Lab E: Sensing Buttons

Introduction

Single Sentence Overview

Learn how to use the interrupt mechanism in your microcontroller via the push button switch.

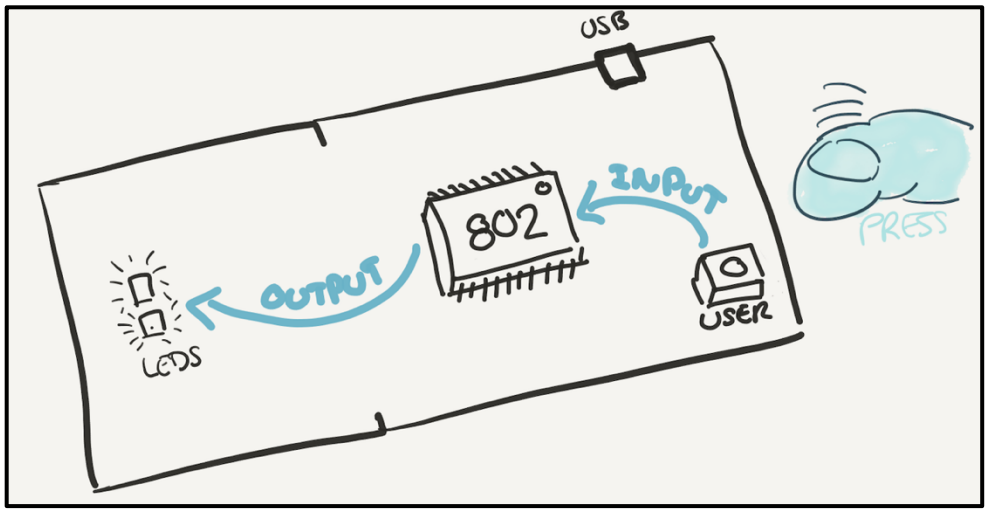


Figure 1 In this lab you are press a button, have the microcontroller detect that button press and then to show that the microcontroller detected the button press by lighting up some LEDs.

Overview of Lab E

This three-part lab is designed to introduce the student to a debugging-centric method for developing a program on a microcontroller. The three parts are:

1. Sense a button input using the debugger (repeat from Lab D)
2. Sense a button in your main loop using a polling (blocking) method
3. Sense a button using an Interrupt Service Routine (non-blocking) method

On the LPC802 board you will use

1. The button connected to GPIO 8 (PIO0_8)
    a. The button is labelled “USER” and is located on the right-hand side of the OM40000 boad.
2. The LEDs connected to GPIO 9 (PIO0_9).
    a. These LEDs appear as LD21 and LD22 on the snap-off portion on the left-hand side of the OM40000 board.

LPC 804	If you are using the LPC804 (OM40001 board), you will use the User Button S1 connected to GPIO 13 (PIO0_13). You will use PIO0_20 to connect to an LED, but the LED needs to be the one on the Capacitive Touch board, so you’ll need to attach that board.

Learning Outcomes of Lab E

By the end of the lab, the student will know how to

1. Use the debugger to manually access the GPIO input register linked to a push button, and to determine if the button is pressed or not
2. Write, in C, a method for polling a memory location to determine if a button has been pushed
3. Write, in C, a method for reporting that a button has been pushed using a non-blocking interrupt service routine.

Success Criteria

Review the grading rubric and evaluate how it relates to achieving the learning outcomes via the three distinct parts of the lab. You are expected to:

1. Demonstrate the use of the debugger in detecting a button press.
2. Demonstrate that a program written in C can detect the button press using a polling method inside the main function.
3. Demonstrate that a program written in C can detect the button press using an interrupt method. *You must show the TA the body of your main function to prove that it is not polling the button.*

Note that the use of C++ is considered equivalent but is not required.

Grading Rubric

During the lab session make sure to conduct all the required demonstrations to the lab instructor.

- *Part 1 demo (Debugger Pushbutton)*
  - *0 pts: no demonstration attempted, or student failed to show that the debugger could be used to read the state of the button.*
  - *1: student showed that the debugger could be used to read the state of the button.*
- *Part 2 demo (Poll the button)*
  - *0 pts: no demonstration attempted, or no functionality appears to work*
  - *1 pts: a polling routine inside the main function appears to be mostly correct but the button is not detected or does not result in a (due to a software routine) toggling of an LED.*
  - *3 pts: a polling routine inside the main function successfully detects a button push and (because of a software routine) toggles an LED as a result.*
- *Part 3 demo (Interrupting button)*
  - *0 pts: no demonstration attempted or no ISR engages.*
  - *3 pts: the ISR engages (detectable via breakpoint) but no LED is toggled via software as a result.*
  - *6 pts: The ISR captures the pushbutton and toggles an LED via a software call as a result.*

While demonstrating the ISR, you must show the TA the body of your main function to prove that it is not polling the button.

