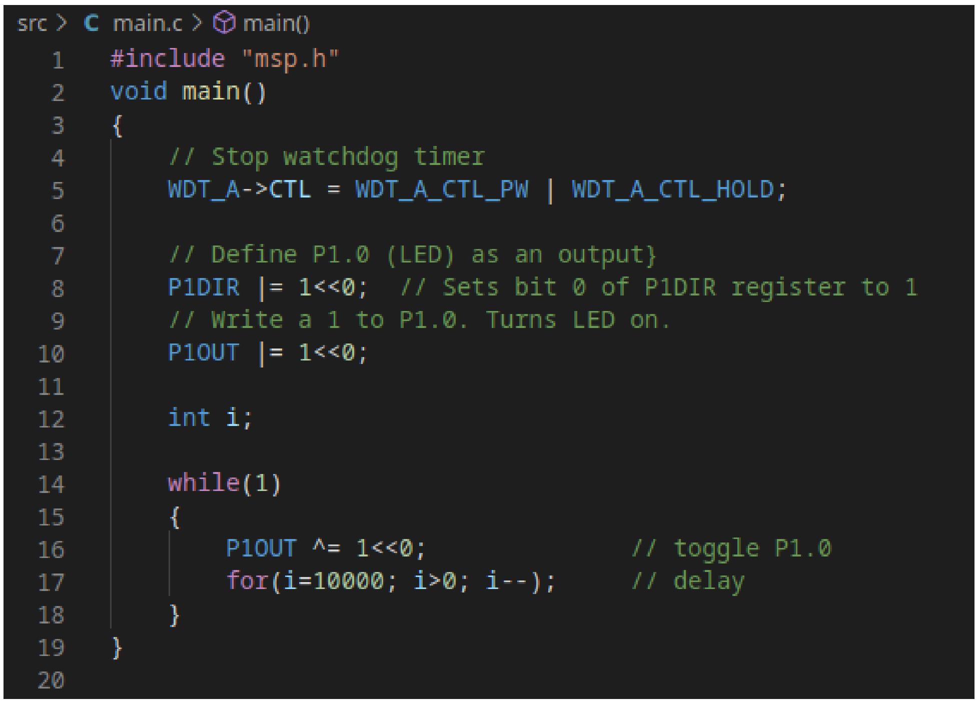
Learning outcome: Learn how to blink the LEDs found on your MSP432.

In your C programming module in Year 1, you learned how to code the classic ‘Hello World’ test. The microcontroller equivalent is blinking an LED, as demonstrated in this section.

- Create a new project in PlatformIO and name it "Ex2-LEDBlinking".
- Copy the code from the previous subtask in order to turn LED1 on.
- We now need to setup an infinite loop that will continuously flash our red LED1. Within the loop we toggle the bit corresponding to P1.0 in the P1OUT register and then generate a delay with a long *for* loop.

```
int i;
while(1)
{
  P1OUT ^= 1<<0;                // toggle P1.0
  for(i=10000; i>0; i--);       // delay
}
```

- Your final code should like the one in the following screenshot. Compile and upload this code by clicking on →. You should then find that the red LED1 (connected to pin P1.0) will blink continuously.



```
#include "msp.h"
void main()
{
    // Stop watchdog timer
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;

    // Define P1.0 (LED) as an output}
    P1DIR |= 1<<0;  // Sets bit 0 of P1DIR register to 1
    // Write a 1 to P1.0. Turns LED on.
    P1OUT |= 1<<0;

    int i;

    while(1)
    {
        P1OUT ^= 1<<0;              // toggle P1.0
        for(i=10000; i>0; i--);     // delay
    }
}
```

1.2.1 EXERCISE 2

Experiment with changing the frequency of the blinking.

1.2.2 EXERCISE 3

Make LED2 blink (you can select any one of the P2.X pins connected to the RGB LED).

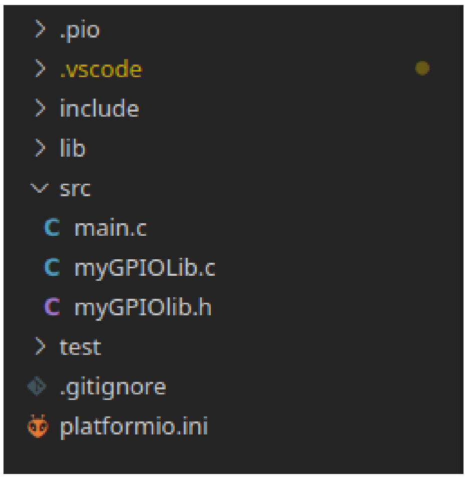
2 Task 2 – Creating your own GPIO Library

In this task, we will create our own GPIO library to mimic the functions used in Energia sketches (this is commonly referred to as the 'Wire' language). We will name the library "myGPIOLib", which will comprise a C file and its corresponding header file.

