EECS 3215 – Lab H: 7-segment displays and CTimer

Dr. James Andrew Smith, PEng

Overview: A microcontroller is often connected to external **display devices** to provide information to the developer or the user. The simplest, non-trivial displays that are used are 7-segment LED displays. You'll wire up and illustrate the functioning of a four digit 7-segment display in this lab. Separately, you'll demonstrate pulse-width modulion using the CTimer peripheral.[1]

Learning Objectives: This module will help you learn generally about 7 Segment LEDs, and how to design the connections to it on a breadboard. You will also learn how to configure the Switch Matrix to directly connect an internal timer peripheral to the exterior pins of the microcontroller.

Success Criteria: When you have completed this module you will be able to demonstrate that you can display information on an external LED-based 7 segment display. Specifically, that you can

1. Turn on each of the 7 segments on the display
2. Display the last four digits of your student ID on the display, one at a time.
3. That you can pulse-width modulate a single LED using the CTimer

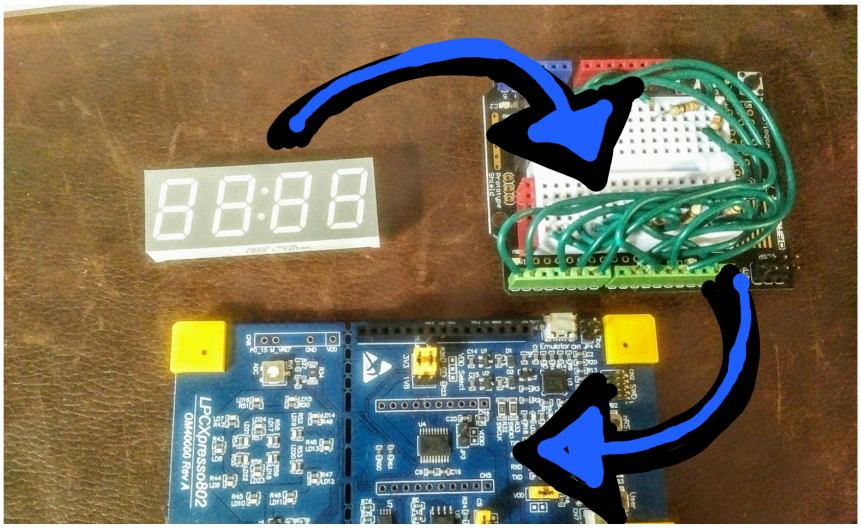


Figure 1 How do you connect the 7 segment display to LPC802 board? You'll find out!

Prerequisites: You must be able to write assembly or C programs for your microcontroller.

[1] This lab is based on the Freescale University Program Laboratory Short Course LABS12DINTRO12 /REV 1 by Fred Cady, Natasha Kholgade and Ken Hsu (RIT)

Important Resources:

To be able to wire up the shield and the seven segment display you'll need to pay attention to the datasheets and schematics. Have a look here:

1. The schematic for your board (OM40000)
   a. https://www.nxp.com/downloads/en/schematics/SCH_LPCXpresso802_Ver_A.zip
2. The datasheet / page for your shield:
   a. https://www.dfrobot.com/product-55.html
3. Display-specific:
   a. The datasheet for the **Broadcom common-anode 7-segment display** is found here: https://bit.ly/36mD0Cp and https://bit.ly/2I8ItST
   b. The datasheet on the **ATA3492** 7-Segment Display (http://bit.ly/2oVCnfu & http://bit.ly/2FpWZ9H)
4. A video illustrating how to connect the *ATA3492 display to a KL43Z board* is found here: https://youtu.be/JKZ6nLniLMQ
   a. The above video is *not an exact match to the LPC802* and your display but it shows the process for identifying locations for the resistors and the connections from the breadboard to the headers on the shield. Combine this with the table of connections and you'll be set.

LPC 804	Note: if you are using the LPC804 (OM40001 board), here's the datasheet page : https://bit.ly/39hrHgk And the schematic: https://bit.ly/2PJdCAu

Marking Guide:

- *Part 0: Wiring up the 7-segment display to the shield: 2 marks*
  - *0 for no attempt*
  - *1 for attempting*
  - *2 for complete wiring (visual inspection only.)*
- *Part 1: All segments lit up: 2 marks*
  - *0 for no attempt*
  - *1 for attempting but unsuccessful;*
  - *2 for all LEDs in 7-segment glowing*
- *Part 2: Student ID displaying (last four digits)*
  - *0 for no attempt*
  - *1 for attempt but incomplete*
  - *2 for all four digits displaying, one at a time*
- *Part 3: CTimer demonstrations*
  - *0 for no attempt*
  - *1 for attempt but no success at making either or both CTimer demos work*
  - *2 for sucessfully getting either CTimer demos to work*
  - *4 for for successfully getting both CTimer demos to work*

7-Segment Displays

LEDs are the easiest, most straight-forward "displays" to use on your microcontroller. All you need is

1. current-limiting resistor,
2. a LED,
3. a GPIO port pin, and
4. ground

to use them. You calculate the value of the resistor based on the following equation:

$$Resistor = \frac{(GPIO\ Pin\ Voltage - LED\ Forward\ Voltage)}{LED\ Constant\ Current} = \frac{3 - 2\ Volts}{0.02\ Amperes} = 50\ Ohms.$$

In general, if you use a 100 to 300 Ohm Resistor, you'll be fine.

If your LED is connected with GPIO as shown in Figure 2 (top) then the following statement in your C source file can be used to turn it on

```
// Turn on the LED
// (LED_1 is pre-defined earlier in the .c file.)
// Assuming GPIO has been configured correctly ahead of time.
GPIO->SET[0] = (1UL<<LED_1) ;
```

And the following statement can be used to turn it off

```
// Turn off the LED
GPIO->CLR[0] = (1UL<<LED_1) ;
```

From this, it's relatively straight forward to get a 7-segment display running because a 7-segment display is just a bunch of LEDs arranged together in a package, as shown in Figure 2 (bottom).

It's important to note two things about 7-segment displays. First, that they have different numbers of digits on them – some have one digit, some have two, and others have four. Second, they come in different flavours: common anode and common cathode. This means that the individual digits are tied together using either the anodes or the cathodes. I've recommended that you purchase a common anode 7-segment.

Finally, when you display things on the four digits, you tend to want to display one digit at a time and to use the four GPIO pins connected to the digit pins on the display to control which one is on at a given point in time. If you alternate very quickly from one digit to the next your user won't be able to tell that only one digit is on at once (that's the reason for doing Part 3 of the lab).

