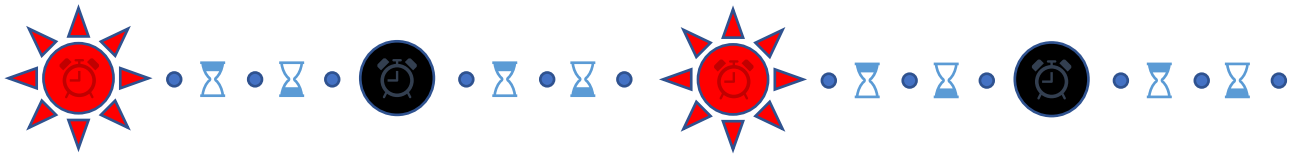
Lab G: One-Shots & Pulse Width with MRT



Introduction

Single Sentence Overview

You will learn how to deal with another timer module, the Multi-Rate Timer (MRT).[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. One LED one-shot
2. One LED PWM (slow)
3. Two LED PWM (bright vs. dim)

Learning Outcomes

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

1. Set up an interrupt service routine tied to a particular hardware timer
2. Be able to implement one-shot and "soft" PWM operation using a timer
3. Vary frequency and pulse width modulation using timers

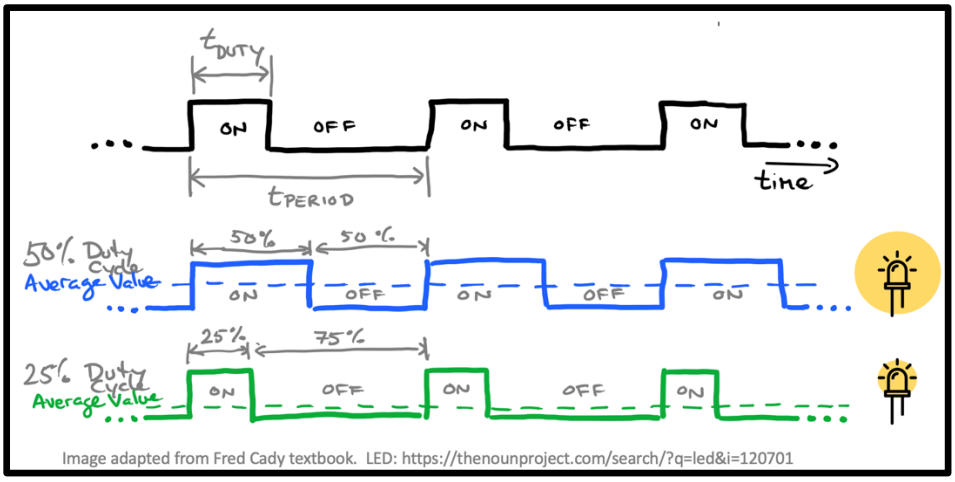


Figure 1 Pulse Width Modulation can be used to brighten or dim LEDs, as well as to create sounds in speakers or to control speed in an electric motor.

[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 multi-rate timer (MRT)
- finishing the previous three previous labs
  - Lab D (GPIO)
  - Lab E (buttons and interrupts)
  - Lab F (SysTick and Wakeup Timer.

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: (one shot)
  - 0 if the demonstration does not work or is not attempted.
  - 1 if the first demonstration partially works.
  - 3 if the first demonstration fully works.
- Part 2: (slow PWM)
  - 0 if the demonstration does not work or is not attempted.
  - 1 if the second demonstration partially works.
  - 3 if the second demonstration fully works.
- Part 3: (double PWM)
  - 0 if the demonstration does not work or is not attempted.
  - 2 if the double timer demonstration partially works.
  - 4 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

We've discussed, in previous labs the importance of "timer" and the "interruption" mechanisms in all contemporary microcontrollers. The System Timer and Wakeup Timers are both examples of hardware modules in your microcontroller that bring these concepts together. We will effectively use these timers like alarm clocks that can set off an alarm signal on a regular basis. We let these alarm interrupt the usual operation in the main loop of the microcontroller and flash an LED with the same frequency as the timer alarms.

While we started with the simplest timer, the SysTick and we continue our journey towards the more complex timer, the CTimer, we will work on a moderate complexity timer, the Multi-Rate Timer (MRT) in today's lab.

