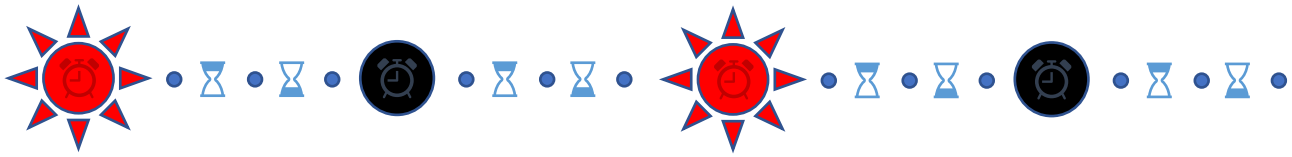
Lab F: HeartBeats (SysTick & WKT Timers)



Introduction

Single Sentence Overview

You will learn how to deal with timer hardware and time-sensitive events called Interrupts.[1]

Overview of Lab E

This three-part lab is designed to introduce the student to timers on a microcontroller. The three parts are:

1. Make a Heartbeat signal on an LED using a blocking delay.
2. Make a heartbeat signal on an LED using a non-blocking delay courtesy of the SysTick ISR.
3. Make a pair of non-blocking heartbeat signals on two LEDs using two separate timers, SysTick and WKT.

On the LPC802 board you will use

1. Two LEDs connected to GPIO 12 and GPIO 17 (PIO0_12 and PIO0_17
2. The SysTick timer (SysTick)
3. The Wakeup Timer (WKT)

LPC 804	If you are using the LPC804 (OM40001 board), you will use the capacitive touch add-on board and the LEDS located there. Use the LEDs connected to GPIO 18 (PIO0_18) and GPIO 18 (PIO0_20) . You will be using the same two timers, configured the same way as on the LPC802.

Learning Outcomes

At the end of this lab the student will be able to

1. Set up a hardware timer and to compare it to the use of manual, blocking delays.
2. Set up an interrupt service routine tied to a particular hardware timer
3. Set up a pair on interrupt service routines tied to two different hardware timers.

[1] This document is based on "Lab 3" written by James Andrew Smith for the PIC16F1619 and taught at INSA Strasbourg in Strasbourg, France.

Success Criteria

The student will demonstrate working timer solutions through the use of flashing LEDs and manual timers like a wrist watch, a stop watch, online timing page, or a cell phone clock.

Prerequisites

- Attending 3215 classes covering interrupts and timers, specifically about
  - the system timer (SysTick), as well as
  - the Wakeup Timer (WKT)
- finishing the previous two labs
  - Lab D (GPIO)
  - Lab E (buttons and interrupts)

Note that the use of C++ solutions, specifically using C++14, are considered equivalent for students who choose to use them but are not required.

Grading Rubric

- Part 1:
  - 0 if the demonstration does not work or is not attempted.
  - 1 if the first demonstration partially works.
  - 2 if the first demonstration fully works.
- Part 2:
  - 0 if the demonstration does not work or is not attempted.
  - 1 if the second demonstration partially works.
  - 2 if the second demonstration fully works.
- Part 2:
  - 0 if the demonstration does not work or is not attempted.
  - 3 if the double timer demonstration partially works.
  - 6 if the double timer demonstration fully works.

More Resources and Information

- The LPC802
  - OM40000 User Manual
  - The LPC802 (chip) user manual & data sheet
- The LPC804
  - OM40001 board schematic
  - The LPC804 (chip) user manual & data sheet

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

Background

Two of the most useful components inside your microcontroller are the "timer" and the "interruption" mechanisms. We will effectively use "SysTick" like an alarm clock that can set off an alarm signal on a regular basis. We will let this alarm to interrupt the usual operation in the main loop of the microcontroller and to flash an LED with the same frequency as the timer alarms.

It is possible to use other timers on the LPC802, as well as other sources of data to interrupt the LPC802's functioning. However, SysTick is one of the most straight-forward ways to learn about both timer operations and interrupt mechanisms.

In Problem 2 of this lab you will contrast the creation of a "heartbeat" signal that uses a simple approximate delay versus a better "heartbeat" that combines the use of a Timer and an Interrupt Service Routine.