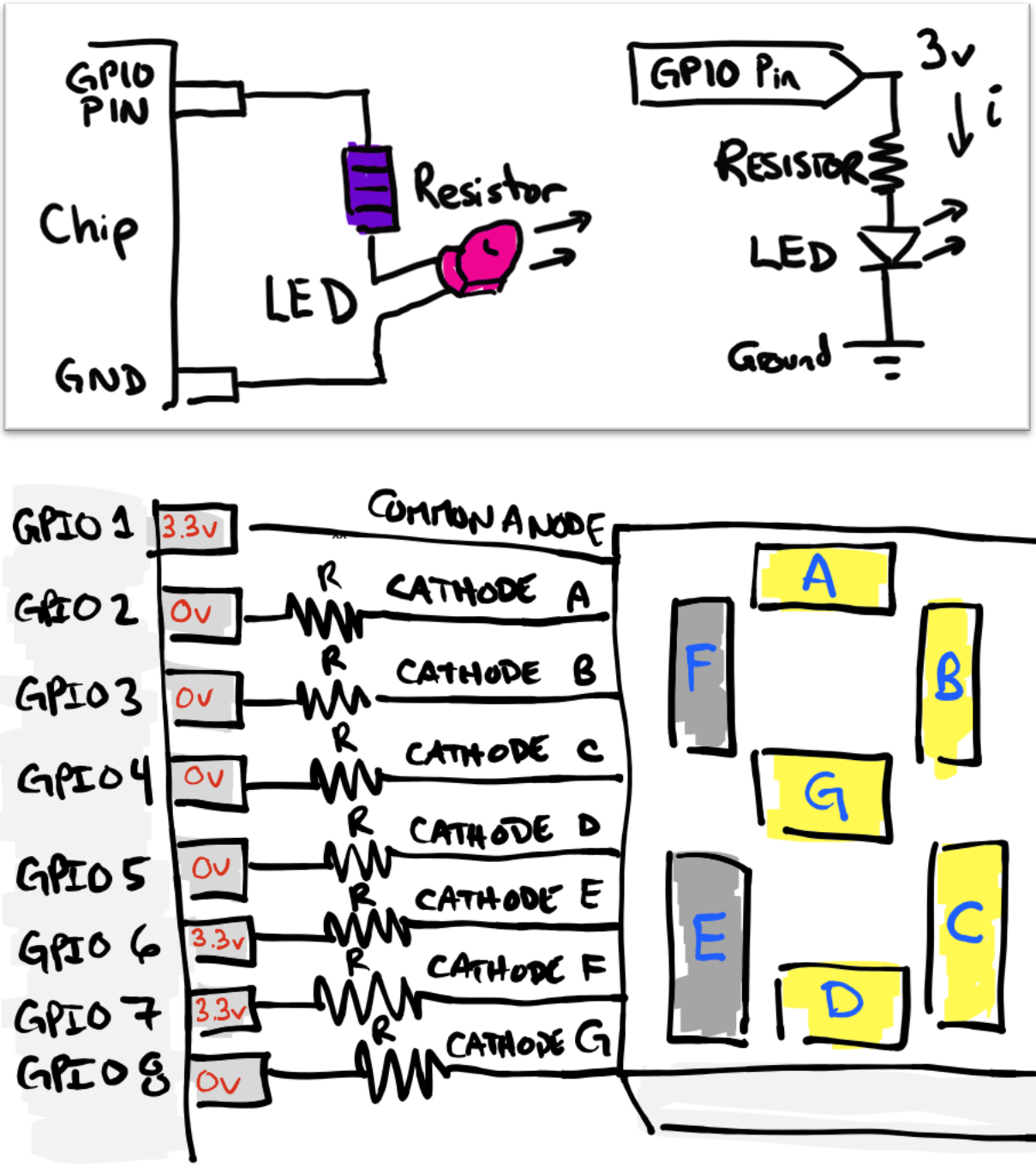
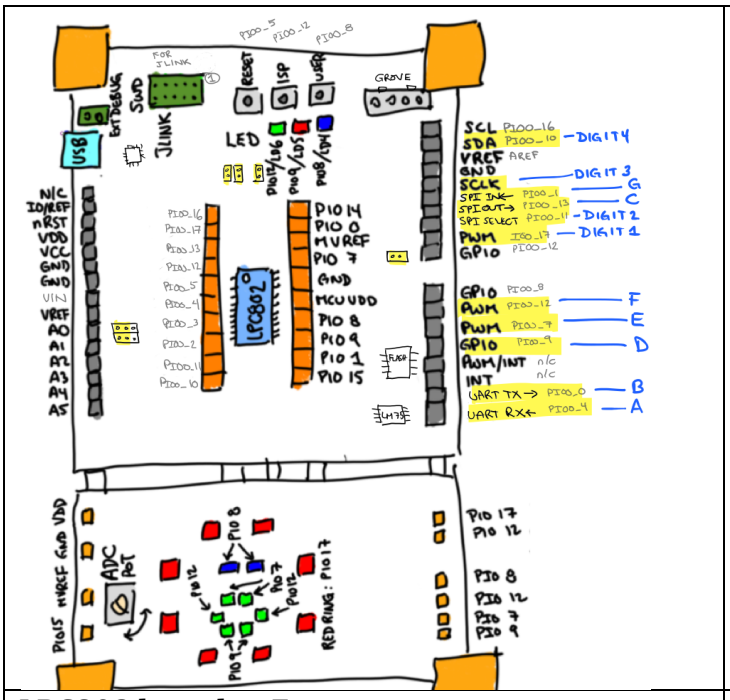


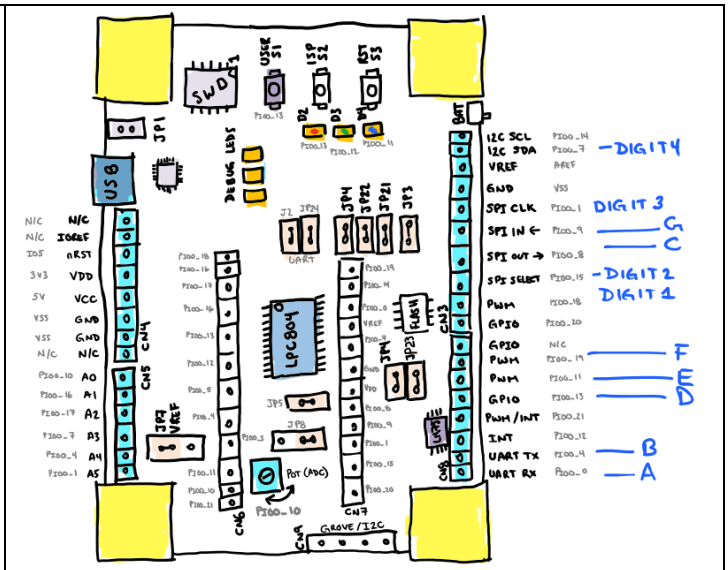
Figure 2 Driving a single LED from a microcontroller (top) vs driving multiple LEDs in a 7-segment LED display (bottom).