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. Multi-Rate Timer (MRT) 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 MRT's "overflow alarm" setting here. The 31-bit counter inside the MRT counts down from a very large number, down to 0 at a specific frequency. Unlike the SysTick, which is really designed with firing off its alarm over and over and to only really do that, the MRT has three possible applications that it can engage at the end of a count-down:

1. Nothing else ("one-shot")
2. Doing it again ("PWM"), and
3. "bus halting" (which we won't deal with)

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 a count-down value. Unlike other timers, the MRT does not have a pre-scale value on its source clock. We will assume that you can only use the FRO as your source clock, meaning that there are three possible frequencies at which you can run the MRT's counter.

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.

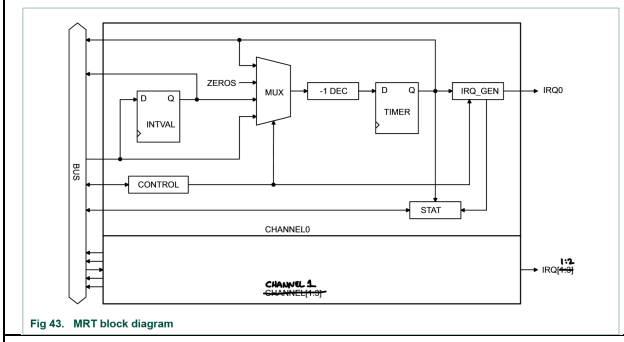


Fig 43. MRT block diagram

Block diagram for the MRT (Source: NXP User Manual: UM11045). Note the typo that has been fixed here: the LPC802 only has two independent counters, "Channel 0" and "Channel 1" in the MRT.

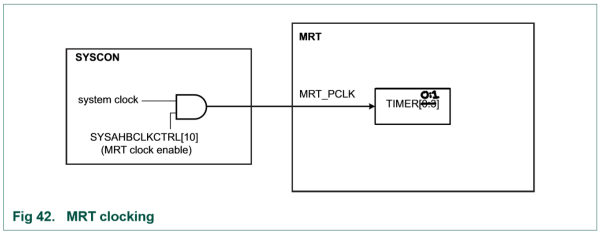
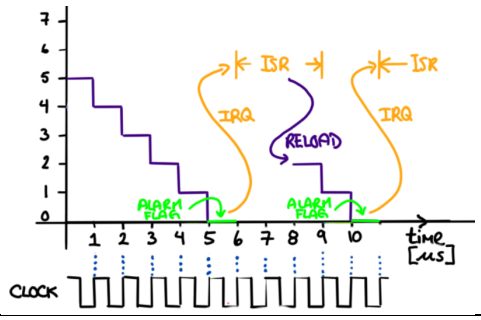


Fig 42. MRT clocking

Block diagram for the MRT (Source: NXP User Manual: UM11045). Note the typo that has been fixed here: the LPC802 only has two independent counters, "Channel 0" and "Channel 1" in the MRT.



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

Figure 2 Functional block diagrams and example sequences for the Multi-Rate timer (MRT).

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.

One Shot Interruptions

Like the Wakeup Timer, the Multi-Rate Timer is capable of one-shot operation. Basically, set up a count down from a large number and trigger a single event at a precise time. Think of it like the count down to a rocket launch.

The difference between the Wakeup Timer and the MRT's one-shot mode is that the MRT can fire two IRQs, one after the other.

Problem 1: One shot to 24 seconds

In class we discussed how to set up two separate count-down sequences using the MRT. Each of the count-down sequences would begin by driving a GPIO output down to 0 volts (Logic 0) and then, at the end of the count-down, would raise the GPIO output to 3.3 volts (Logic 1). In the figure below you can see this process. The whole MRT is clocked using the FRO oscillator, set to 12 MHz, which means that the two MRT counters, one in Channel 0 and another in Channel 1, would increment their counts once every 83.33 nanoseconds.