Be careful! The button has an LED connected to it that changes simple due to the electrical connection to the button. That LED will change no matter what is happening to the program running on the

microcontroller. The LED that must be demonstrated is one that you control using software running on the LPC802. Demonstrations that do not use the right LED will not be assigned full points.

There is **no requirement to submit a document**. Some labs have "stimulate" or "explore" questions. If this lab has such questions they are for general interest and may be discussed further in class.

Prerequisites

You are expected to have done the following before attending the lab session:

- Have attended class and performed the previous lab.

More Resources and Information

- The LPC802 board (OM40000) User Manual (available on Moodle)
- The LPC802 (OM40000) board schematic (available on Moodle)

Note that you can use *either* the C11 or the C++14 compilers in MCUXpresso for these exercises.

A note about weird symbols and words

There are all kinds of weird symbols and words that need tob e used when programming a microcontroller like the LPC802. It's intimidating. Anyone that tells you otherwise is either a liar or has forgotten what it was like when they first got started.

There are a few technical things to try to get accustomed to:

Figure 2 Confused? That's normal. The weird symbols and words that we use in the programs we write are strange. Let's take the time in class and in the labs to get familiar with them.

1. Memory names. Things like GPIO->NOT refer to modules inside the microcontroller. In this case it's the NOT register of the General Purpose Input/Output module. We put numbers inside of the NOT register using symbols like = ("equal"), |= ("or-equal") and &= ("and-equal")
    a. The ISO646 header file provides written word equivalents to these logic operators. You can use those if you find them helpful. See the table below for examples.
2. The numbers that are placed in the registered are defined in different ways:
    a. As a decimal (base ten) number (i.e. 64)
    b. As a binary (base two) number (i.e. 0b010000000)
    c. As a hexadecimal (base 16) number (i.e. 0x40)
    d. As a bit-shifted value (i.e. 1<<7 or 1UL<<7)
    e. As a predefined value (i.e. MY_VALUE where a header file contains #define MYVALUE (64))
3. The predefined values are found in either LPC802.h or LPC804.h
    a. Some pre-defined values are "masks"
        i. They can be used directly as numeric values
    b. Some pre-defined values are "bit-shift" values
        i. They are used with the << operator to typically change a single value in a memory register
    c. You're not expected to memorize these names or their numeric values
    d. You ARE expected to be able to look them up in the header file or in the data sheet.

The table below is an attempt to show you different ways we assign and manipulate numeric values to have an effect on registers in memory.

	Major Bitwise Logic Operation	Generic form (Replace myvar or mybit with your own variable or register and n with the bit number)	Explanation of Example	Typical (explicit)	Typical (compact with shifting)	"Alt ISO646" (explicit)	"Alt ISO646" (compact with shifting)
Set: Assign a bit to 1	Or	myvar |= (1 << n);	Make Bit 6 a "1".	myvar = myvar | 0b01000000;	myvar |= (1 << 6);	myvar = myvar bitor 0b01000000;	myvar or_eq (1<<3);
Clear: Assign a bit to 0	And, Complement	myvar &= ~(1 << n);	Make Bit 5 a "0".	myvar = myvar & 0b11011111;	myvar &= ~(1 << 5);	myvar = myvar bitand 0b10111111;	myvar and_eq compl(1<<5);
Toggle: Invert a bit	Xor	myvar ^= (1<<n);	Change Bit 4 to its opposite.	myvar = myvar ^ 0b01000000;	myvar ^= (1 << 4);	myvar xor_eq 0b00100000;	myvar xor_eq (1UL << 5);
Examine status of a bit	And	mybit = (number >> n) & 1;	What is the value of bit 5?	-	mybit = (myvar >> 5) & 1;	-	bit = (myvar >> 5) bitand 1;

17

1. Standard bitwise operations are explained here: https://en.wikipedia.org/wiki/Bitwise_operations_in_C
2. The alternate operator spelling is common in modern C++. To use the alternate operator spelling in C make sure to use #include <iso646.h> in your code.

Figure 3 We often use bitwise logical operations in C and C++ when dealing with register manipulation. This table outlines a few different ways that we can do it, using explicit binary values, bit-shifting and special operations from the iso646.h library.