Signal (7-seg)		AWS3492 Pin	Broadcom HDSP-B0xE	802 GPIO	OM40000	804 GPIO	OM40001
		Common Anode	Common Anode				
		4 Digit	4 Digit				
		Small 7 Seg	Large 7 Seg				
	URL	https://bit.ly/2VF0yjg	bit.ly/36mD0Cp		Refer to Schematic		Refer to Schematic
	URL	https://bit.ly/2PGQP8j	https://bit.ly/2I8ltST	chip GPIO	board pin	chip GPIO	board pin
Digit 1		1	12	17	CN3 23 ("PWM")	18	("PWM")
Digit 2		2	9	11	CN3 22 ("PWM/SSEL")	15	("PWM/SSEL" or "SPI Select")
Digit 3		6	8	14	CN3 19 ("SCK")	1	("SCK" or "SPI CLK")
Digit 4		8	6	10	CN3 16 ("SDA")	7	("I2C SDA")
Segment A		14	11	4	CN3 32 ("CPU_RX")	4	("RXD" or "UART RX")
Segment B		16	7	0	CN3 31 ("CPU_TX")	0	("TXD" or "UART TX")
Segment C		13	4	13	CN3 21 ("PWM/MOSI")	8	("MOSI" or "SPI OUT")
Segment D		3	2	9	CN3 28 ("PWM")	13	("GPIO")
Segment E		5	1	7	CN3 27 ("PWM")	11	("PWM")
Segment F		11	10	12	CN3 26 ("PWM")	19	("PWM")
Segment G		15	5	1	CN3 20 ("MISO")	9	("MISO" or "SPI IN")

Figure 3 Pin mappings for the two common-anode 7-segment displays. Most students will likely be using the Broadcom 7-segment, with the LPC802 (OM40000 board).



LPC802 board to 7-segment



LPC 804 board to 7-segment

Problem 0: Build your 7-segment display add-on board.

You need to put together the four digit 7-segment display on the mini breadboard that came with your shield.

Ask yourself the following questions:

- How many LEDs do you need to connect?
- Do you notice that many of the LEDs share common connections?
- Do you notice that there are four groups of seven LEDs connected together?
- Do you notice that there are a few LEDs that are separate from the groups of seven?
    - these are “dots” in the display
- Did you buy the suggested 7 segment display?
    - If you bought a different one, is it a “common anode” or “common cathode” 7 segment?
    - If it’s a common cathode then it will be connected slightly differently than the instructions given here.

Your laboratory student learning kit hardware contains the common anode 7-segment LED (Broadcom HDSP was suggested), Arduino-style prototyping “shield” (DFRobot shield was suggested), small breadboard, with adhesive backing, wires and resistors. Inspect that you have all of these parts.

Look at the datasheet for the 7-segment display. First is the one for the Broadcom display, then the AW3492 display. You may have a different display. As always, check the datasheet for the one you bought. You can see the schematic for the LEDs in the Broadcom display in the following images:

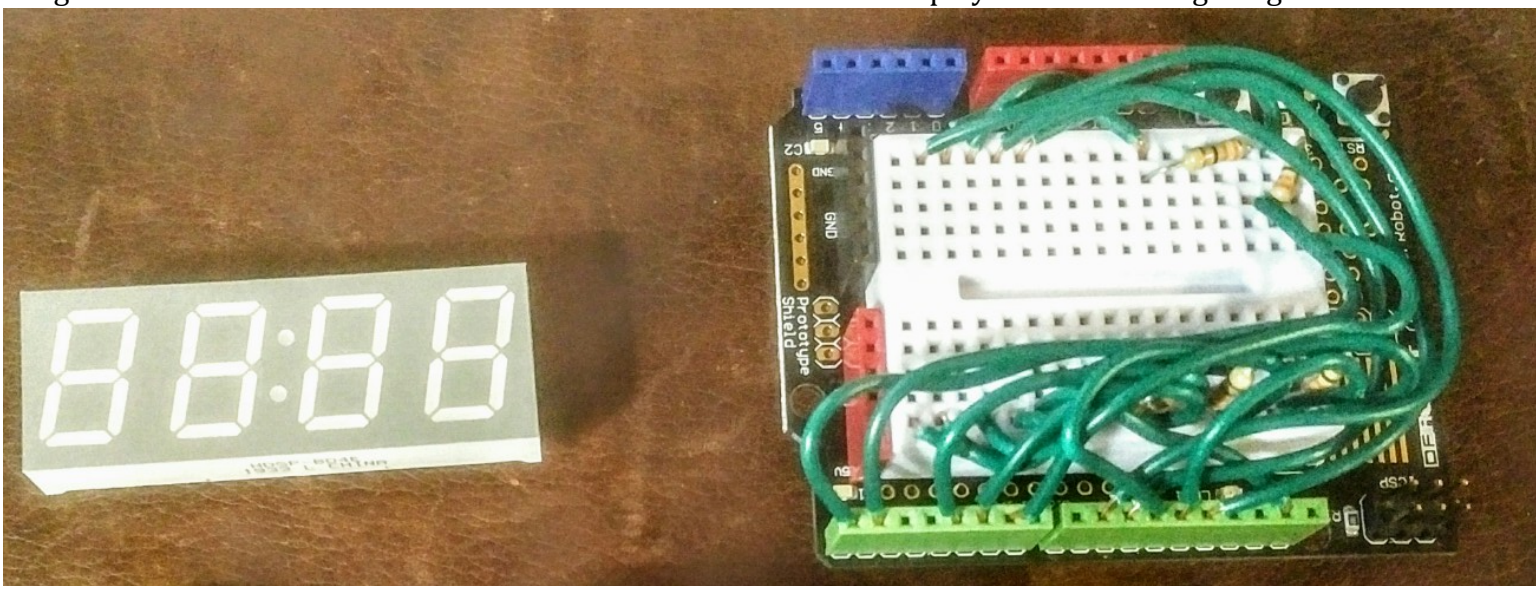
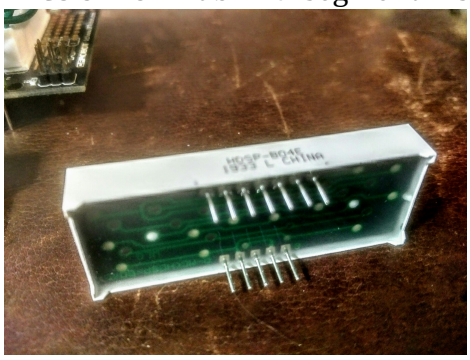


Figure 4 You'll need to wire up your shield and breadboard, inserting resistors as described earlier. It's a little tricky since there isn't a lot of space for the display, the wires and the resistors on the breadboard. Look carefully at how I crammed it all in.



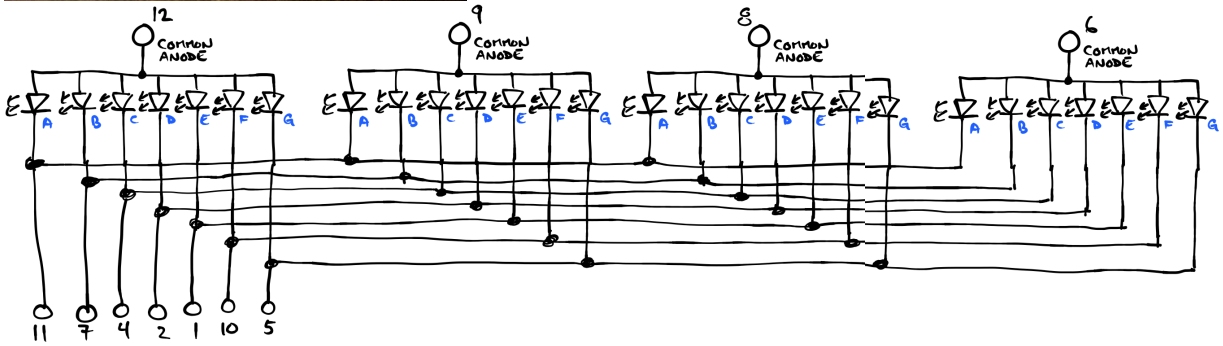


Figure 5 Look under the Broadcom HDSP-B04E display to view the two rows of pins. You'll need to connect them to the shield, paying attention to the anodes and cathodes from the datasheet. Pinouts for the Broadcom common anode 7-segment display are redrawn here. Digit 1 connects to Pin 12, Digit 2 to Pin 9, Digit 3 to Pin 8 and Digit 4 to Pin 6. Segments A through G connect to pins 11, 7, 4, 2, 1, 10, 5, respectively. Extra LEDs for dots are not shown.

