EMBEDDED SYSTEMS LABORATORY

LABORATORY MANUAL

Configuring GPIO on MSP432

AY 22/23

OBJECTIVES

- Get to know the MSP-EXP432P401R
- Learn how to use registers for configuration
- Get to understand and use library functions
- Configuring GPIO as input and output

EQUIPMENT

- Computer or laptop that supports Code Composer Studio (CCS) 9.3 IDE
- Texas Instrument MSP-EXP432P401R LaunchPad Development Kit
- Micro-USB Cable

NOTE:

- Only students wearing fully covered shoes are allowed in the lab due to safety.
- Bring your laptop with Code Composer Studio installed.
- For your understanding and a better quiz preparation, take notes about the given tasks, especially questions or unexpected code behaviour.

Introduction

In this lab session, we learn about the development platform MSP-EXP432P401R LaunchPad and the Code Composer Studio (CCS), which are used throughout all ES labs. We will also familiarise you with concepts like direct register access, polling, and serial communication. The following introduction should give a broad overview of the working environment, but it is unnecessary to comprehend all the details to complete this laboratory.

MSP-EXP432P401R LaunchPad Development Kit

The MSP-EXP432P401R LaunchPad Development Kit is an evaluation module based on the TI MSP432P401 microprocessor. The MSP432P401 is a low power mixed-signal microprocessor featuring a 32-Bit ARM Cortex M4F CPU. It contains several different memories (flash main memory, information memory, SRAM, and ROM) and has the Texas Instruments (TI) MSP432P401 peripherals driver libraries installed. Furthermore, it offers flexible clocking (tune-able clock sources), several timing units (timers, Pulse Width Modulation (PWM), capture and compare, etc.) and serial communication interfaces (Universal Asynchronous Receiver Transmitter (UART), I2C, Serial Peripheral Bus (SPI), etc). The MSP432P401 also includes an Analogue to Digital Converter (ADC) unit, analogue comparators, and various Input/Output (I/O) pins.

In addition to the microprocessor, the kit features simple peripherals (buttons, LEDs, etc.) and the XDS110-ET debug probe. The XDS110-ET debug probe allows direct programming of the target device (in our case, the MSP432P401) without using an external programming device and allows debugging an application and direct communication with the PC. The block diagrams in Figure 2 illustrate the basic structure of the MSP-EXP432P401R. All debug and communication signals run through the XDS110-ET debug probe; therefore, there is no direct connection between the target and the computer. A direct link between the computer and the MSP432P401 is only possible using an external setup. Furthermore, for high precision tests, it is possible to isolate the target through the J101 Isolation block (jumpers).