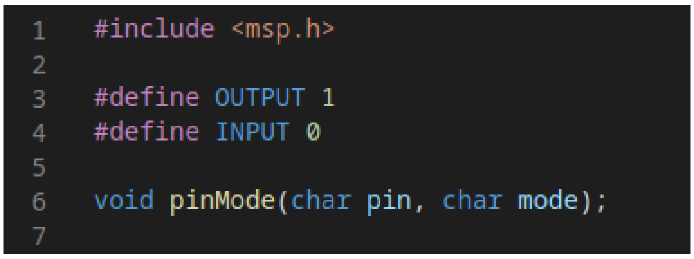
2.1 MyGPIOLib: pinMode

Learning outcome: Learn how to create your own library and configure the pin modes.

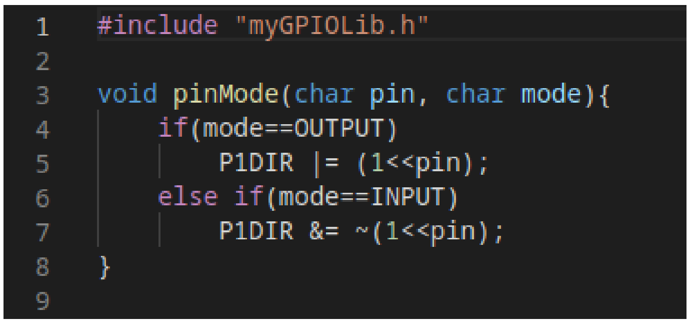
- Create a new project and name it "Ex3-MyGPIOLib-pinMode"
- Right-click on the src directory and select 'New File'. Name the file "myGPIOLib.h"
- Repeat the previous step and create another file called "myGPIOLib.c". Your project file structure should look like the one in the following screenshot.


```
> .pio
> .vscode
> include
> lib
v src
   main.c
   myGPIOLib.c
   myGPIOlib.h
> test
  .gitignore
  platformio.ini
```

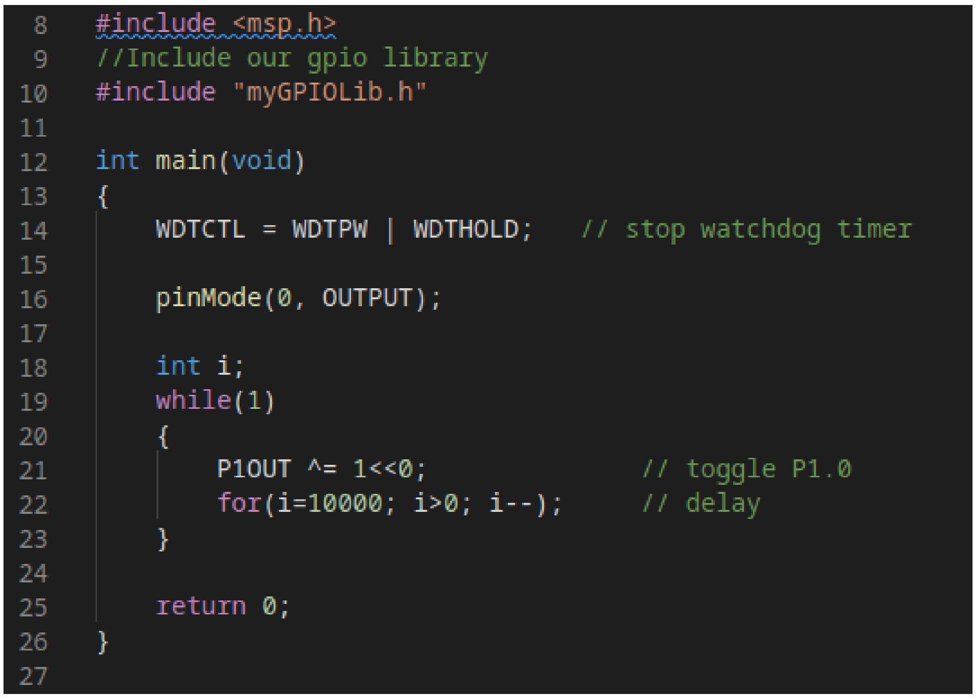
- The first function to implement is the pinMode function. We need to define the equivalent OUTPUT and INPUT constants, that the Wire language uses, as well as the function declaration of pinMode. Hence, the content of our myGPIOLib.h header file should look as follows


```c
#include <msp.h>

#define OUTPUT 1
#define INPUT 0

void pinMode(char pin, char mode);