Part 1: Debug the button

Use the debugger to manually access the GPIO input register linked to a push button to determine if the button is pressed or not. This is a repeat of the lab exercise in Lab D. Demonstrate this to the TA. This is a warm up exercise.

Part 2: Poll the button

Write, in C (or C++), a method for polling a memory location ("register") to determine if a button has been pushed. Start the top of your C source code file as follows:

Table 1 This is the start of your main C file.

Make sure to link to the right header file for your chip via the #include statement. This is dependent on the type of chip you're using (e.g. LPC802 or LPC804). Also, define words that replace the numeric values of the GPIO pins that you're interested in using for the button and switch. This makes your code more understandable and maintainable. The compiler's preprocessor will swap those defined words for numeric values before compiling happens (the parentheses are to ensure that numeric values and operators are clearly separated during this swap).

```
// Lab E, Part 2 C11 (LPC802)
// User button is on PIO0_8 (GPIO 8)
// LED is PIO0_9 (GPIO 9)
#include "LPC802.h"

#define LED             (9)             // One of the LED (PIO0_9)
#define BUTTON_USER1    (8)             // GPIO 8 (PIO0_8) is connected to the LED.
```

LPC 804	Note 1: use LPC804.h Note 2: if you are using the LPC804 (OM40001 board), you will use the User Button S1 connected to GPIO 13 (PIO0_13). You will use PIO0_20 to connect to an LED, but the LED needs to be the one on the Capacitive Touch board, so you'll need to connect that.

Table 2 This is how you start your main function.

Then start your main function

```
int main(void)
{
```

Inside of your main function, do the following:

Table 3 Turn on the GPIO module.

Turn on the GPIO modue inside the LPC802. It's off to save power. To do so, use the AHB Clock Control register of the System Control Module. (applies to both the '802 and '804)

Tip 1: type "SYSCON–>" and then wait to see how the IDE helps you with the next step.
Tip 2: Right click on SYSAHBCLKCTRL0 and click on "Open Declaration" in the menu and look for the defined values for mask values. This is a really useful way to look up their names (and values).

```
        // GPIO module turned on
        SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_GPIO0_MASK);   // GPIO is on
```

Table 4 Reset (restart) the GPIO module.

Reset the GPIO module using a mask for the Peripheral Reset Control register in the System Control Module. (applies to both the '802 and '804). The logical operators used here can be a little confusing at first. Check out the table in Figure 3 for more information on your options.

```
        // Put 0 in the GPIO reset bit to reset it.
        // Then put a 1 in the GPIO  reset bit to allow it to operate.
        // manual: Section 6.6.10
        SYSCON->PRESETCTRL0 &= ~(SYSCON_PRESETCTRL0_GPIO0_RST_N_MASK);  // bit to 0 (reset)
        SYSCON->PRESETCTRL0 |= (SYSCON_PRESETCTRL0_GPIO0_RST_N_MASK);   // bit to 1 (set)
```

Table 5 Configure the input and output pins on the GPIO module for button and LED.

Modify the GPIO settings. First, set up the pushbutton connection to GPIO 8 (a.k.a. PIO0_8) on the LPC802. Clear bit 8 by putting a 1 in Bit 8 of the Direction Clear register of the GPIO module. We use a bit shift trick on the right hand side of the equal sign to make the operation "easy" to follow (with practice). Then set up the LED pin (GPIO 9 on the LPC802) by first forcing its output value to be zero by putting a 1 in Bit 9 of the Clear register of the GPIO module. Then, set up the pin direction as an output by putting a 1 in Bit 9 of the Direction Set register of the GPIO module.

```
        // Config the Pushbutton (GPIO 8) for input and LED (GPIO 9) for output
        // Remember:    only bits set to 1 have an effect on DIRCLR and DIRSET registers.
        //                      bits cleared to 0 are ignored.
        // Therefore, use DIRCLR to select input and DIRSET to select output
        GPIO->DIRCLR[0] = (1UL<<BUTTON_USER1);  // input on GPIO 8 (BUTTON_USER1)

        GPIO->CLR[0]    = (1UL<<LED);// pre-emptive turning off the LED
        GPIO->DIRSET[0] = (1UL<<LED);// output on the LED
```

Table 6 Your main loop.