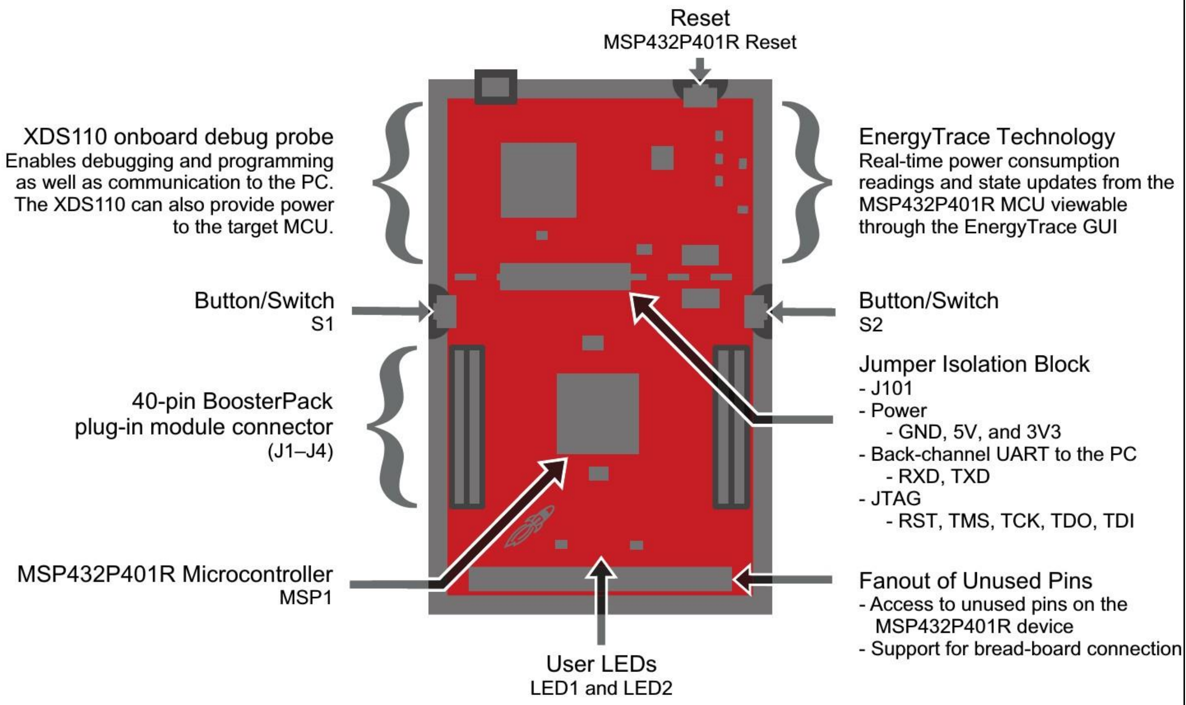
Reset
MSP432P401R Reset
XDS110 onboard debug probe
Enables debugging and programming as well as communication to the PC. The XDS110 can also provide power to the target MCU.
EnergyTrace Technology
Real-time power consumption readings and state updates from the MSP432P401R MCU viewable through the EnergyTrace GUI
Button/Switch
S1
Button/Switch
S2
Jumper Isolation Block
- J101
- Power
- GND, 5V, and 3V3
- Back-channel UART to the PC
- RXD, TXD
- JTAG
- RST, TMS, TCK, TDO, TDI
40-pin BoosterPack plug-in module connector
(J1–J4)
MSP432P401R Microcontroller
MSP1
Fanout of Unused Pins
- Access to unused pins on the MSP432P401R device
- Support for bread-board connection
User LEDs
LED1 and LED2

Figure 1: A basic overview of the MSP432P401R Launchpad Development Kit

Embedded System Documentation

The most important documents of an embedded system are, among others, **datasheets**, **user guides**, **technical reference manuals**, **application notes**, **errata** or **schematics**. These documents are available for:

- the microprocessor at the manufacturer’s website, e. g. TI (http://www.ti.com/)
- components of the microprocessor, e. g. ARM microprocessors documentation
- hardware boards, e. g. the TI MSP-EXP432P401R LaunchPad

- software, e.g. the TI MSP432P401 Peripheral Driver Library
- complementary information in wikis or online forum