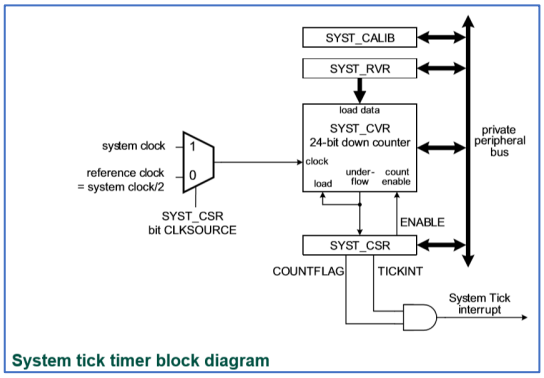
In Problem 3 of this lab you'll create two parallel heartbeats: one with the SysTick and one with the Wakeup Timer.

Timers

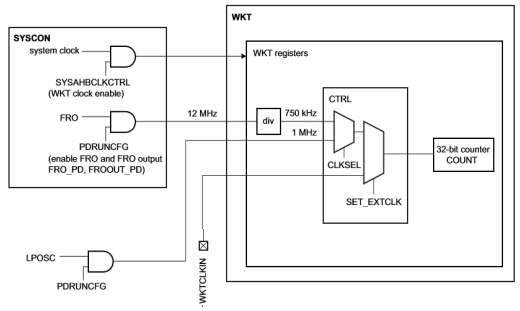
When you need to know time, with a watch or cell phone, you either look at the display on demand or you set an alarm to warn you at a pre-arranged time. SysTick on the LPC802 works the same way. You can either request to know its time or you can have it set off an alarm at a regular interval. We will use SysTick's "overflow alarm" setting here. The 24-bit counter inside SysTick counts down from a very large number, down to 0 at a specific frequency. When it reaches 0 it sets off an alarm that can make an Interrupt Service Request and then starts counting again. The frequency at which these alarms go off is set by specifying a source clock and then, optionally, setting a prescale value to reduce the frequency.

It's important to point out that the System Clock (FRO), by default out of reset, is *internal* to the chip and is set to 12 MHz. You can adjust the FRO to be up to 15 MHz *or* down to 9 MHz, depending on your application. However, for this lab, 12 MHz is fine.

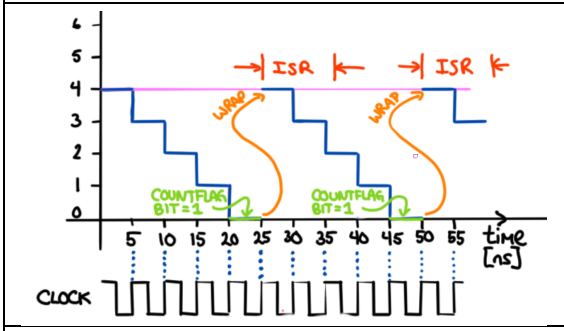
There are multiple timers on the LPC802, including the SysTick and the Wakeup Timer. You'll use the SysTick in Problem 2. You'll use both the SysTick and WKT in Problem 3. The block diagrams and signal progression charts are shown for both, below.



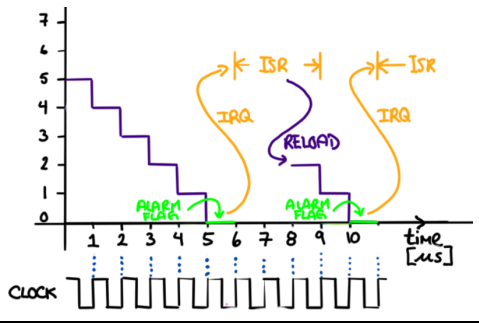
Block diagram for the SysTick (Source: NXP User Manual: UM11045)



Block diagram for the Wakeup Timer (Source: NXP User Manual: UM11045)



Example count down and interrupt request call for the System Timer.



Example count down and interrupt request call for the Wakeup Timer.

Figure 1 Functional block diagrams and example sequences for the SysTick and Wakeup Timer.