Create an infinite blocking loop. It will run forever due to the while(1). Inside, the if statement tests the Byte register, which is defined as a 2-dimensional array in the C header file. We're looking at row zero ("[0]") and column eight ("[8]" or "[BUTTON_USER1]"), which contains a logic value based on the state of the button. We AND that with 1 to test if it's Logic 1 or Logic 0. We then either clear (to zero) the output GPIO pin connected to the LED or set it (to one), thus changing the state of the LED.

```
        // Create a blocking loop
        while(1)
        {
                // check GPIO0 Byte register 8 for value on PIO0_8
                        if(GPIO->B[0][BUTTON_USER1] & 1)                // is it high?
                        {
                                // turn on the LED
                                GPIO->CLR[0] = (1UL<<LED);
                        }
                        else                                            // button is "low"
                        {
                                GPIO->SET[0] = (1UL<<LED);
                        }
        }
```

Table 7 Wrap up.

End your main function with a return 0;

```
        return 0;
}
```

Part 3: Interrupt the button

Write, in C (or C++), a method for reporting that a button has been pushed using a non-blocking interrupt service routine.

Sometimes we call Interrupts "Interrupt ReQuests" and the functions that we write to deal with them are called Interrupt Service Routines ("ISRs") or Interrupt ReQuest Handlers ("IRQ Handlers").

We start by defining what inputs and outputs we need:

Table 8 Define which I/Os you need.

Make sure to link to the right header file for your chip via the #include statement. This is dependent on the type of chip you're using (e.g. LPC802 or LPC804). Also, define the values for your GPIO.

```
// Lab E, Part 3 C11 (LPC802)
// User button is on PIO0_8 (GPIO 8)
// LED is PIO0_9 (GPIO 9)
#include "LPC802.h"

#define LED             (9)             // One of the LED (PIO0_9)
#define BUTTON_USER1    (8)             // GPIO 8 (PIO0_8) is connected to the LED.
```

LPC 804	Note 1: use LPC804.h Note 2: if you are using the LPC804 (OM40001 board), you will use the User Button S1 connected to GPIO 13 (PIO0_13). You will use PIO0_20 to connect to an LED, but the LED needs to be the one on the Capacitive Touch board, so you'll need to connect that.

To have the button trigger a call to the ISR we need to tell the microcontroller that we intend for that to happen. We will do that first, before defining the ISR. Start with your main function:

Table 9 Disable interrupt functinality while setting up for interrupts.

Disable all interrupt request (IRQ) sources, as well as the one specific to the GPIO pins. Do this at the top of your main function.

```
int main(void) {

    // disable interrupts
    __disable_irq();                            // turn off globally
    NVIC_DisableIRQ(PIN_INT0_IRQn);   // turn off the PIN INT0 interrupt.
```

Table 10 Turn on _both_ the GPIO module and the interrupt module for GPIO.

Right after turning off the IRQs turn on and reset both the GPIO and GPIO interrupt modules.

```
    // ------------------------ Begin GPIO setup -----------------------------------
        // Set up a general GPIO for use
        // Only the ISR needs the GPIO.
        SYSCON->SYSAHBCLKCTRL0 |= ( SYSCON_SYSAHBCLKCTRL0_GPIO0_MASK |           // GPIO is on
                                    SYSCON_SYSAHBCLKCTRL0_GPIO_INT_MASK);        // GPIO Interrupt is on

        // Put 0 in the GPIO and GPIO Interrupt reset bit to reset it.
        // Then put a 1 in the GPIO and GPIO Interrupt reset bit to allow both to operate.
        // manual: Section 6.6.10
        SYSCON->PRESETCTRL0 &= ~(SYSCON_PRESETCTRL0_GPIO0_RST_N_MASK |
                                 SYSCON_PRESETCTRL0_GPIOINT_RST_N_MASK);// reset GPIO and GPIO Interrupt   (bit=0)
        SYSCON->PRESETCTRL0 |= (SYSCON_PRESETCTRL0_GPIO0_RST_N_MASK |
                                SYSCON_PRESETCTRL0_GPIOINT_RST_N_MASK);// clear reset  (bit=1)
```

Table 11 Configure one pin to be input and one to be output.

Configure one pin to be an input (for the button) and one for output (for the LED).

```
        // Config the Pushbutton (GPIO 8) for input and LED (GPIO 9) for output
        // Remember: only bits set to 1 have an effect on DIRCLR and DIRSET registers.
        //                                  bits cleared to 0 are ignored.
        //                           Therefore, use DIRCLR to select input and DIRSET to select output
        GPIO->DIRCLR[0] = (1UL<<BUTTON_USER1);                          // input on PB8 (BUTTON_USER1)

        GPIO->CLR[0] = (1UL<<LED_USER2);                                // LED is on PB9( turn off   )
        GPIO->DIRSET[0] = (1UL<<LED_USER2);                             // output on PB9 (LED_USER2)

        // ------------------------ end of GPIO setup ------------------------------------
```

Table 12 Inform your microcontroller about edge sensitivity of the voltage signal for the button.