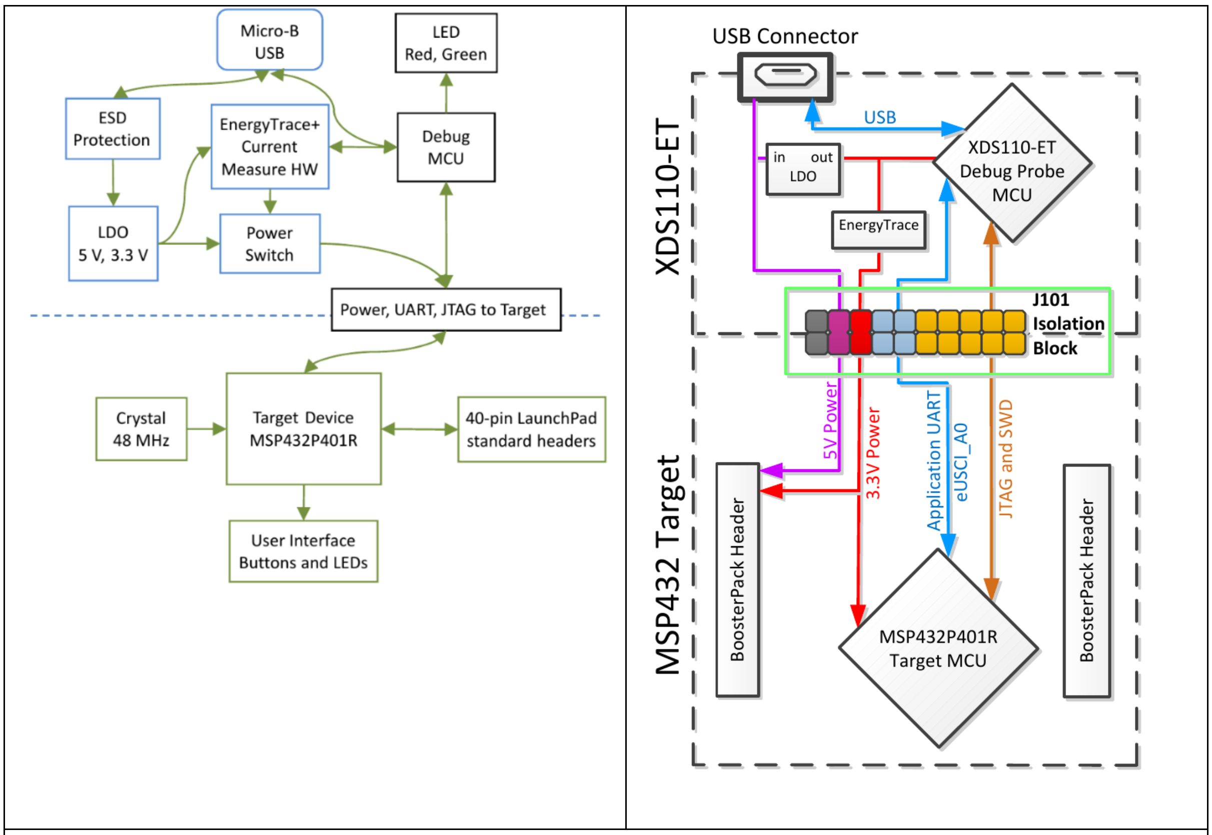

Figure 2: Block diagrams of the MSP-EXP432P401R LaunchPad Development Board. A general block diagram (left) and one highlighting the XDS110-ET isolation block and the debug components (right).

Therefore, every embedded system comes with many documentation files. For this lab and subsequent labs, all the necessary files are provided on the course xsite website. It is essential to have access to all parts of the documentation to use the functionality of an embedded system to its fullest extent. The documentation is typically needed to control:

- the pins peripherals are connected to
- register addresses and memory-mapped peripherals
- configuration values for modules
- bitmasks and their meaning
- the functionality of modules (e.g. serial communication modules, etc.)

Bare-Metal Programming

Programming bare-metal means to program an embedded system without using an underlying Operating System (OS) like Linux, FreeRTOS or a scheduler of any sort. Typically, they consist of a base loop, and all activity is either polled or interrupt-driven. It may include vendor-supplied header files or driver subroutines, but the code determines what happens when and is in total control. These systems are the most deterministic, are easier to test and debug, and are the way to go for most microcontroller-based applications that tend to be specific.

Watchdog Timer

The watchdog timer is an internal timer unit of an embedded system that will reset the system if it times out. Under normal operation, the system will reset the watchdog timer regularly to prevent it from timing out, but in case of an error, the watchdog timer will reset the device. This is used to avoid a deployed embedded system that cannot be easily accessed (e.g. Mars Rover) from being stuck if an error occurs by resetting it once the timer expires. Do note that the watchdog timer for the MSP432 is **turned on by default**. Thus, if you do not reset the watchdog timer before the deadline, your device will reset, leading to unnecessary frustration and unwanted debugging. Therefore, as the watchdog feature is not required during our lab sessions, we will disable it (see Code Snippet #1). The watchdog timer will be taught in greater detail in the Week 4 Lecture.

```
// Stopping the Watchdog Timer
 WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;
```

Code Snippet 1: Stopping the Watchdog Timer

Consider This: Can you find out what is the register address to disable the watchdog timer? What data is written to this register address to disable the watchdog timer? What could happen if this instruction was completed later (e.g. after line 31)?