```

- We now need to implement the function in the corresponding myGPIOLib.c file. The pinMode implementation is a simple one and only modifies the P1DIR register


```c
#include "myGPIOLib.h"

void pinMode(char pin, char mode){
    if(mode==OUTPUT)
        P1DIR |= (1<<pin);
    else if(mode==INPUT)
        P1DIR &= ~(1<<pin);
}

```

- Let's now look at the function in use. We are going to go back to our blinking LED example. We will include our new library and make use of our pinMode function.



```c
#include <msp.h>
//Include our gpio library
#include "myGPIOLib.h"

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    pinMode(0, OUTPUT);

    int i;
    while(1)
    {
        P1OUT ^= 1<<0;              // toggle P1.0
        for(i=10000; i>0; i--);     // delay
    }

    return 0;
}
```

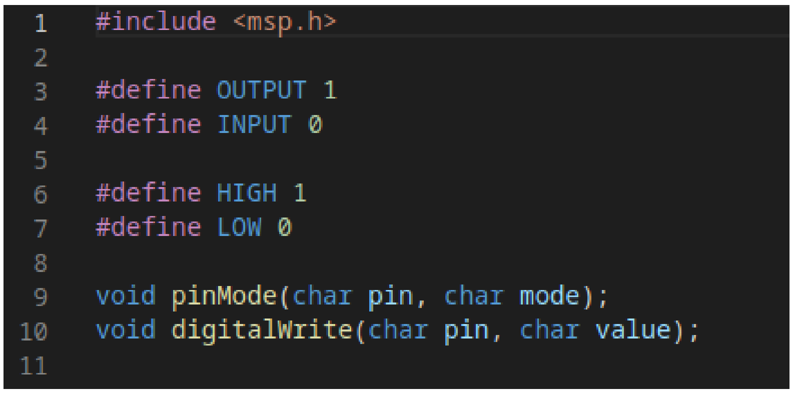
2.1.1 EXERCISE 4

The new library we created has a limitation, in that the functions can only operate on Port 1 pins, that is P1.X pins. Modify your library to include Port 2 pins, that is, P2.X pins.

2.2 MyGPIOLib: digitalWrite

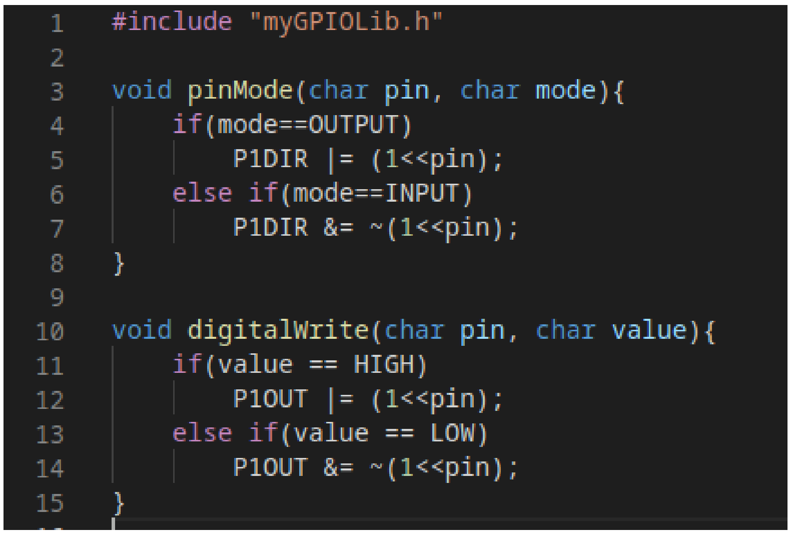
Learning outcome: Expand your library to include functions for writing output values to pins.

- Create a new project and name it "Ex4-MyGPIOLib-digitalWrite".
- Copy your myGPIO header and program files from the previous subtask.
- Append the constants HIGH and LOW, as well as a function declaration for *digitalWrite*, in your GPIO library header file, as shown in the screenshot below.