Note that we call these devices **both** Timers and Counters. That's because these devices count the progression of time. The count value found inside them represent units of time. What unit? It depends on the clocking signal that feeds the Timer/Counter. If the clocking signal is high frequency then the change in count inside the Timer/Counter will be a proportionally small amount of time. If the frequency feeding the Timer/Counter is slow then the changes in Timer/Counter count values will be greater in absolute values of time.

Interrupt like an Alarm Clock

It is important to be able to stop a task on the microcontroller when something more important needs to be dealt with. This is referred to as an "interrupt." The interruption can be caused by internal events (like a timer alarm) or external events (like an emergency button press in a factory). When an interrupt occurs on the LPC802, the microcontroller stops what it was doing and goes to a special, pre-defined memory location to find instructions on how to proceed. It is your job to write the "interrupt service routine" (ISR) that this process engages. In the case of this lab you will be looking at how the SysTick can *make a request* for an interrupt. This is similar to how an alarm clock can be set up to tell you when it's time to wake up in the morning.

A note about weird symbols and words (repeat from LabE)

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.

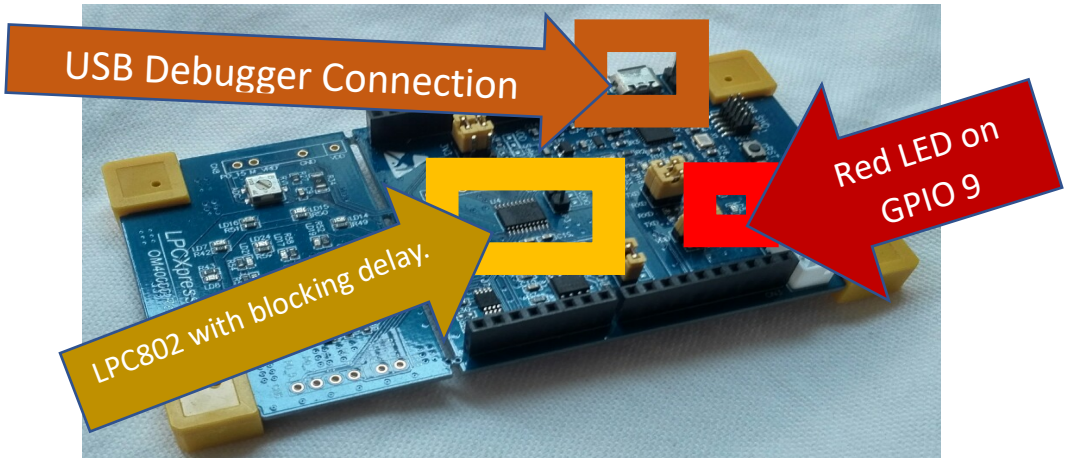
Problem 1: Make a Bad Heartbeat (with a Blocking Delay)

To appreciate a good, interrupt-based timer, you need to make its opposite: a bad "blocking" timer. You'll use this bad timer to flash an LED on and off, imitating a heartbeat on your microcontroller board.

Create a loop in your main function that turns an LED on and off in a repeatable way. An example of an infinite loop is shown in Figure 4 Basic layout of a typical C program in embedded applications.. This method doesn't use interrupts, because verifying that your I/O hardware is working helps reduce possible sources of bugs when you've engaged the interruption mechanism.

```
/* my c file */

// put #pragma and #define statements here

// put #include statements here

// Need to define an interrupt service routine?  You can do that here.

// main function
int void main(void)
{
    // need a variable?  Define it here.

    // initialize registers here

    while(1)        // beginning of while loop.
    {

        // write boring microcontroller jobs here
        // to be done from now until the end of time.


    }               // end of while loop.


    return 0;
}
```

Figure 4 Basic layout of a typical C program in embedded applications. You'll need to customize it for your particular microcontroller, generally by specifying a header file that contains defined memory locations, etc.



Inside your while loop you should **insert a For Loop** that counts from 0 to a really big number (like 500,000 or 3,000,000). Once that For Loop is done, you should toggle the LED using

```
GPIO->NOT[0]    = ..... (what should go here??)
```

Make the LED flash once per second or once over two or three seconds.