Configure the interrupt module so that it connects to the GPIO pin for the button, then say the interrupt mechanism is sensitive to voltage edges (rising or falling)

```
        // Set up GPIO IRQ: interrupt channel 0 (PINTSEL0) to GPIO 8
        SYSCON->PINTSEL[0] = BUTTON_USER1;   // PINTSEL0 is P0_8

        // Configure the Pin interrupt mode register (a.k.a ISEL) for edge-sensitive
        // on PINTSEL0.  0 is edge sensitive. 1 is level sensitive.
        PINT->ISEL = 0x00; // channel 0 bit is 0: is edge sensitive (so are the other channels)
                                                // Use IENR or IENF (or S/CIENF or S/CIENR) to set edge type

        // Configure Chan 0 for only falling edge detection (no rising edge detection)
        PINT->CIENR = 0b00000001;   // bit 0 is 1: disable channel 0 IRQ for rising edge
        PINT->SIENF = 0b00000001;   // bit 0 is 1: enable channel 0 IRQ for falling edge

        // Remove any pending or left-over interrupt flags
        PINT->IST = 0xFF;           // each bit set to 1 removes any pending flag.
```

Table 13 Turn on the interrupt detection mechanisms again.

Turn on interrupts (all and the one for the GPIO)

```
        // enable global interrupts & GPIO INT channel 0
        NVIC_EnableIRQ(PIN_INT0_IRQn);       // GPIO interrupt

        __enable_irq();  // global
```

Table 14 Insert the infinite loop with nothing in it.

Make an empty infinite loop and wrap up the main function.

```
    /* Enter an infinite loop, check the pushbutton on GPIO8.*/
    while(1)
    {
        asm("NOP");
    }
    return 0 ;
}
```

Now that you've written all the setup code, it's time to write the interrupt service routine. Place this before the main function.

Here, in the Interrupt Request Handler function (or Interrupt Service Routine), ISR checks the IST register of the Peripheral Interrupt (PINT) module to see what originated the interrupt request. It checks Bit 0 because we assigned GPIO to trigger interrupt requests on Channel 0. After checking Bit 0 we then have to put a 0 in the Bit 0 location of PINT's IST register. This "clears" the interrupt request. After, we invert the status of the LED using a particular bit location in the NOT register in the GPIO module. This bit location is defined by the LED_USER2 defined value.

After the "return" at the end of the ISR, the ISR returns control to whatever was running before it was called.

Table 15 An Interrupt Service routine, written in C. Place this before (above) your main function in the main C file.

Write your Interrupt Service Routine (ISR) that will automatically engage when a button is pressed. It will interrupt the main function to do so.

```
  void PIN_INT0_IRQHandler(void)
  {

          // was an IRQ requested for Channel 0 of GPIO INT?
          if (PINT->IST & (1<<0))
          {
                    // remove the any IRQ flag for Channel 0 of GPIO INT
                    PINT->IST = (1<<0);
                    // TOggle the LED
                    GPIO->NOT[0] = (1UL<<LED_USER2); //
          }
          else
          {
                      asm("NOP");  // Place a breakpt here if debugging.
          }

          return;
  }
```

If you're writing in C++, note that the ISR is slightly different… just put a wrapper around it.

Table 16 A C++ version of the same Interrupt Service Routine.

ISRs in C++ are handled a little differently than in C. You need to wrap them in an Extern C „wrapper"

```
// for C++ https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B#Linking_C_and_C.2B.2B_code
// ----------------------------------------
extern "C" {
  void PIN_INT0_IRQHandler(void)
  {

          // was an IRQ requested for Channel 0 of GPIO INT?
          if (PINT->IST & (1<<0))
          {
                    // remove the any IRQ flag for Channel 0 of GPIO INT
                    PINT->IST = (1<<0);
                    // TOggle the LED
                    GPIO->NOT[0] = (1UL<<LED_USER2); //
          }
          else
          {
                      asm("NOP");  // Place a breakpt here if debugging.
          }

          return;
  }
}
```