And for the AW3492 display in the following image:

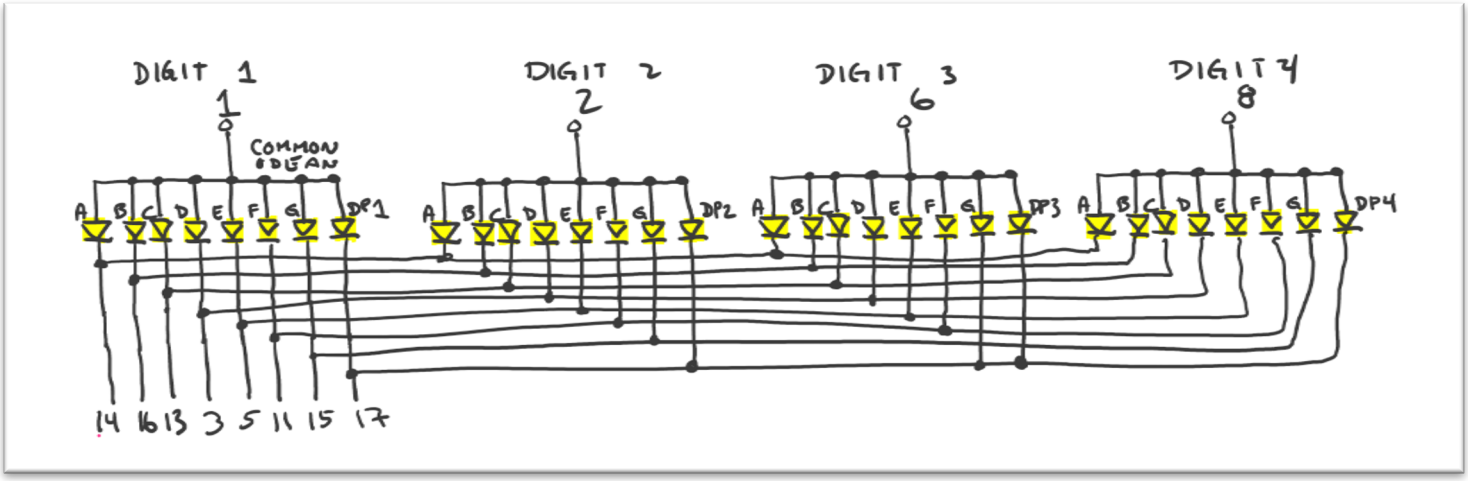


Figure 6 The pinouts for the AW3492 display. This schematic is missing a few of the dot LEDs. Don't worry about those.

As I did in a previous lab for the Kinetis KL43Z and a AW3492 7-segment display (https://youtu.be/U3OPSImEiEE) draw a pin assignment that shows how you will connect up your 7 segment display and the LPC802 (or LPC804) board. If it is common anode, like the one that you were requested to purchase for your lab kit, then you should arrange connections, generally, as follows:

- *Direct* connections from four of the GPIO pins on the LPC802 to the four common anode pins on the display
- *Indirect* connections, via resistors in series that limit current to the LEDs, from seven GPIO on the LPC802 to seven common connections to segments on the LED display.

Here is an example wiring drawing for an AW3492 7-segment display and the Kinetis KL43 microcontroller. You should do something similar, but for your specific display and the LPC802 or LPC804 board.

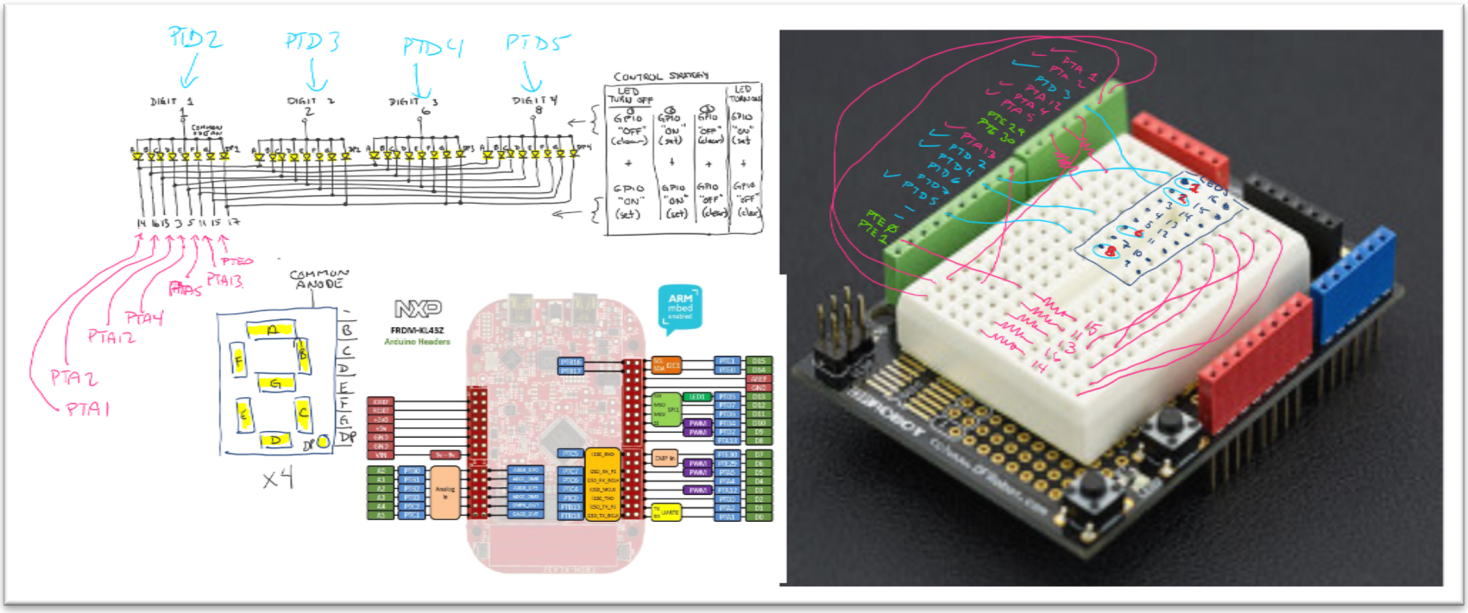


Figure 7 Wiring plan for a 7 segment display and a different microcontroller, the NXP KL43Z.. Your wiring diagram (plan) will be different for the LPC802 or LPC804 board that you're using. I documented this in a video here: https://youtu.be/U3OPSImEiEE

Note that each of these seven resistor-connected lines leads to four different LEDs – one on each of the four digits on the display. [2]

---

*In the video, you may notice that when I started putting together the display on the prototyping board, I screwed up wiring the 7 segment display twice but I **caught my mistake** because I was documenting the wiring and verifying the actual wired board against my documentation. This is good practice as it helps you catch common errors. It initially takes more time, but it **saves you time during debugging**! Consider doing the same thing!*

---

[2] Note that there are actually eight connections made in the video and in the datasheet. Seven of these are to rectangular segments that form the digits and one to a dot next to each digit.