Demonstrate to the lab instructor or teaching assistant that you've got the heartbeat working.

LPC 804	If you are using the LPC804 (OM40001 board), you will use LED on the capacitive touch add-on board connected to GPIO 18 (PIO0_18).

Stimulate[2]

Delays are easy to create on your LPC802. The easiest ones use loops and are referred to as "blocking." Why is a blocking loop a bad idea? Refer to https://bit.ly/2OKCtBf for discussion using Arduinos.

Explore

Does your microcontroller IDE (compiler) come with a built-in command for a manual, blocking delay? If not, look up what Microchip's XC8 compiler (used with PIC16 and PIC18 chips) has.

[2] Stimulate and Explore sections inspired by Fred Cady's course material for NXP/Freescale's 9s12 processor. Thanks Fred!

Problem 2: The Timing of a Heartbeat (SysTick)

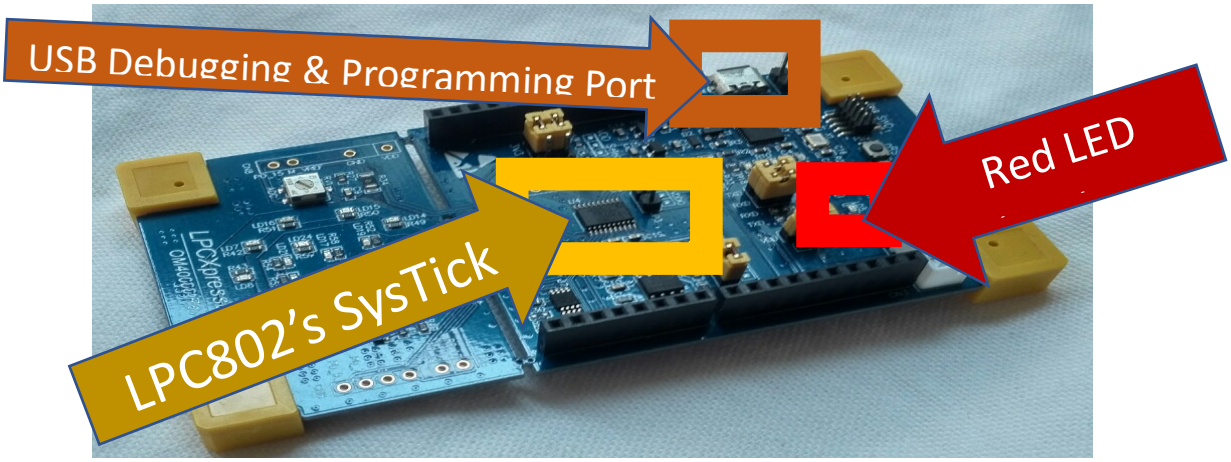
Single Sentence Overview

Create a visible "all is working" heartbeat on the microcontroller board that **works independently** of the main function.

Background

Two of the most useful components inside your microcontroller are the "timer" and the "interruption" mechanisms. We will effectively use "SysTick" like an alarm clock that can set off an alarm signal on a regular basis. We will let this alarm to interrupt the usual operation in the main loop of the microcontroller and to flash an LED with the same frequency as the timer alarms.

It is possible to use other timers on the LPC802, as well as other sources of data to interrupt the LPC802's functioning. However, SysTick is one of the most straight-forward ways to learn about both timer operations and interrupt mechanisms.



In Problem 2 of this lab you will contrast the creation of a "heartbeat" signal that uses a simple approximate delay versus a better "heartbeat" that combines the use of a Timer and an Interrupt Service Routine.

Use the notes from class to write up the routine for setting up the SysTick, for establishing an interrupt service routine and for connecting that to an LED.

Demonstrate to the lab instructor or teaching assistant that you've got the SysTick-based heartbeat working.

LPC 804	If you are using the LPC804 (OM40001 board), you will use LED on the capacitive touch add-on board connected to GPIO 18 (PIO0_18).