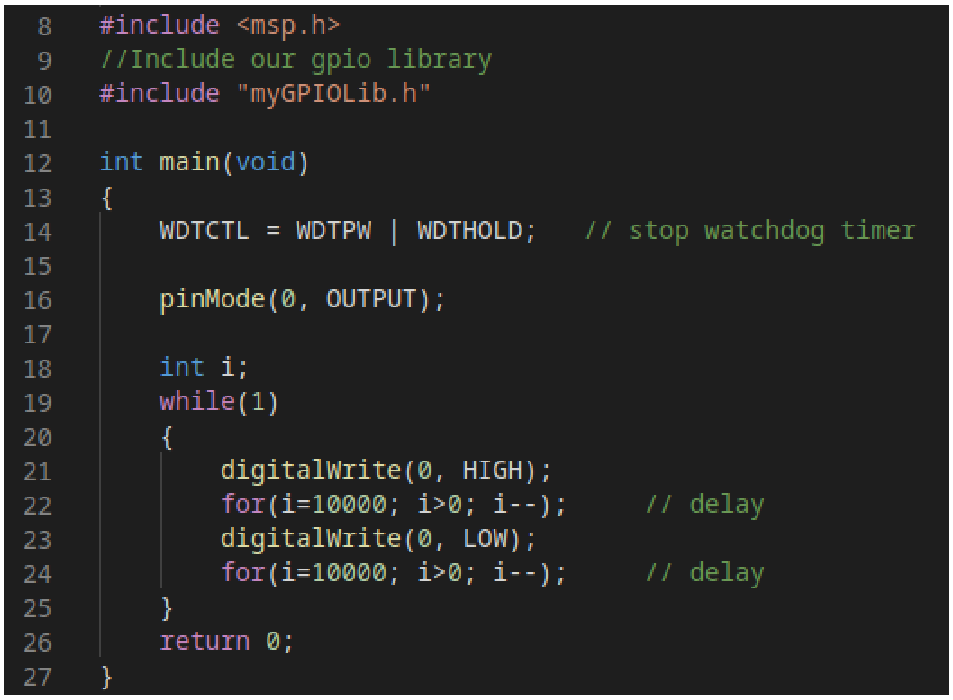
```c
#include <msp.h>

#define OUTPUT 1
#define INPUT 0

#define HIGH 1
#define LOW 0

void pinMode(char pin, char mode);
void digitalWrite(char pin, char value);
```

- The implementation is once again a simple one. It only modifies the P1OUT register. We append this to our original C file for myGPIOLib.



```c
#include "myGPIOLib.h"

void pinMode(char pin, char mode){
    if(mode==OUTPUT)
        P1DIR |= (1<<pin);
    else if(mode==INPUT)
        P1DIR &= ~(1<<pin);
}

void digitalWrite(char pin, char value){
    if(value == HIGH)
        P1OUT |= (1<<pin);
    else if(value == LOW)
        P1OUT &= ~(1<<pin);
}
```

- Once we have updated our GPIO library, we return to our blinking LED code and make use of it.



```c
 8  #include <msp.h>
 9  //Include our gpio library
10  #include "myGPIOLib.h"
11
12  int main(void)
13  {
14      WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
15
16      pinMode(0, OUTPUT);
17
18      int i;
19      while(1)
20      {
21          digitalWrite(0, HIGH);
22          for(i=10000; i>0; i--);     // delay
23          digitalWrite(0, LOW);
24          for(i=10000; i>0; i--);     // delay
25      }
26      return 0;
27  }
```

2.2.1 EXERCISE 5

Modify your library to accommodate Port 2 pins.

2.2.2 EXERCISE 6

Add your own function to toggle the value of the pin. You can then replace the four lines of the blink code (shown in lines 21-24 above) with just two lines of code.

NOTE: The technique employed in this task to implement a delay is very inefficient. The delay is generated using a ‘for’ loop which tells the CPU to do “nothing” for the number of iterations specified in the loop. A much more efficient (and proper) technique to generate delays is by using interrupts and timers. We shall explore this technique in subsequent lectures.

3 Task 3 – Reading Inputs

So far we have demonstrated how to use GPIO pins as outputs. We shall now explore how to use GPIO pins as inputs.

3.1 Read a push button

Learning outcome: Learn how to configure your program to use the on-board push buttons.

Looking at the schematic diagram for the MSP432P401R LaunchPad (or indeed looking at the board itself), you will observe that push button PUSH1 is connected to pin P1.1. We therefore wish to set P1.1 to be an input.

To set P1.1 to be an input, we need to clear the corresponding bit of the P1DIR register, that is, we need to set the value of this bit to zero.

```
P1DIR &= ~(1<<1); //Define P1.1 as an input
```

An additional configuration that we need to make for input pins is setting the pull-up or pull-down resistors. This means that any input pin should have a resistor pulling its value to Vcc or to ground (0V). This stops the pin from floating and causing errors. The resistor is referred to as either being a pull-up or pull-down resistor depending on which supply it pulls the pin to. The MSP432 is only capable of implementing a pull-up resistor.

When the button is not pressed, the pull-up resistor will pull the pin to Vcc. When the button is pressed, the pin is pulled to 0V.
The pull-up resistors are controlled using the P1REN and P1OUT registers.