Defines, Macros and in-line Functions

Preprocessor directives like defines and macros, and in-line functions are often used when programming embedded systems to make the code more readable and easier to maintain. Examples for a ***define***, ***macro*** and ***in-line*** function are given in Code Snippet 2, 3 and 4. The ***#define*** directive, used for ***defines*** and ***macros***, is a preprocessor directive that instructs the compiler to replace certain code parts as defined. In contrast to that, the ***in-line*** function is just a request to the compiler. This means, depending on the optimisation configuration of the compiler, function calls to ***in-line*** functions can be replaced with the function body or are treated like standard function calls. However, in contrast to ***macros***, ***in-line*** functions are subject to strict parameter checking and considered *safer*.

```
#define GPIO_PIN0            (0x0001)
```

Code Snippet 2: Example for #define

```
#define MACRO_lab1_configureUART(config)
{
    /* Selecting P1.2 and P1.3 in UART mode    */
    GPIO_setAsPeripheralModuleFunctionOutputPin(                   \
            GPIO_PORT_P1, GPIO_PIN2 | GPIO_PIN3,                   \
            GPIO_PRIMARY_MODULE_FUNCTION);                         \

    /* Configuring UART Module          */
    UART_initModule(EUSCI_A0_BASE, config);

    /* Enable UART Module */
    UART_enableModule(EUSCI_A0_BASE);
}
```

Code Snippet 3: Examples for a macro

```
Inline void lab1_configureUART(const eUSCI_UART_Config *config)
{
    /* Selecting P1.2 and P1.3 in UART mode      */
    GPIO_setAsPeripheralModuleFunctionOutputPin(                    \
            GPIO_PORT_P1, GPIO_PIN2 | GPIO_PIN3,                    \
            GPIO_PRIMARY_MODULE_FUNCTION);                          \

    /* Configuring UART Module           */
    UART_initModule(UART_INTERFACE, config);

    /* Enable UART Module */
    UART_enableModule(UART_INTERFACE);
}
```

Code Snippet 4: Examples for an in-line function

Consider This: When would in-line functions be better than macros?

A more detailed explanation and their differences can be found on the following site: https://www.thegeekstuff.com/2013/04/c-macros-inline-functions/

Task 1: Library Functions vs Hard-Coded Register Access

Using the CCS, we can quickly establish the connection to an MSP-EXP432P401R connected via USB and program it. This task will teach you how to flash a functioning application using GPIO functions onto the microprocessor. These GPIO pins can be configured either as input or output. GPIO pins can also drive peripherals (i.e. LEDs, small actuators, relays) or read inputs (i.e. sensors, buttons). Furthermore, you will learn how to swap and extend the application using hard-coded register access and library functions.

In this task, we want to use hard-coded register access instead of any library functions. Load the LAB2 sample code and modify the hard-coded register access to library functions. Code Snippet 5 illustrates the macros that will be used to replace the hard-coded register access. Then build the program and observe if the behaviour is the same.

```
/* These are register base address and register address offsets */
/* They can be found in the MSP432P4xx Reference Manual Chapter 10 */

#define REGBASEADR ((uint32_t)(0x40004C00))  // Base addr. of Port 1 configuration register
#define REGOFS_SEL0 ((uint32_t)(0x0000000A))  // Addr. Offset for Select0-Register in Port 1
#define REGOFS_SEL1 ((uint32_t)(0x0000000C))  // Select1 offset in Port 1 conf. reg.
#define REGOFS_DIR ((uint32_t)(0x00000004))     // Direction offset in Port 1 conf. reg.
#define REGOFS_OUTV ((uint32_t)(0x00000002)) // Output Value offset in Port 1 conf. reg.
```

Code Snippet 5: Macros defined for Task 1

When programming embedded systems, it is possible to access special registers to allow low-level hardware configurations directly. This can either be done through macros using addresses or predefined structs. In many cases, manufacturers provide a software library called Hardware Abstraction Layer (HAL) or drivers to make register access easier when developing applications for either bare-metal or OS-based.