Problem 3: Run the SysTick LED heartbeat at 1 Hz and the WKT LED heartbeat at 2 Hz

Now it's time to combine things. The reason? Because, often, you want to run tasks in parallel or near-parallel on a microcontroller. This is another step in that direction.

Initialise both the SysTick and WKT timers so that they are interrupt enabled. Use the lower power oscillator as your source on the WKT and use the FRO as your source on SysTick. Use the following LEDs

	LPC**802**	*Note for LPC802*	LPC**804**	*Note for LPC804*
SysTick Heartbeat at 1 Hz	PIO0_17	*Red circle of LEDs on left side of OM4000 board*	PIO0_20	*D1 LED on Capacitive Touch peripheral board*
WKT Heartbeat at 3 Hz	PIO0_12	*Pair of green LEDs on left side of the OM4000 board.*	PIO0_18	*D2 LED on Capacitive peripheral board*

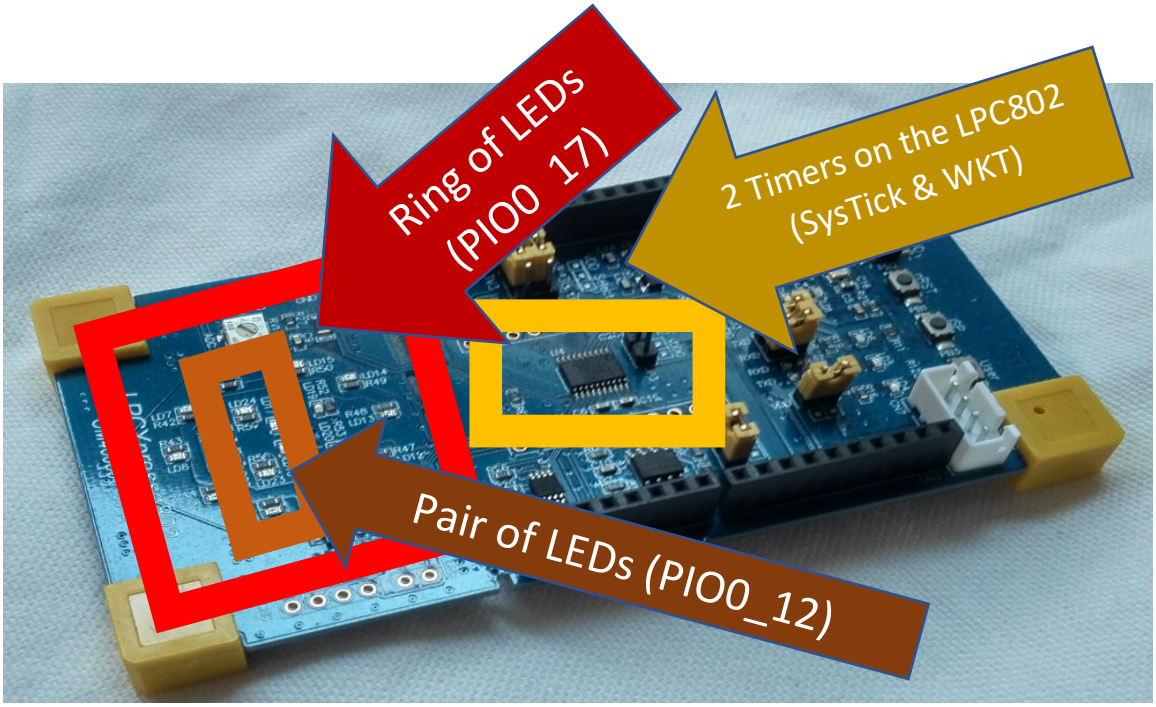


Figure 5 To indicate functioning of the microcontroller's timers you should output to LEDs. PIO0_12 and PIO0_17 connect to LEDs on the left hand side of the board.

Initialize the two timers, as well as the GPIO pins that connect to the LEDs. Timer initialization and GPIO setup have been discussed in class. Note that you will need to have two separate interrupt service routines, one for the interruption signals from SysTick and another one for interruption signals generated by the Wakeup Timer (WKT).

Demonstrate these two LEDs flashing using non-blocking interrupt service routines tied to the SysTick and Wakeup Timers to the teaching assistants.