Once you have connected up the 7 segment display test it by programming one of the GPIO pins connected to a common anode on the display and one of the GPIO pins connected to a cathode. This process is illustrated in the video (https://youtu.be/U3OPSImEiEE). By using the MCUXpresso debugger and the debugger connection on the LPC802 board you can step through your code, step by step to verify that the registers are set properly and that the LED lights up at the right time.

Now that you have tested one segment, you need to test all of the other ones. Design a program that can step through lighting up each segment on each digit, one at a time.

Show your wired-up display to the TA. None of the LEDs in the display needs to be lit up.

Problem 1: Light up all segments in the four digit 7-segment display

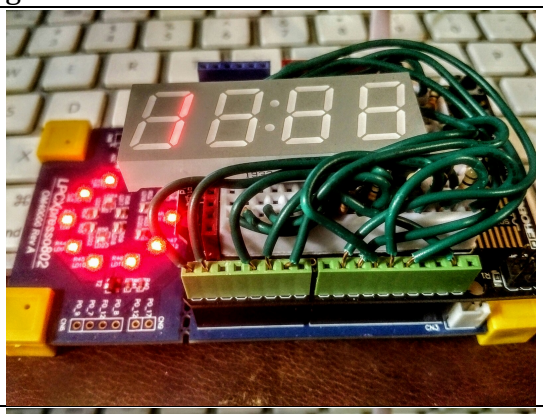
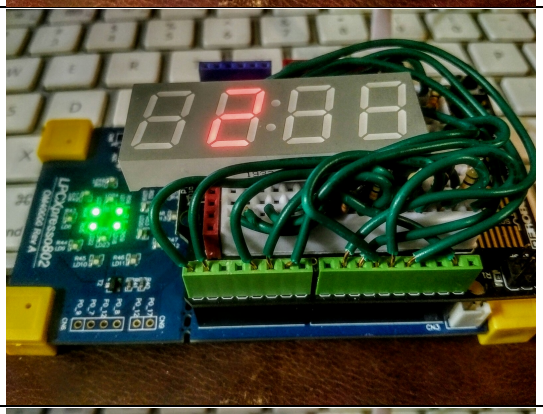
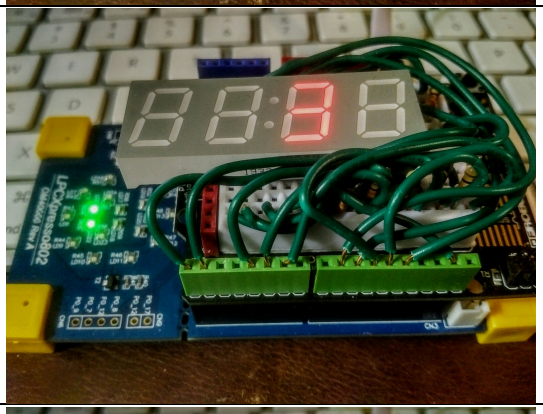
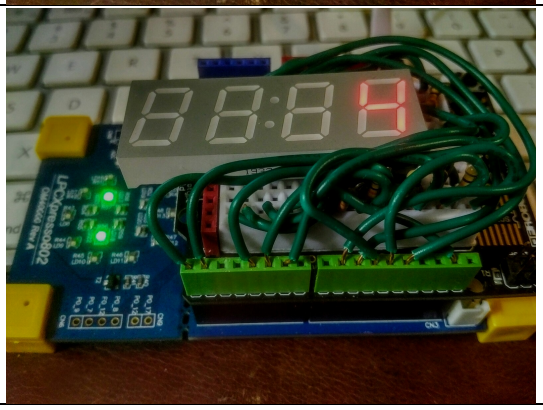
Set the four GPIO pins associated with digits in to logic "1" (3.3v). Set all the GPIO pins associated with the segments to logic "0" (0.0v). All the LEDs in the four digit 7-segment display should light up. Some will be bright, while others won't be so much. That's due to the current flow through the current-limiting resistors.

Show this to the TA.

Problem 2: Display your student ID

For this task, write a program that will display the last four digits of your student ID, one digit at a time. If the last four digits of your student ID are 1234 you would do the following:

1. Turn off all the LEDs in the four digit seven segment display
2. Turn on the segments to display "1" in the leftmost 7-segment digit in the display (that's segments B and C).
3. Wait a third of a third of a second (or so)
4. Turn off all the digits and all the segments so that nothing is visible. Don't add a delay.
5. Immediately after, turn on the segments to display "2" (segments A,B,G,E,D) and then turn on the second-from-left digit, while keeping the other digits off.
6. Repeat steps 3 and 4
7. Immediately after, turn on the segments to display "3" and then turn on the second-from-right digit, while keeping the other digits off.
8. Repeat steps 3 and 4
9. Immediately after, turn on the segments to display "4" and then turn on the right-most digit, while keeping the other digits off.
10. Repeat steps 3 and 4
11. Go back to step 2.

Display 1st digit	
Display 2nd digit	
Display 3rd digit	
Display 4th digit	

In your case you need to light up the segments that display the last four digits in your student ID. Each student in a pair needs to do their own value.

You can implement the delay with a blocking loop or with a timer and interrupt service routine.

Show this to the TA.

Problem 3: Pulse width modulate using the CTimer

In earlier labs we SysTick timer to create a clocking signal is an example of a variable-frequency signal that spends half its time generating an "on" signal and half its time generating an "off" signal. You don't control the ratio of on-to-off times, but you do control the frequency of the SysTick clock. Similarly, the CTIMER can generate a controllable frequency signal with a 50% duty factor. Here, we'll use Channel 3 inside CTIMER to control frequency and we'll output the result to the LED connected to PI0_9 on the outside of the chip. We'll assume that you're using a 12 MHz internal clock to drive the CTIMER.

At its heart the CTIMER will run a 32 bit counter at (default) 12 MHz. The value in the counter (CTIMER0->TC) will constantly update until its value matches that of one of the "match" registers in CTIMER, at which point the CTIMER will examine the EMR register to determine if it should stop, reset or call an interrupt. If it's configured to stop, the timer will simply stop and nothing further will happen. This one-shot behaviour can be useful for blocking delays, for instance. If it is configured to reset then the timer will begin its count again. This is useful for 50% duty cycle pulse streams. If it is configured to interrupt it will call an interrupt service routine.

Here is a simple case of using the **CTIMER to call an ISR to toggle an LED**. This is similar to how we've used other timers to toggle GPIO.