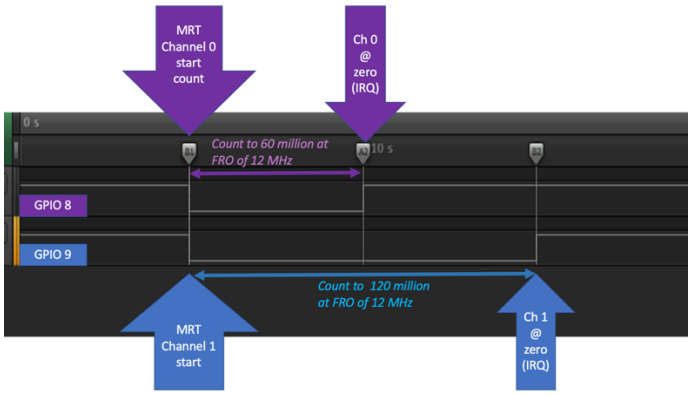


Figure 3 Illustrating of using two channels in the MRT to control two separate one-shot countdowns (first ends after five seconds; second ends after ten seconds). In this problem, do a single one-shot to 24 seconds.

The programming solution that was presented in class to make this pair of signal sequences happen required that the MRT interrupt service routine be called twice: once after five seconds to service the GPIO 8 output and another time at 10 seconds to service the GPIO 9 output. Both requests for interrupt call the same interrupt service routine and it is within the ISR that the distinction is made between which channel was responsible for the request and which GPIO needs to be updated.

For Problem 1 please *modify* the source code shown in class to accomplish the following:

1. Assign output to only GPIO 8 (on the LPC802)
2. Only use Channel 0 for counting
3. Write an interrupt service routine for the MRT that checks to see whether Channel 0 or Channel 1 requested the interruption.
4. The GPIO should be inverted or changed in a way that, when the ISR runs, it turns off the LED
5. In your main function, at the beginning of your routine, make the GPIO value Logic 1 in order to turn off the LED attached to it.
6. Later in the main function, begin the 24 second countdown and then enter an infinite while loop.

Have the TA come over and **press the reset button** on your board. Your code will restart and the TA will measure the amount of **time it takes for the LED to turn off**. It should take **24** +/- 0.5 **seconds** for full marks on this problem.

LPC 804	If you are using the LPC804 (OM40001 board), you will use LED on the capacitive touch add-on board connected to GPIO 20 (PIO0_20).

Problem 2: Slow PWM @ 1 Hz with 10% Duty Cycle

While many applications that use PWM call for the PWM to be at moderate (1 kHz to 100 kHz) frequencies these can be hard to test and debug without oscilloscopes or logic analyzers. Most of the time, to get the concepts implemented right you just need a slow signal (0.01 to 10 Hz) so that changes are visible to the naked eye.

Here, modify the PWM code shown in class so that the LED on your LPC802 flashes at a rate of once per second (1 Hz) and displays a 10% duty factor. That is, it remains on for a brief 0.100 seconds and then turns off for 0.900 seconds before starting the process all over again.

Note that you'll have to remember that that GPIO output set to Logic 1 means that the LED is off and vice versa.

Have the TA come over, press the reset button on your board and see that the ratio of on time to off time in a 1 Hz series of pulses (or "pulse train") is correct. That means using a stop watch or other accurate clock (like the NIST web server) to measure 30 pulses … and to **verify** that **30 pulses occurred in 30 seconds**.