Explore

Reflect on the following questions. Discuss them with your lab partner in the lab or in a coffee shop.

1. By default, we use the timers to drive the LED on half the time and off half the time. This is referred to as "Fifty Percent Duty Cycle" on a "Pulse Width Modulated" signal. How would you make the LED associated with the WKT interrupt service routine run the LED on 10% of the time and off 90% of the time (i.e. 10% duty cycle PWM)? How about 75% duty cycle?
2. In the event that the System Timer and the Wakeup Timer generate Interrupt Service Routines at the same, which one has the higher priority? What happens in the Wakeup Timer is servicing its ISR and the SysTick interrupt request is detected? What happens in reverse case?

Part 4: Wrap-up

Reflection on Learning

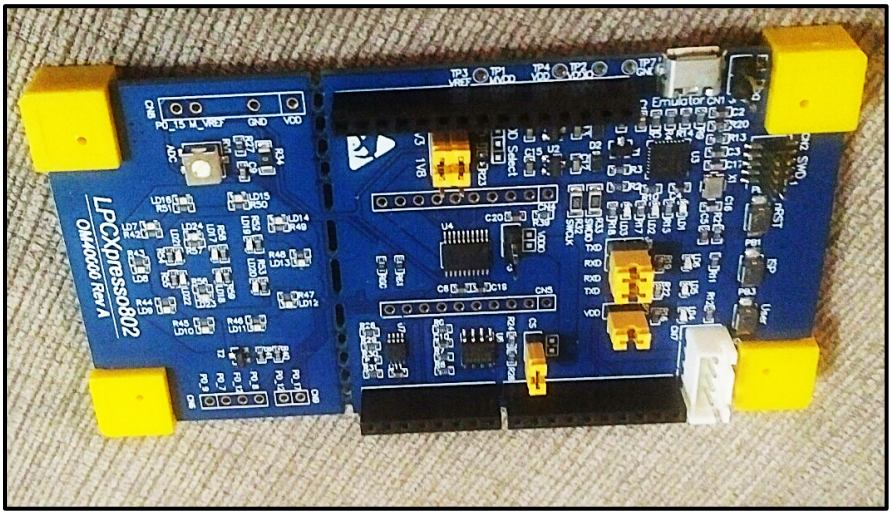
In order to get an LED or other digital output device to react to the alarms generated by timers we need to have the processor stop what it was doing and then service the interrupt request. That takes time. It's a small amount of time, but it can be significant in certain situations. Imagine if the ISR wasn't required and the timer could access the microcontroller's pins directly without the need to use a GPIO register. Maybe, in a future lab, we'll cover something like that… maybe… imagine the possibilities!