Task 1.1: Making it easier to understand

When you open your LAB2, you will notice the code using direct registers access. The following line of code is used to set **bit 0** of **Select 0 register** to **0** as shown in Code Snippet 6.

```
(*((volatile uint16_t*)(((uint32_t)(0x40004C00)) + ((uint32_t)(0x0000000A)))))&= ~(0x0001);
```

Code Snippet 6: Set Bit 0 of Select 0 Register to 0

This is a step-by-step example of how we would go about converting such a line of code using direct registers to use macros to make it easier to understand the code. Using code snippet 6 as a guide, to compute the address of the **Select 0 Register**, you will need the Base Address (in Red) and the Select 0 Offset (in purple). Refer to your reference manual chapter 10; you will notice that the Base Address is 0x4C00 while Select 0 Offset is 0x000A. Include the macros defined in code snippet 5 into the main.c file.

To set Bit 0 of Port 1, we need to find the address of this bit (0x0001) (seen in blue for code snippet 6). Therefore, we can declare a label BIT0 as follows:

```
#define BIT0 (uint16_t)(0x0001)
```

Therefore, integrating the above information provided, we can convert from code snippet 6 to realize the following:

```
(*((volatile uint16_t*)(REGBASEADR + REGOFS_SEL0))) &= ~(BIT0)
```

We can now set the bit 0 to 0 by combining these bitwise operators, AND (&) and NOT (~). Do note that the (~) operator transpires before the (&) operator. The (~) operator will convert BIT0 which is defined as 0x0001 to become 0x1110. The subsequent (&) operator will ensure that only the least significant bit (bit 0) will be set to zero as anything AND-operated with a zero will lead to zero (bitwise AND-operator).

Attempt to convert the other lines of code using the macros defined in code snippet 5.

Consider This: What is the observable outcome of this code?

Task 2: Incorporating DriverLib

In the previous task, we learn how to incorporate #defines and macros into the programs. However, there are times when drivers or libraries have been developed in the previous project and are readily used in future projects. In addition, these libraries have typically been tested and verified. In some cases, they are also optimised for a specific processor or platform. In this task, we will be incorporating the Driverlib library developed by Texas Instruments (TI). Both the source code and user guide can be found on TI's website. Do note to download the correct version of the library (for MSP432P4xx) for fewer debugging sessions. :P

The MSP432 Driver Library (DriverLib) is a set of fully functional APIs used to configure, control, and manipulate the hardware peripherals of the MSP432 platform. The DriverLib is meant to provide a "library" layer to the programmer to facilitate a higher programming abstraction than direct register accesses. Every aspect of an MSP432 device can be configured and operated using the DriverLib APIs. The high-level library APIs provided by DriverLib facilitates users to rapidly generate powerful and intuitive code that is portable between the MSP432 platform variations and potentially between different families in the MSP430 platforms.

Subsequent laboratory sessions, lab-test and group projects will utilise the DriverLib. In the next section, we shall look into using the DriverLib to implement Task 1.1.

Task 2.1: Configuring GPIO Pins as Output (LED)

If you did not know by now, the code in Task 1.1 would configure GPIO P1.0 that is linked to LED1 (Red LED) as an output pin and toggle the pin high/low every second. The three lines of code (shown below) that configure the GPIO pin can be replaced with "GPIO_setAsOutputPin".

```
(*((volatile uint16_t *)(((uint32_t)(0x40004C00)) + ((uint32_t)(0x0000000A))))) &= ~(0x0001);
(*((volatile uint16_t*)(((uint32_t)(0x40004C00)) + ((uint32_t)(0x0000000C))))) &= ~(0x0001);
(*((volatile uint16_t*)(((uint32_t)(0x40004C00)) + ((uint32_t)(0x00000004))))) |= (0x0001);
```

A single API call as seen below takes in two inputs; port and pin number:

```
GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
```

The following line of code toggles the output GPIO pin:

```
(*((volatile uint16_t*)(((uint32_t)(0x40004C00)) + ((uint32_t)(0x00000002))))) ^= (0x0001);
```

This can be replaced with a single API call that also takes in the port and pin number:

```
GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
```

It can be observed that using the DriverLib makes the code more legible and neat. Moreover, libraries developed by the hardware vendors are more optimised for their products and are better supported.

Consider This: Modify the code to toggle the LED1 and LED2 (in purple) using DriverLib? LED2 is an RGB LED, and unlike LED1 that is only Red when turned on (P1.0), the LED2 can be controlled by three separate GPIO pins that can turn on/off Red, Green or Blue LED (P2.0, P2.1 & P2.2) in any combination.