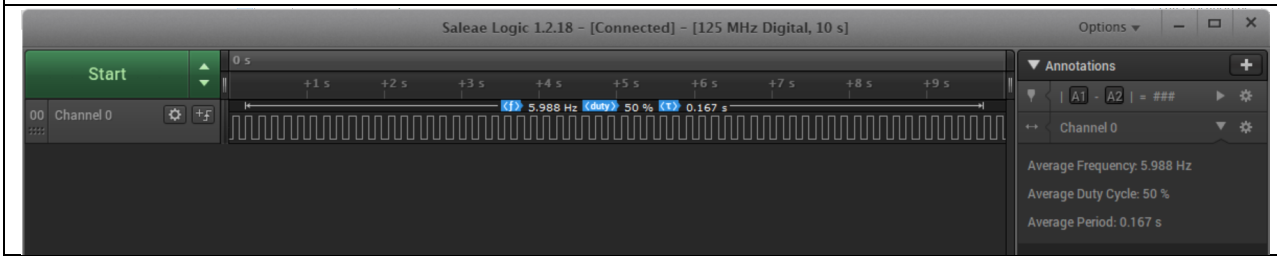
In this function, only one match is done, via Ch. 0. No reset of the timer is done automatically. Rather, reset of the timer only happens within the ISR, as does the GPIO, because there is no routing of the Timer via the Switch Matrix out to the pins. What is the Switch Matrix? We'll talk about that soon in the second example. But first, here's the first example.

Part A of Problem 3: CTIMER to drive an ISR to change GPIO directly.

```
/**
 * @file    ch6_ctimer_ch0_match_isr_50duty.c
 * Use the CTimer Match to output pulses to
 *    1. Blue LED (GPIO 8) indirectly via Interrupt on Match on CTIMER Ch. 0.
 *
 * -James
 * May 2, 2019 (updated March 2020)
*/

#include "LPC802.h"
#include "clock_config.h"  // for BOARD_BootClockFRO30M(), etc.

#define CTIMER_MATCH_VALUE (1000000)// visible LED blinking: 1000000 +
#define CTIMER_PRESCALE (1)   // (1 to 2^32): Prescale divisor for APB clock for CTIMER counter

#define LED_USER1 (8) // Blue LED (LD19 LD20)
#define LED_USER2 (9) // Red LED (LD21 and LD22; and LD5)


int main(void) {

    // disable interrupts
    __disable_irq();                    // turn off globally
    NVIC_DisableIRQ(CTIMER0_IRQn);    // turn off the CTIMER interrupt.

    // ---------------------- Begin Core Clock Select  ----------------------
    // Specify that we will use the Free-Running Oscillator
    // Set the main clock to be the FRO
    // 0x0 is FRO; 0x1 is external clock ; 0x2 is Low pwr osc.; 0x3 is FRO_DIV
    // Place in bits 1:0 of MAINCLKSEL.
    SYSCON->MAINCLKSEL = (0x0<<SYSCON_MAINCLKSEL_SEL_SHIFT);

    // Update the Main Clock
    // Step 1. write 0 to bit 0 of this register
    // Step 2. write 1 to bit 0 this register
    SYSCON->MAINCLKUEN &= ~(0x1);    // step 1. (Sec. 6.6.4 of manual)
    SYSCON->MAINCLKUEN |= 0x1;              // step 2. (Sec. 6.6.4 of manual)
```

```
    // Set the FRO frequency (clock_config.h in SDK)
    //
    // For FRO at     9MHz: BOARD_BootClockFRO18M();
    //                      12MHz: BOARD_BootClockFRO24M();
    //                      15MHz: BOARD_BootClockFRO30M();
    // See: Section 7.4 User Manual
    // This is more complete than just using set_fro_frequency(24000);
    BOARD_BootClockFRO24M();  // 30, 24 or 18 to get 15, 12 or 9 MHZ FRO.
    // ---------------------- End of Core Clock Select  ----------------------

    // ---------------------- Begin GPIO setup ---------------------------------
       // Set up a general GPIO for use within the Interrupt Service Routine for CTIMER
    // Only the ISR needs the GPIO.  (GPIO 8)
       SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_GPIO0_MASK);     // GPIO is on
       GPIO->DIRSET[0] = (1UL<<LED_USER1);                               // output on GPIO 8
       GPIO->CLR[0] = (1UL<<LED_USER1);                                  // LED is on GPIO 8
       // ---------------------- end of GPIO setup ---------------------------------


       // ---------------------- Begin CTIMER code (Match on Ch 0) ---------------
    // Match and Interrupt on Ch. 0.  Toggle within the ISR.
    //
    // Enable the CTIMER clock: SYSCON->SYSAHBCLKCTRL0 CTIMER0 clock
    SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_CTIMER0_MASK);

    // Reset the CTIMER module
       // 1. Assert CTIMER-RST-N
       // 2. Clear CTIMER-RST-N
       SYSCON->PRESETCTRL0 &= ~(SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);   // Reset
       SYSCON->PRESETCTRL0 |= (SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);          // clear the reset.

       // Match Channel 0 and generate IRQ
       CTIMER0->MCR |= CTIMER_IR_MR0INT_MASK;   // interrupt on Ch 0 match

       // Match at this rate for ch 0 for ISR
       CTIMER0->MR[0] = CTIMER_MATCH_VALUE; // this should determine period

       // prescale the counter clock from APB bus.
       CTIMER0->PR = (CTIMER_PRESCALE-1); // PR = 0: Divide by 1 of APB clock
                                                                  // PR = 1: Divide by 2
                                                                  // PR = 2: Divide by 3...
       // Enable the timer.
       CTIMER0->TCR |= CTIMER_TCR_CEN_MASK;

    // At this point, the TCR ENable bit gets set to 1
    // and then, automatically, we should see the timer TCVAL value change.
    // that should confirm that the timer is actually running.
       // ---------------------- end of CTIMER code -------------------------

       // enable global interrupts & the CTIMER IRQ.
       NVIC_EnableIRQ(CTIMER0_IRQn);
       __enable_irq();  // global



    /* Enter an infinite loop, do nothing in particular here.*/
    while(1) {
        asm("NOP");
    }
    return 0 ;
}

// Interrupt Service Routine for _all_ CTIMER functions
void CTIMER0_IRQHandler(void) {
```

```
        // Test the bit for Interrupt on Match Channel 0
        if(((CTIMER0->IR)>>CTIMER_IR_MR0INT_SHIFT) & 1)
        {
                // RESET the interrupt flag by writing a ONE (1) to the bit.
                // Only concerned with Channel 0 Matching.
                CTIMER0->IR = CTIMER_IR_MR0INT_MASK;

                // toggle LED
                GPIO->NOT[0] = (1UL<<LED_USER1);

                // Manually reset the timer.
                // Set the timer counter and prescale counter back to 0
                CTIMER0->TCR |= CTIMER_TCR_CRST_MASK;       // set bit 1 to 1
                CTIMER0->TCR &= ~(CTIMER_TCR_CRST_MASK);    // clear bit 1 to 0
        }
        else
        {
                // all other interrupt requests for CTIMER would end up here.
                asm("NOP");
        }
  return;
}
```



Show the TA this working code.

Switch Matrix: Setting up the connections from the Ctimer to the External Pins

The **CTIMER can control the LPC802's external pins directly**, without using the GPIO module. Before you set up the timer you need to connect the timer to the external pins where the PWM signals will come out. You do this through the switch matrix. The **Switch Matrix** functions a bit like an old-school telephone switchboard.