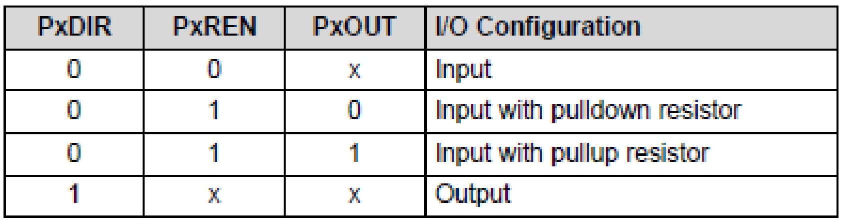
- A bit value of 1 in the P1REN register enables the (pull-up/pull-down) resistors for that corresponding pin. In the example below, we enable the resistors for pin P1.1.

  ```
  P1REN |= (1<<1); //Enable resistors on P1.1
  ```

- A bit value of 1 in the P1OUT register enables the pull-up resistors for that corresponding pin. In the example below, we enable the pull-up resistor for pin P1.1.

  ```
  P1OUT |= (1<<1); //Enable a pull-up resistor on P1.1
  ```

Table 12-1. I/O Configuration



PxDIR	PxREN	PxOUT	I/O Configuration
0	0	x	Input
0	1	0	Input with pulldown resistor
0	1	1	Input with pullup resistor
1	x	x	Output

- We still wish to make use of LEDs in this task so let's set P1.0 to be an output.

  ```
  P1DIR |= (1<<0); //Define P1.0 as an output;
  ```

  The value of an input pin is read using the P1IN register. You must select only the bit corresponding to the pin in question. This is done using bit masking[1], that is, carrying out an AND operation with the correct mask.

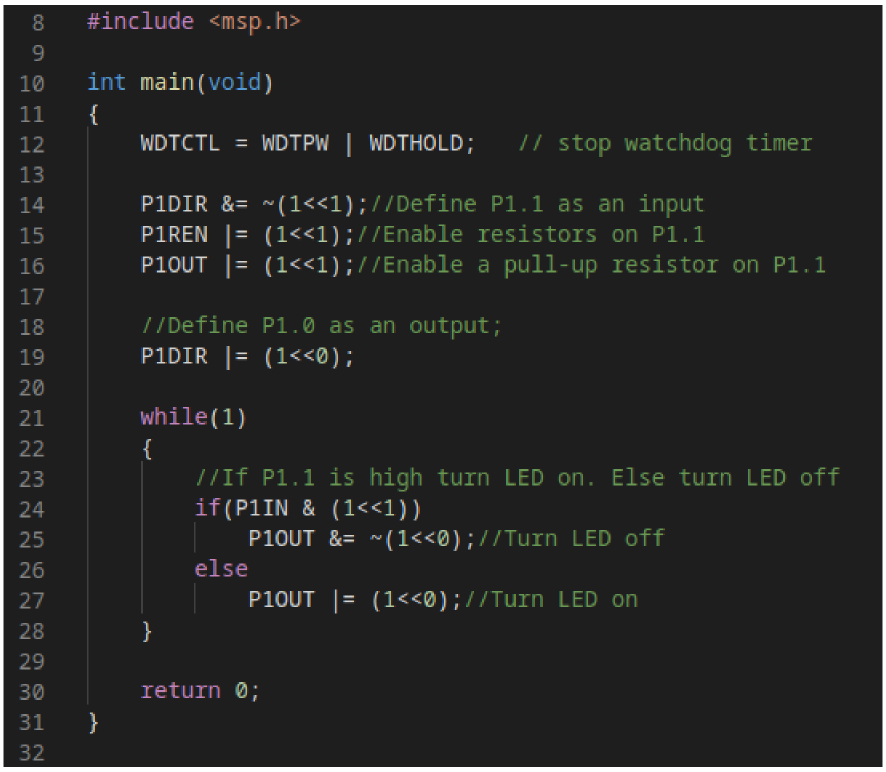
[1] If you do not understand bit masking, then look at this tutorial: `https://www.arduino.cc/en/Tutorial/BitMask`

- In this task, we read the value of our input pin (P1.1). If equal to 1, the button is not pressed and LED1 is turned off. If equal to 0, the button is pressed and LED1 is turned on.