Task 2.2: Configuring GPIO Pins as Input (Button)

In this task, we shall write an application that will switch the LED2 to each colour (Red → Green → Blue) on every S2 button press, while S1 button will toggle LED1. A method to continuously sample certain information (i.e. register or input) is called polling. This can be used to access peripheral or internal status information or read data. GPIO pins can also be used for input. In this task, we want to configure two GPIO inputs to enable user interaction using the S1 and S2 on the MSP-EXP432P401R connected to P1.1 and P1.4, respectively. To avoid a floating pin, pull-up or pull-down resistors are used. A pin is called floating when the pin does not have a fixed voltage level, leading to unexpected behaviours if the pin is read. In Figure 3, the two concepts of pull-up and pull-down resistors are illustrated. A pull-up resistor will ensure the pin is on high voltage level when the button is not pressed (open circuit) and low if the button is pressed (short circuit). The pull-down resistor works vice versa.

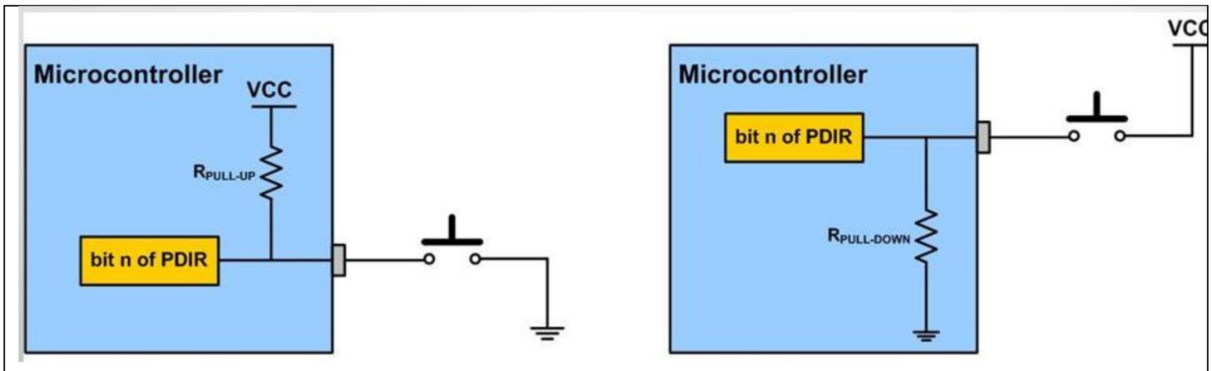

Figure 3: Schematic of a pull-up (left) and pull-down resistor

The following API call (that also takes in port and pin number as input) can be used to configure the GPIO as an input pin:

GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);

There are two other API that can be used to configure the GPIO as input pin:

GPIO_setAsInputPin(GPIO_PORT_P1, GPIO_PIN1);

GPIO_setAsInputPinWithPullDownResistor(GPIO_PORT_P1, GPIO_PIN1);

Consider This: Why is the GPIO input pin configured as a pull-up resistor?

Once the GPIO is configured as an input, the pin can now be polled to check if the button has been pressed using the following API call that takes in the port and pin number and returns an uint8_t type data: GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1) as seen in code snippet 6.

```
void main(void)
{
  WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;  // Stopping the Watchdog Timer
   …
  GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
  GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);
  GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);

  while(1)
  {
    if(GPIO_getInputPinValue(GPIO_PORT_P1,GPIO_PIN1) == ((uint8_t)0x00))
          GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
    for(count = 0; count < g_waitcycles; count++)   { }
   }
}
```

Code Snippet 6: Configuring P1.1 as GPIO input

Complete Task 2.2 by finishing the functionality of Button S2.

Submission

Implement the following:

1. **The red LED1 and red RGB LED must blink alternatingly.**
2. **While button S1 is pressed, the green RGB LED turns on.**
3. **While button S2 is pressed the blue RGB LED turns on.**
4. **Note that the red LED1 and red RGB LED must continue to blink alternatingly when the S1 and S2 buttons are pressed.**

Only submit the .C file to DropBox. Include both team members' names.