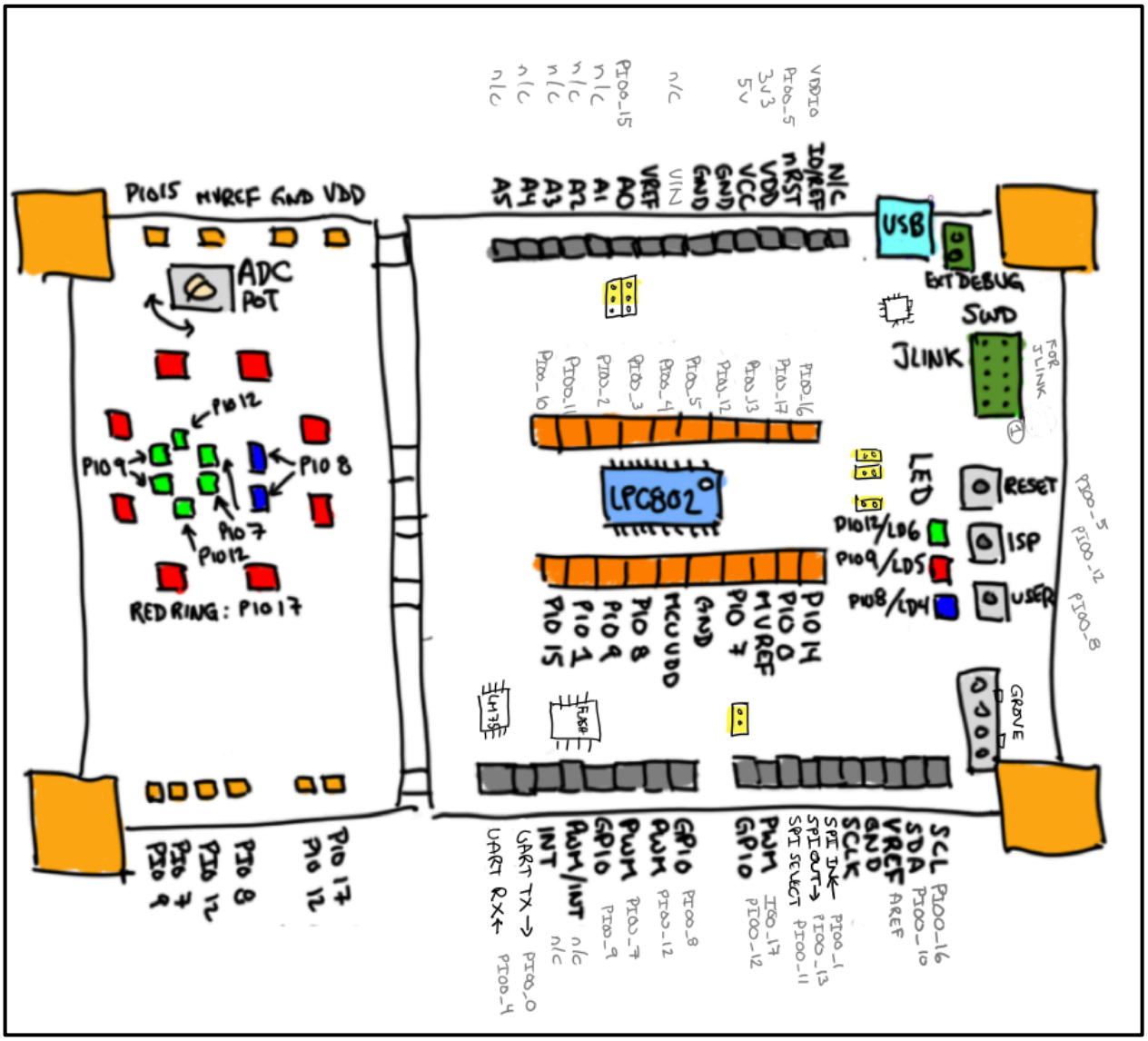
Communication – Reporting

There is no report. Read the grading rubric at the beginning of this document. Make sure that you demonstrate your working programs as discussed before leaving the lab.