  Take notice of the bit masking that is being used within the *if* statement.

```
// If P1.1 is high turn LED on. Else turn LED off
if(P1IN & (1<<1))
    P1OUT &= ~(1<<0); //Turn LED off
else
    P1OUT |= (1<<0); //Turn LED on
```

- Your code should eventually look like the one in the following screenshot.



```
#include <msp.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    P1DIR &= ~(1<<1);//Define P1.1 as an input
    P1REN |= (1<<1);//Enable resistors on P1.1
    P1OUT |= (1<<1);//Enable a pull-up resistor on P1.1

    //Define P1.0 as an output;
    P1DIR |= (1<<0);

    while(1)
    {
        //If P1.1 is high turn LED on. Else turn LED off
        if(P1IN & (1<<1))
            P1OUT &= ~(1<<0);//Turn LED off
        else
            P1OUT |= (1<<0);//Turn LED on
    }

    return 0;
}
```

3.1.1 EXERCISE 7

Modify your code to use push button PUSH2 on the MSP432 LaunchPad.

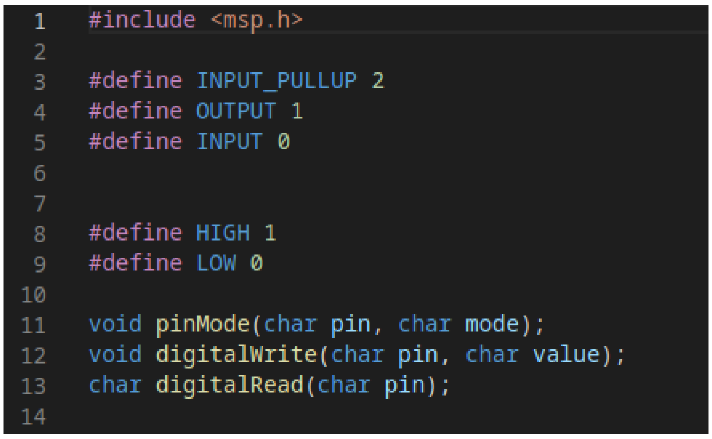
3.1.2 EXERCISE 8

Modify your code so that LED1 is only on when both push buttons are pressed concurrently.

3.2 MyGPIOLib: inputs

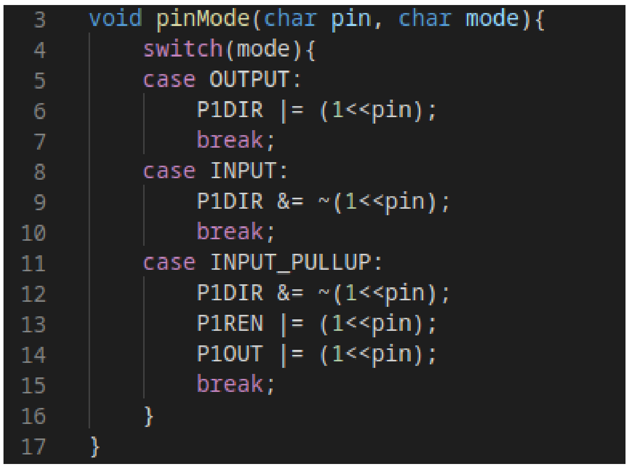
Learning outcome: Expand your library to include functions for reading input pins.

- Create a new project and name it "Ex6-MyGPIOLib-inputs".
- Copy your myGPIOLib header and program files from Task 2.
- Define a new INPUT_PULLUP constant in your header file, as well as a function declaration for *digitalRead*. This constant is used by the *pinMode* function to enable the pull-up resistor for specific input pins.



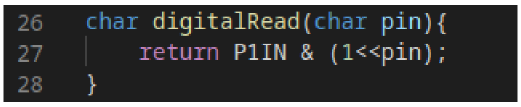
```c
#include <msp.h>

#define INPUT_PULLUP 2
#define OUTPUT 1
#define INPUT 0


#define HIGH 1
#define LOW 0

void pinMode(char pin, char mode);
void digitalWrite(char pin, char value);
char digitalRead(char pin);

```

- In the myGPIOLib program file, you need to modify the pinMode function to handle the P1REN register for any GPIOs defined as INPUT_PULLUP pins.