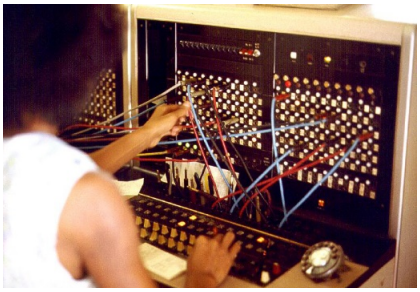



Figure 8 https://upload.wikimedia.org/wikipedia/commons/d/dc/Jersey_Telecom_switchboard_and_operator.jpg and https://en.wikipedia.org/wiki/Telephone_switchboard#/media/File:Offutt_Air_Force_Base_operator.jpg

Out of reset, by default, all pins are GPIO except the ones involved in debugger programming (SWDIO (GPIO PIO0_2 / Physical Pin 8), SWCLK (GPIO PIO0_3 / Physical Pin 9) and RESET* (GPIO PIO0_5 / Physical Pin 5) and power (physical pins 15 and 16).[3] You have to take GPIO control away from other pins if you want other peripheral devices to directly use them. (GPIO is considered just another peripheral module)

The switch matrix operates as per this diagram

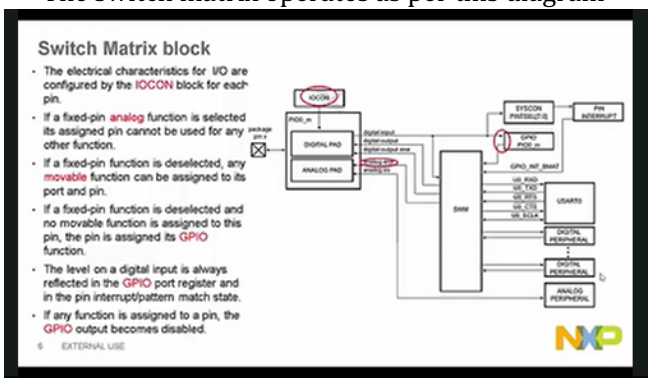


Figure 9 https://web.archive.org/web/20190403072749/https://www.nxp.com/video/lpc80x 微控制器系列开关矩阵技术详解:LPC802-Switch (https://bit.ly/2TOyXYx)

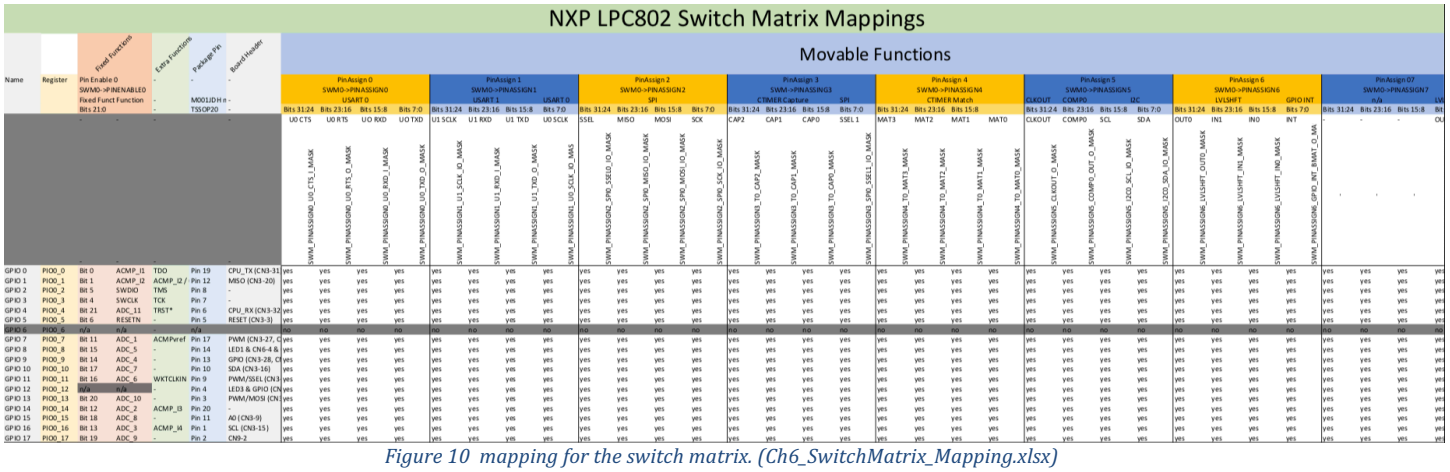


Figure 10 mapping for the switch matrix. (Ch6_SwitchMatrix_Mapping.xlsx)

[3] Section 8.3.2 of the User Manual.

Look Ma, No Interrupt!: Pulse Width Modulation by connecting the Timer directly to the pins

The CTIMER ("Counter Timer") module on the LPC802 is a little intimidating to configure as it contains more registers than the equivalent PIC16 or ATMEGA. NXP has LPCWare-based examples and MCUXpresso Software Development Kit (SDK) tools available at the time of this writing that can ease you into the intricacies of the CTIMER. However, if you follow the example below you'll see that it's actually not that bad.

Let's begin by looking inside the timer module. It is driven by a clock from the AHB bus. Out of reset this is a 12 MHz clock, although this can be changed. The CTIMER0 module will initially start counting at a rate of 12 MHz. There are two basic variables we need to control PWM on any device: frequency and duty factor. Two variables means that we need at least two registers to contain values that permit frequency and duty factor to be calculated with respect to the main clocking signal. In this example, we'll use the MR register for Channel 0 and the MR register for Channel 3 to control the two variables. You'll see that in many examples we will have up to three PWM channels outputting signals and Channel 3 is dedicated to supporting the second control variable for PWM on all channels.

In our most basic example, we first example simply set up one PWM signal on Pin 13 of the LPC802M001 chip. Pin 13 is normally used by the GPIO to drive an LED on the LPCXpresso board. Out of reset this pin is used for general purpose I/O by the GPIO module and so it is usually called "PIO0_9", IO 9 of the GPIO peripheral. This provides us with our first challenge: we need to disconnect the GPIO peripheral module from Pin 13.

The ATMEGA328 is an example of the approach taken in the 1990s and early 2000s of dedicating one (or two) functions to each pin of a microcontroller. With the LPC802, the SAMD21 and even the 8-bit PIC16F1619, there are more peripheral devices than available pins. The internal silicon is cheap… it's the pins on the outside of the chip that are expensive. So, the modern approach – the most cost-effective approach – is to provide only a few pins and lots and lots of peripherals on the inside, allowing the programmer or engineer to use software to manage the interconnects.