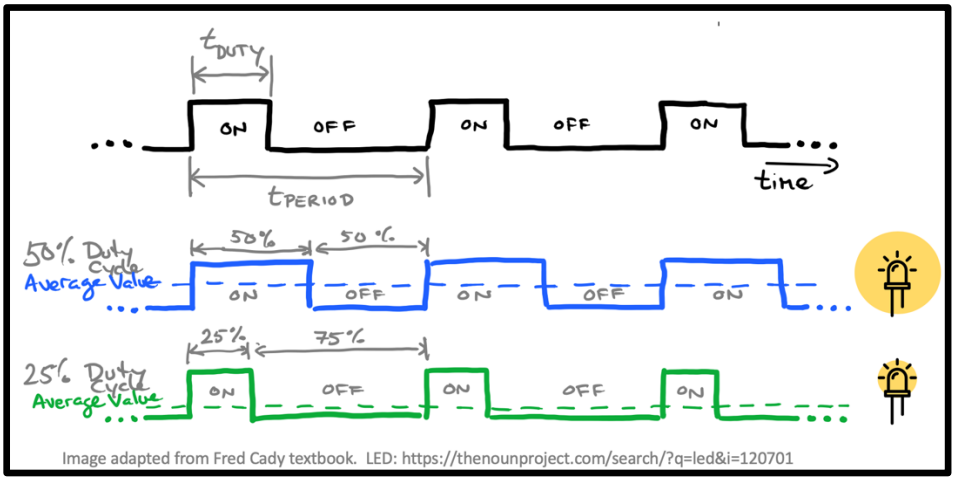


Figure 4 Pulse Width Modulation can be used to brighten or dim LEDs, as well as to create sounds in speakers or to control speed in an electric motor.

LPC 804	If you are using the LPC804 (OM40001 board), you will use LED on the capacitive touch add-on board connected to GPIO 20 (PIO0_20).

Problem 3: Two LEDs on fast PWM: one dim, one bright @ 100 Hz

Now, modify the solution to Problem 2 such that there are now two LEDs being pulse-width modulated. Both should have a frequency of 100 Hz (or at least, a frequency so that the naked eye can't distinguish between "on" and "off" time). One should be flashing with a duty cycle of 5% and the other should be flashing with a 95% duty cycle. The effects should be that one LED will be really, really dim and the other will be quite the opposite: very bright.

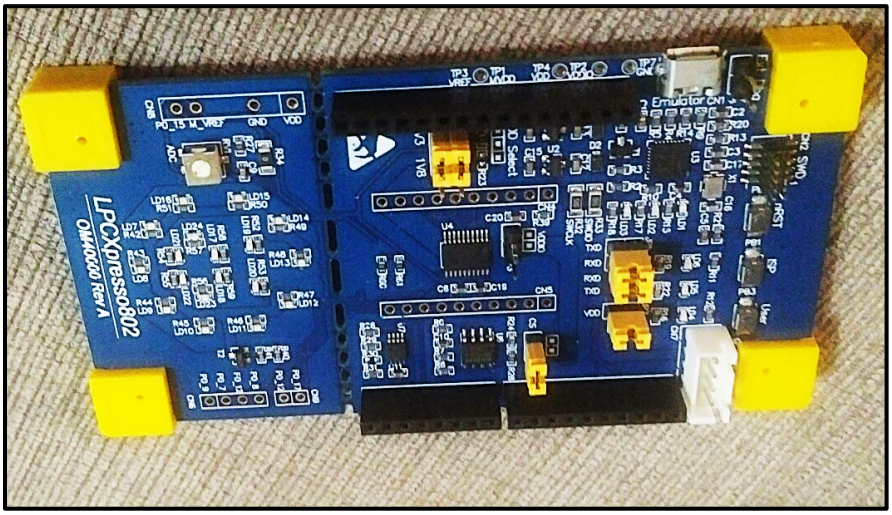
LPC 804	If you are using the LPC804 (OM40001 board), you will use LEDs on the capacitive touch add-on board connected to GPIO 20 (PIO0_20) and GPIO 18 (PIO0_18). Note that you may need to set one LED to 99% duty cycle and the other to 1% to see the difference in brightness with the naked eye.