Part 4: Wrap-up

Reflection on Learning

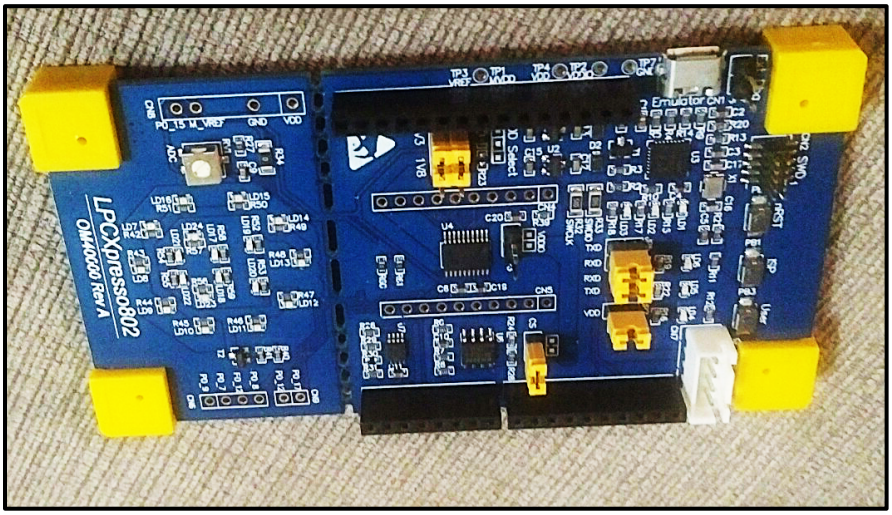
Talk to the other students in the lab. Can you see how using the debugger features allows you to better understand the inner workings of the microprocessor? Are there other ways to view memory on the chip that could be useful? If you were to start an engineering project with a microcontroller, how much would it cost you to get a stand-alone debugger like a PICKit4 or a Segger J-Link (EDU or standard)? Compare that to the standard hourly wage for an Electrical, Software or Mechatronics engineer. Does it sound like it would be worth it to use one in future projects?