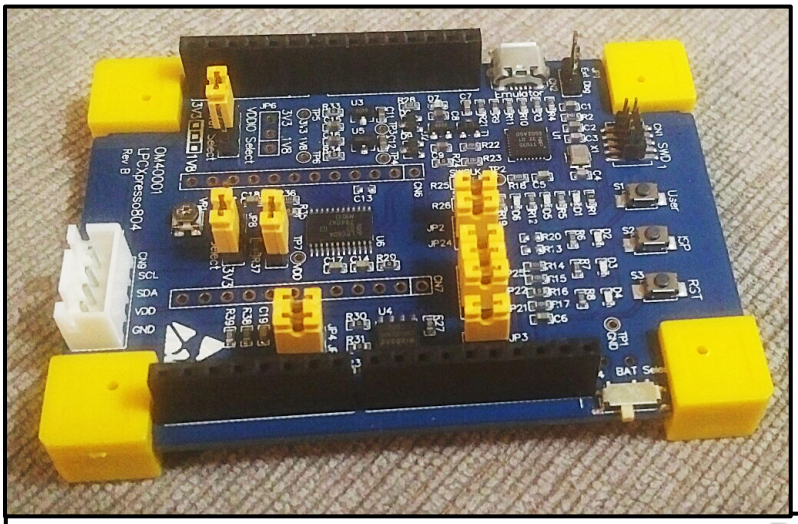
Appendix

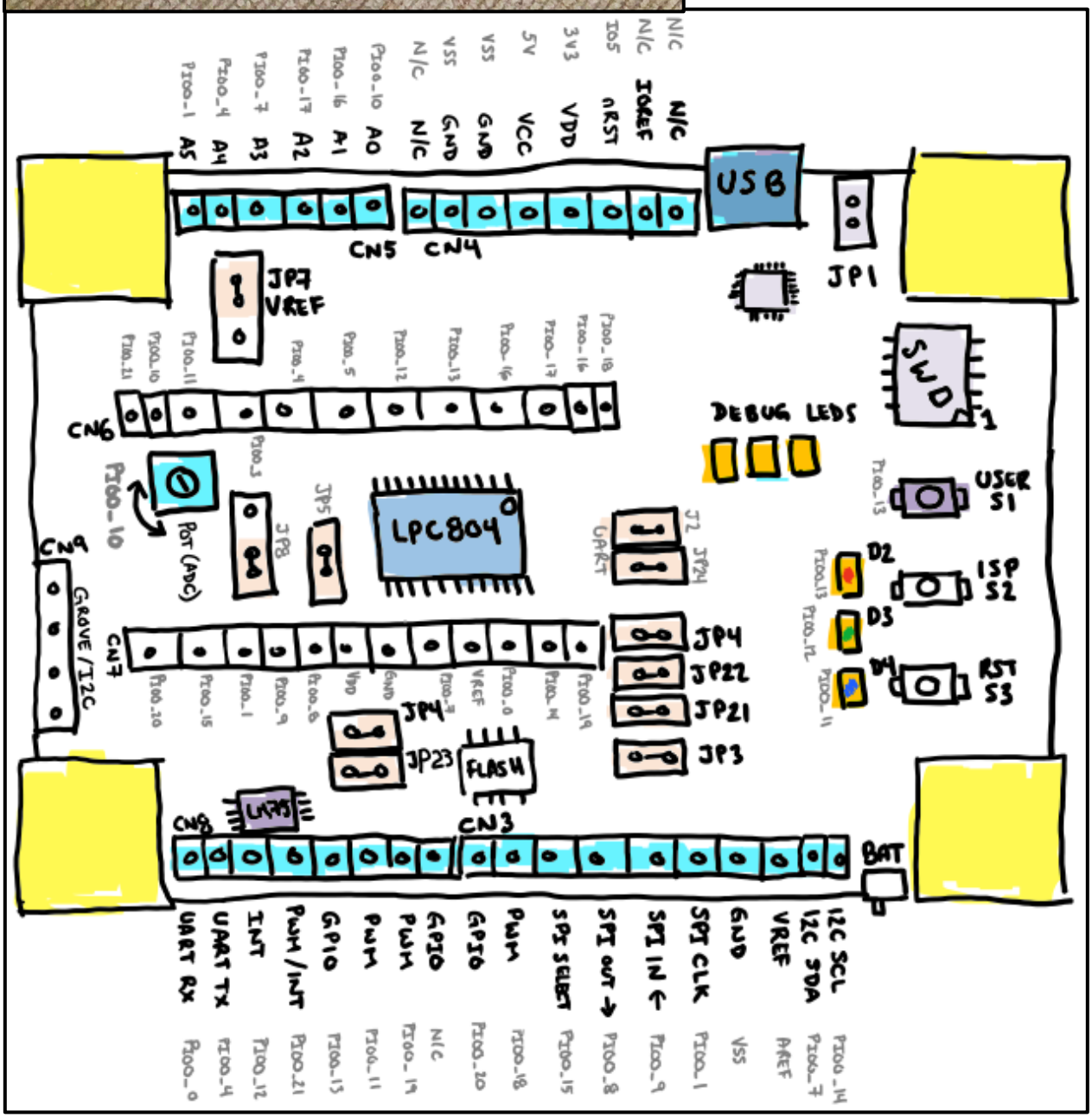
The LPC802 board (OM40000) and pinout description





The LPC804 board (OM40001) and pinout description





A note about using the LPC804 board.

Some of you are using the LPC804. For those of you using the LPC804, 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.

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