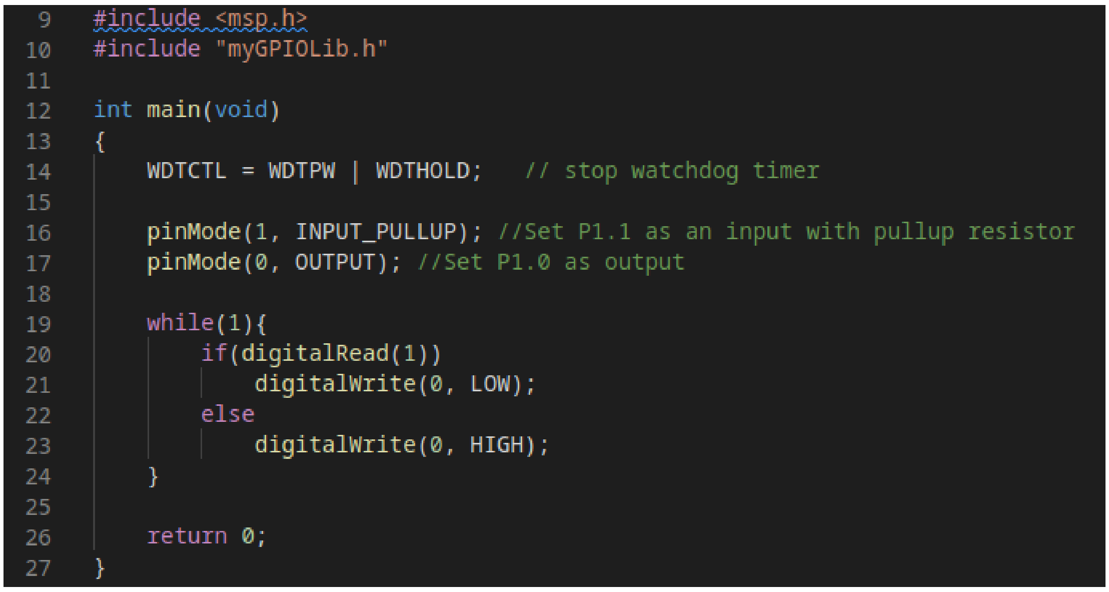
```c
void pinMode(char pin, char mode){
    switch(mode){
    case OUTPUT:
        P1DIR |= (1<<pin);
        break;
    case INPUT:
        P1DIR &= ~(1<<pin);
        break;
    case INPUT_PULLUP:
        P1DIR &= ~(1<<pin);
        P1REN |= (1<<pin);
        P1OUT |= (1<<pin);
        break;
    }
}
```

- You also need to implement the function for digitalRead. While simple, it does use bit masking.



```c
char digitalRead(char pin){
    return P1IN & (1<<pin);
}
```

- Finally, your main code should look like the one in the following screenshot



```c
#include <msp.h>
#include "myGPIOLib.h"

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    pinMode(1, INPUT_PULLUP); //Set P1.1 as an input with pullup resistor
    pinMode(0, OUTPUT); //Set P1.0 as output

    while(1){
        if(digitalRead(1))
            digitalWrite(0, LOW);
        else
            digitalWrite(0, HIGH);
    }

    return 0;
}
```

Hence, we have now reached the point where we have been able to read a push button and toggle an LED using a set of functions that we created (in our “myGPIOLib” library) which resemble those we used in Energia sketches in the past.

You have completed the journey from the ‘Wire’ programming technique to the ‘Register-level’ programming technique.

In the next task, we shall explore additional, advanced concepts configuring clocks and GPIO registers.

4 Task 4 – RGB LED

In the next task, we shall demonstrate how we can configure the RGB LED (LED2) on the MSP432 LaunchPad.

4.1 Configuring the MSP432's RGB LED

Learning outcome: Learn how to configure the RGB LED (LED2) on the MSP432 LaunchPad.

The MSP432 has an integrated RGB LED. We can mix the RGB components to produce a variety of colours. The LED itself consists of three separate LEDs; a red, green and blue LED each of which are connected to pins P2.0, P2.1 and P2.2 respectively.

In this task, we will use the two push buttons on the MSP432 board to turn the green and blue components on or off, while leaving the red component on at all times. This will demonstrate the versatility of the RGB LED.

- Create a new project and name it "Ex7-rgbled"
- The first thing we need to do is to configure the inputs. We will use the push buttons connected to pins P1.1 and P1.4, both of which require a pull-up resistor. We are modifying the same registers as we did in the previous tutorials. That is, P1DIR defines whether the pin is an input or output, while P1REN enables the pull-up resistor for a given input.

```
// Set P1.1 and P1.4 as inputs with pull up resistors enabled
P1DIR &= ~((1<<1)|(1<<4));
P1REN |= (1<<1)|(1<<4);
P1OUT |= (1<<1)|(1<<4);
```