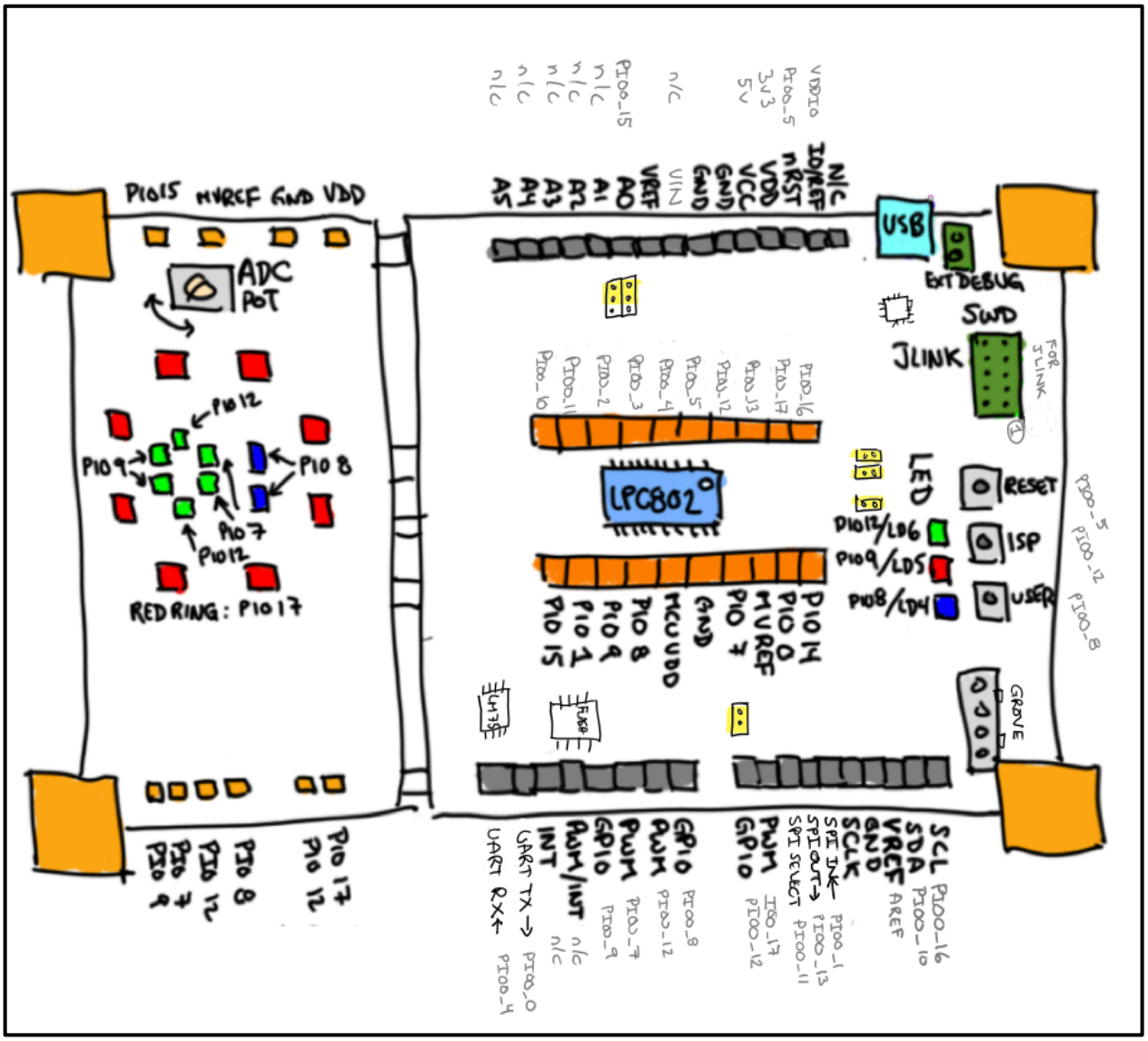
Communication – Reporting

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

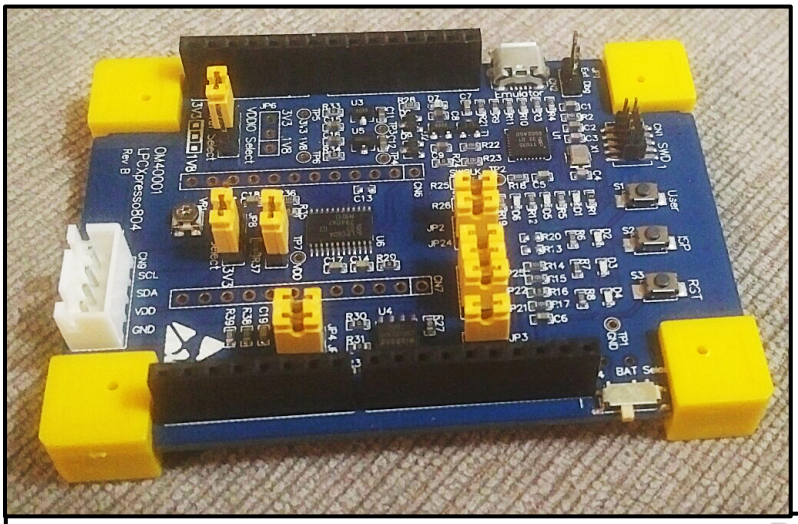
Appendix

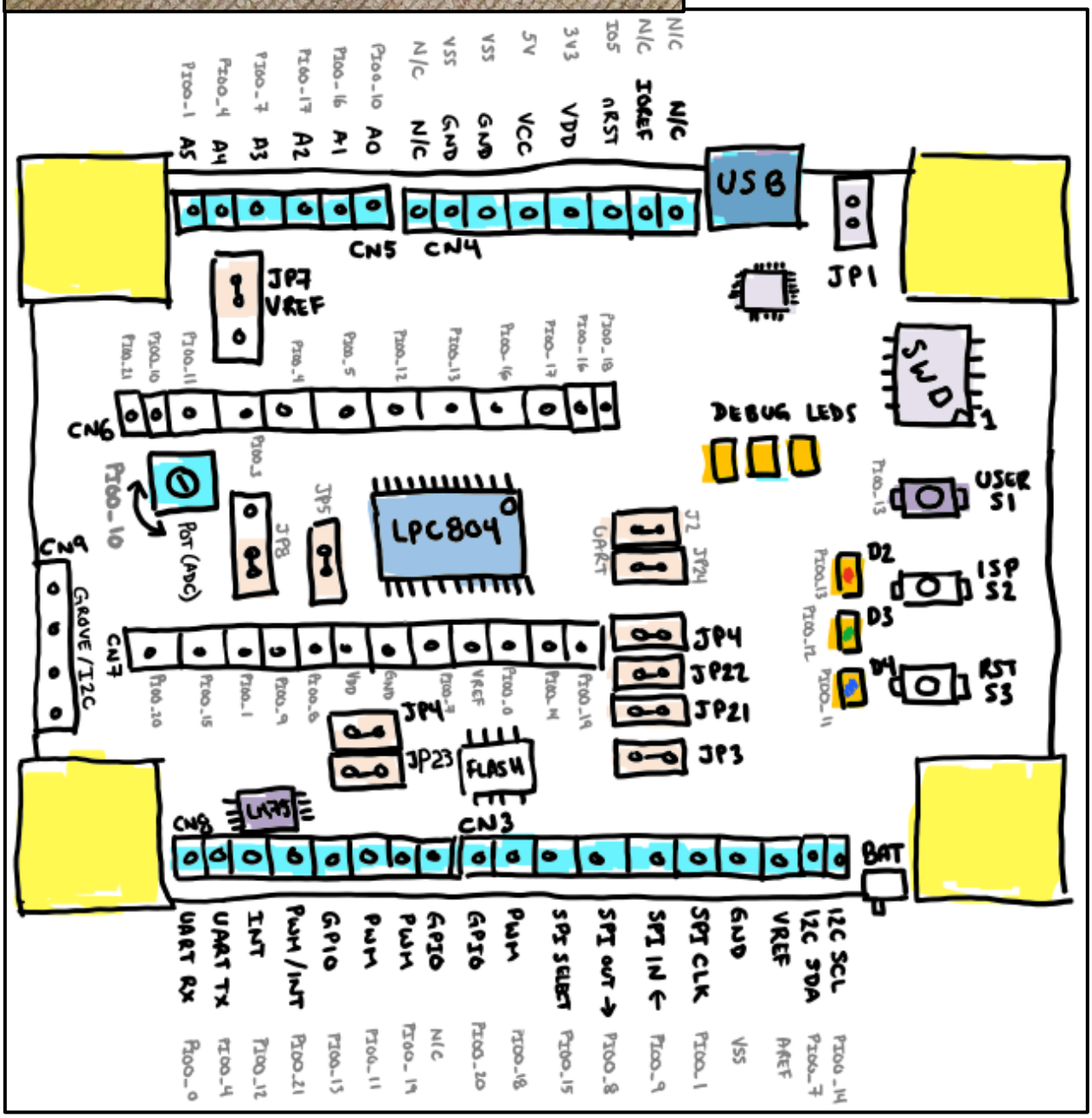
The LPC802 board (OM40000) and pinout description





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.

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.

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 5 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 6 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