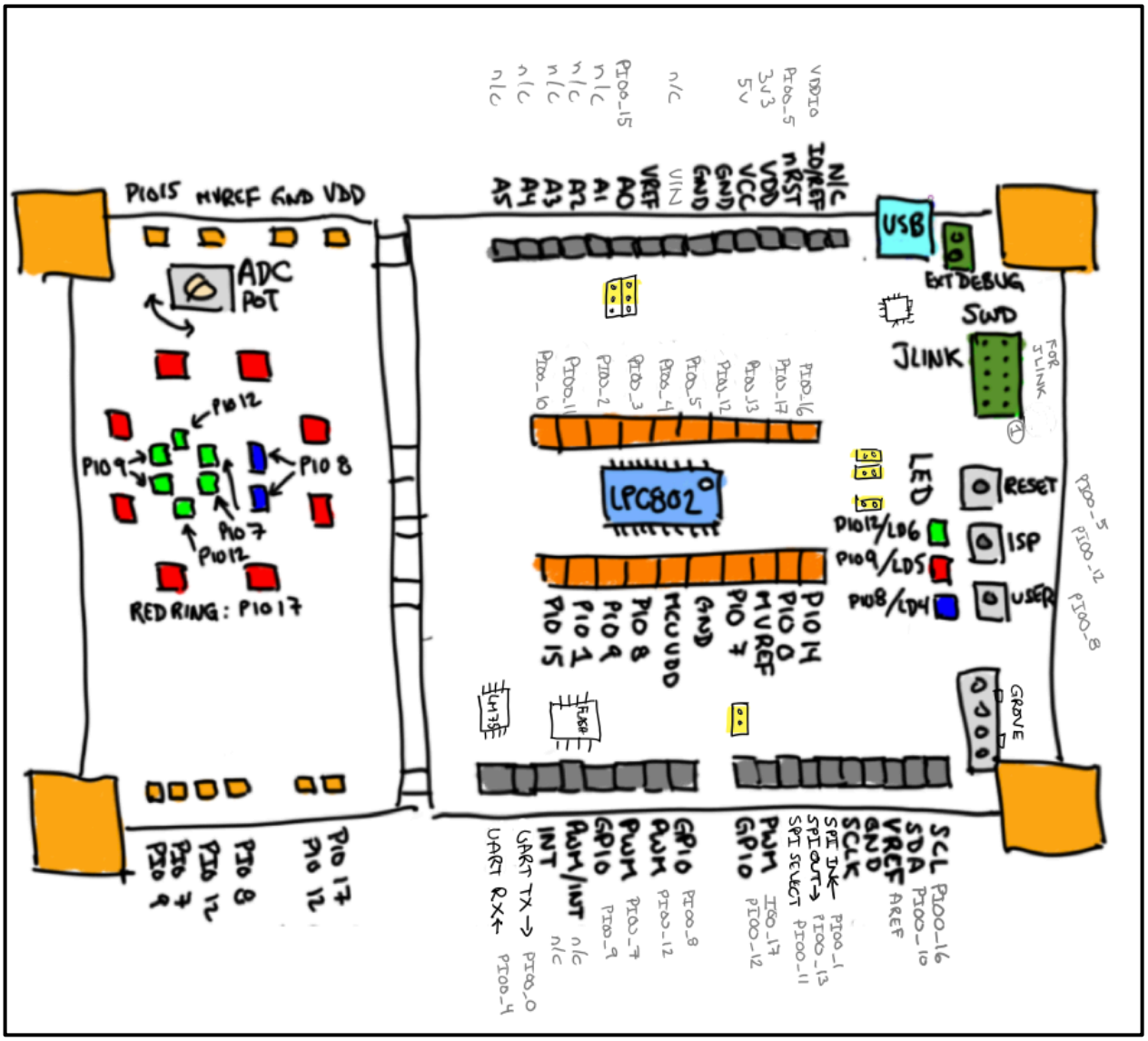
Communication – Reporting

There is no report for this lab. Simply ensure that you perform the demonstrations during the lab session.

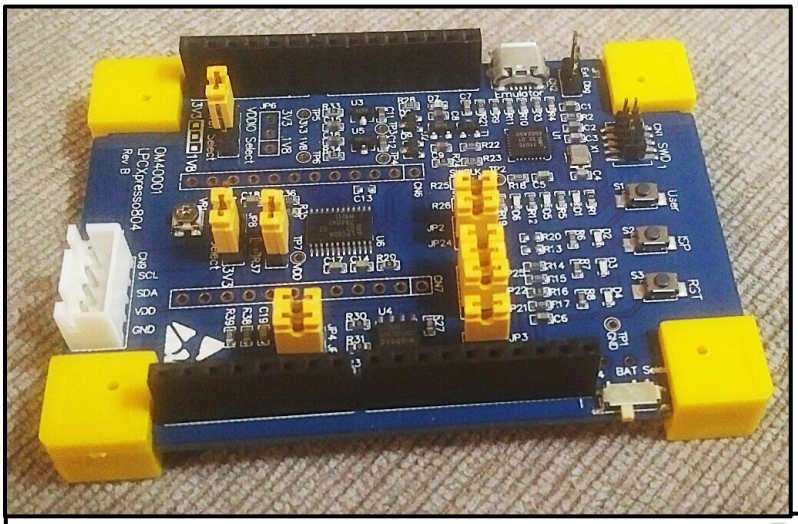
Appendix

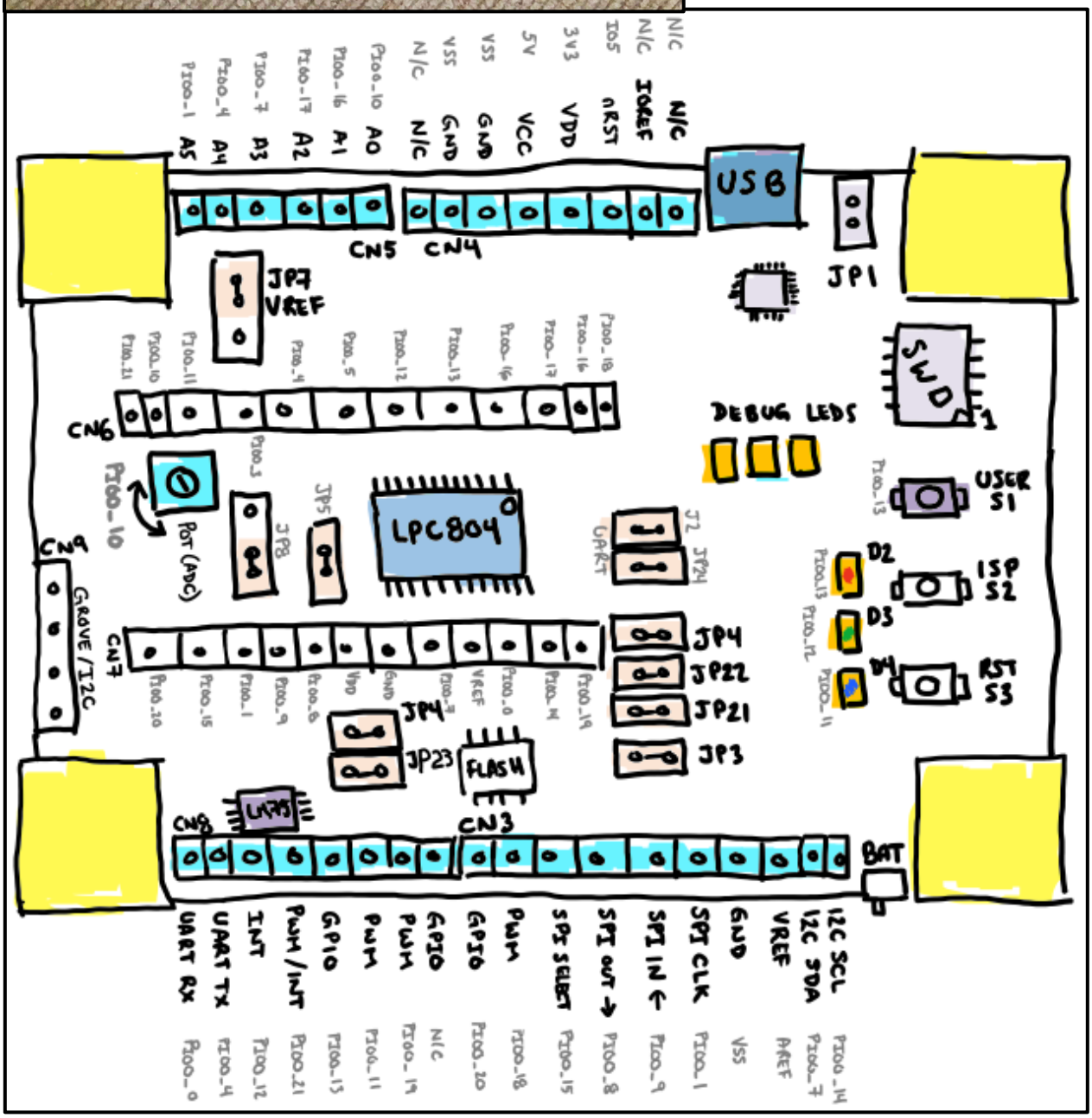
The LPC802 board (OM40000) and pinout description