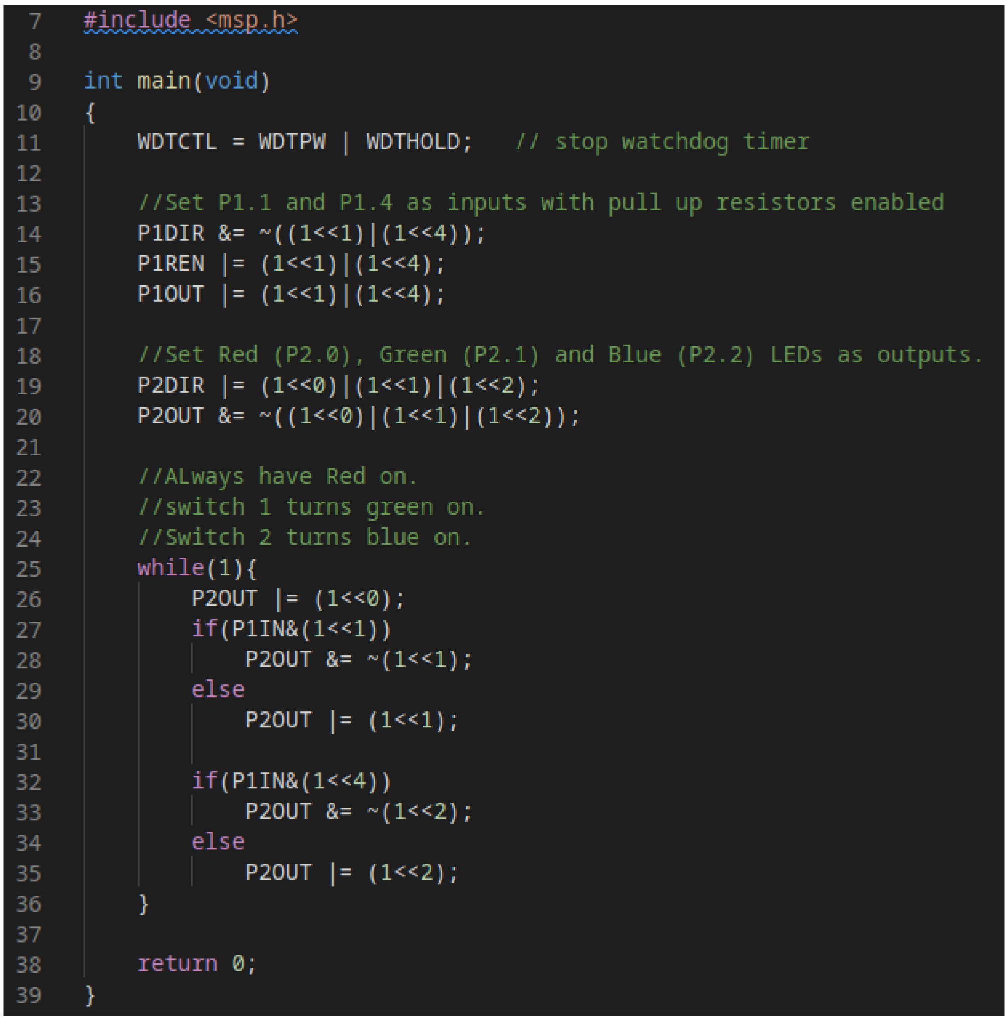
- We now need to configure the output pins connected to the RGB LED. All RGB pins are connected to Port 2 and all are turned off initially. As a reminder, P2DIR defines the pin as either an input or output and P2OUT sets the voltage at the pin

```
//Set Red (P2.0), Green (P2.1) and Blue (P2.2) LEDs as outputs.
P2DIR |= (1<<0)|(1<<1)|(1<<2);
P2OUT &= ~((1<<0)|(1<<1)|(1<<2));
```

  Finally, we need to code the actual behaviour we wish to implement. The red component will always be on, whereas for the green and blue components, we will use a conditional if statement. If their corresponding push button is pressed, that component will be turned on, otherwise it will be turned off.

- The above logic is executed out in an infinite while loop and is shown in the screenshot below.

  The while loop avoids repeating the configuration of the pins (endlessly), when this initialisation only needs to be done once at the beginning.



```c
#include <msp.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer

    //Set P1.1 and P1.4 as inputs with pull up resistors enabled
    P1DIR &= ~((1<<1)|(1<<4));
    P1REN |= (1<<1)|(1<<4);
    P1OUT |= (1<<1)|(1<<4);

    //Set Red (P2.0), Green (P2.1) and Blue (P2.2) LEDs as outputs.
    P2DIR |= (1<<0)|(1<<1)|(1<<2);
    P2OUT &= ~((1<<0)|(1<<1)|(1<<2));

    //ALways have Red on.
    //switch 1 turns green on.
    //Switch 2 turns blue on.
    while(1){
        P2OUT |= (1<<0);
        if(P1IN&(1<<1))
            P2OUT &= ~(1<<1);
        else
            P2OUT |= (1<<1);

        if(P1IN&(1<<4))
            P2OUT &= ~(1<<2);
        else
            P2OUT |= (1<<2);
    }

    return 0;
}
```