The LPC804 board (OM40001) and pinout description





A note about using the LPC804 board.

In this lab you'll need to add the "capacitive touch" application board to the LPC804 (OM40001) main board. It mounts on top, using the Arduino-style headers.

The inputs and outputs used in this lab are:

1. Button on GPIO 13 (i.e. PIO0_13)
    a. This is on the main board, labelled as "USER"
2. LED on GPIO 20 (i.e. PIO0_20).
    a. This is on the capacitive touch board.

Most of the code examples given for the LPC802 will also work for the '804. Just keep in mind that you have to

1. Select the LPC804/OM40001 SDK when starting a project
2. Include the "LPC804.h" header file, and
3. Use the write GPIO assignments that correspond to buttons and LEDs on the OM40001 board

Note that the '804 chip is hidden *under* the application board in the image below.

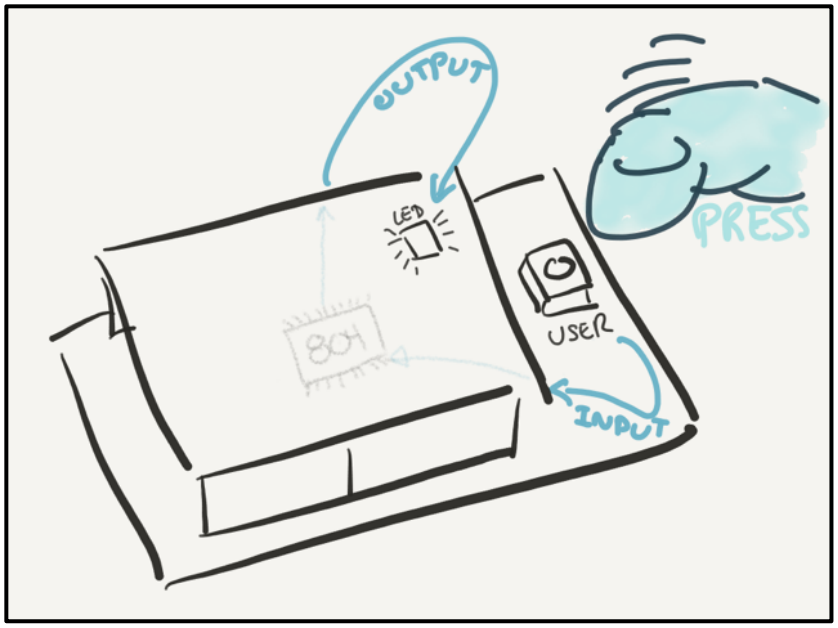


Figure 4 If you're using the '804 board you'll need to add the capacitive touch application board in order to use the LEDs in this lab.