To connect and disconnect internal peripheral modules from the external pins we have to use the "Switch Matrix." There are multiple registers within the Switch Matrix to connect these different peripherals. Looking at the User Manual, we see that to connect the CTIMER to any external pin we need to use register "Pin Assign 4":

SWM0->PINASSIGN.PINASSIGN4

We're going to use Channel 0 within the CTIMER as the PWM generator that connects to the outside world, so we need to tell PINASSIGN4 that. The User Manual informs us that bits 7:0 control Channel 0, so we have to load SWM0->PINASSIGN.PINASSIGN4 bits 7:0 with the value 9, which means PIO0_9. We do this as follows:

1. Turn on the Switch Matrix
   a. `SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_SWM_MASK);`
2. Reset the Switch Matrix
   a. `SYSCON->PRESETCTRL0 &= ~(SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);  // Reset`
   b. `SYSCON->PRESETCTRL0 |= (SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);  // clear the reset.`
3. Zero-out the bits in the PINASSIGN register that correspond to the IO pin of interest
   a. SWM0->PINASSIGN.PINASSIGN4&= ~(0xff); // set bits 7:0 to zero in Pin Assign 4
4. Fill the bits with the IO pin of interest
   a. SWM0->PINASSIGN.PINASSIGN4 |= (0x9); // set bits 7:0 to 9 in Pin Assign 4 to correspond to GPIO pin PIO0_9 (pin 13 on the TSSOP package)
5. Make one of the Channels in CTIMER dedicated to PWM. In this case, Channel 0:
   a. `CTIMER0->EMR |= CTIMER_EMR_EM0_MASK; // (rule complicated)`
   b. `CTIMER0->PWMC |= CTIMER_PWMC_PWMEN0_MASK; // 0 is match mode; 1 is PWM mode.`
6. `Clear all the control bits for the CTIMER's MCR`
   a. `CTIMER0->MCR &= ~(CTIMER_MCR_MR0R_MASK | CTIMER_MCR_MR0S_MASK | CTIMER_MCR_MR0I_MASK);`
7. Begin by setting Channel 3 to act as a holder of important reload values
   a. `CTIMER0->MCR |= CTIMER_MCR_MR3R_MASK;  // MR3R bit to 1`
8. Define the period of your pulsing within Channel 3
   a. `CTIMER0->MR[3] = 12000000;  // # of counts for complete PWM cycle.`
9. `Define the PWM pulse period within Channel 0 (duty factor of 75%)`
   a. `CTIMER0->MR[0] = (int)((1-0.75)*12000000);  // # of counts for off-time`
10. Start the timer and walk away.
    a. `CTIMER0->TCR |= CTIMER_TCR_CEN_MASK;  // 0 is disabled. 1 is enabled.`

At this point the timer will begin (you can even see the initial value of the timer within the CTIMER0->TC register as soon as the TCR receives the enable bit value.

We can assume that the clock driving the CTIMER0 is running at 12 MHz. This means that each count taken by the CTIMER takes 8.3333333x10^(-8) seconds (0.83 nanoseconds). For a PWM period of 1 second (freq: 1Hz) we need to count to 12 million, so MR[3] needs to be loaded with 12000000. For a PWM period of 1kHz we need to count to 12000 (0.001/(8.3333333x10^(-8))).

The on-off time ratio is determined as a fraction of the MR[3] value, placed in MR[0]. Take the MR[3] value multiple it by the inverse of the desired duty factor and round it to the nearest integer: (int) (1/0.75)*12000000 will give a 25% duty factor.

By default the CTIMER0 system is driven by the 12 MHz clocking signal on the AHB bus to most peripherals.

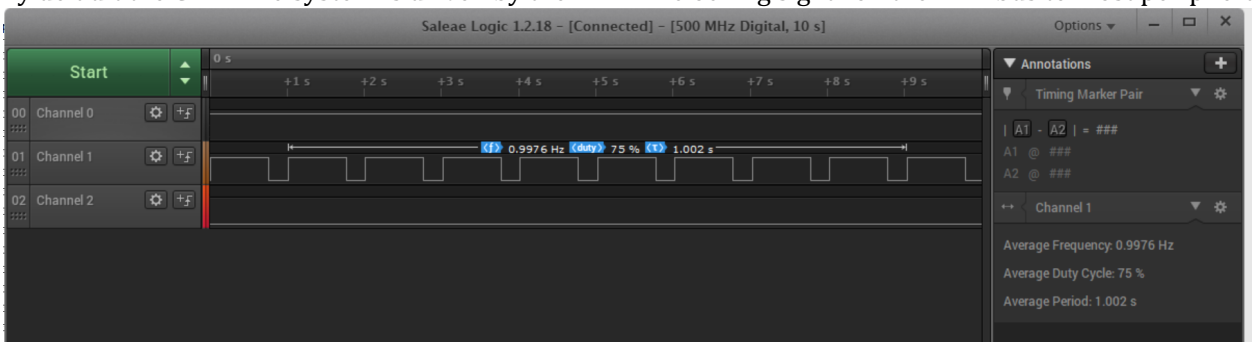


Figure 11 When AHB clock is the default 12MHz, setting MR[3] to 12000000 and MR[0] to 25% of 12000000 (3000000) we get a 1Hz pulse at 75% duty factor. (LPC802_Project_ch6_ctimer_match_v2_working)

Problem 3, Part B: Example code connecting the CTimer directly to the pins using the Switch Matrix

```
/**
 * @file    ch6_ctimer_ch3_75percent_duty_PWM.c
 *
 * Perform 75% Duty Factor using Ch0 and Ch 3 matching on CTIMER.
 *
*/

/**
 * @file    LPC802_Project_ch6_ctimer_v3.c
 * @brief   Application entry point.
 */
#include "LPC802.h"

#define CTIMER_FREQ (12000000)
#define PWM_PERCENT (0.75)// value b/w 0 and 1.
#define CTIMER_MATCH (CTIMER_FREQ*(1-PWM_PERCENT))


int main(void)
{

     // assume 12 MHz FRO clock

    // enable switch matrix
    SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_SWM_MASK);

    // Set switch matrix
    // ---> PINASSIGN4 T0-MAT0 is (bits 7:0) is 0x9
    // Connect CTimer0 channel 0 to an external LED pin
    SWM0->PINASSIGN.PINASSIGN4 &= ~(0xff); // clear the bottom 8 bits
    SWM0->PINASSIGN.PINASSIGN4 |= (0x9);// put 0x9 is the bottom 8 bits.
    // this should make the GPIO with the Green LED pair the output pin. (PB 9)


       // Enable CTIMER clock
       SYSCON->SYSAHBCLKCTRL0 |= (SYSCON_SYSAHBCLKCTRL0_CTIMER0_MASK);

       // Reset the CTIMER module
       // 1. Assert CTIMER-RST-N
       // 2. Clear CTIMER-RST-N
       SYSCON->PRESETCTRL0 &= ~(SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);  // Reset
       SYSCON->PRESETCTRL0 |= (SYSCON_PRESETCTRL0_CTIMER0_RST_N_MASK);   // clear the reset.

       // Enable PWM mode on the channel 0
       // ************************************
       // 1. EMR bit 0 set to 1  (External Match Register)
       // 2. PWMEN0 (PWMC bit 0) set to "PWM"
       // ************************************
       CTIMER0->EMR |= CTIMER_EMR_EM0_MASK;               // (rule complicated)
       CTIMER0->PWMC |= CTIMER_PWMC_PWMEN0_MASK;          // 0 is match mode; 1 is PWM mode.

       // Clear all the Channel 0 bits in the MCR
       CTIMER0->MCR &= ~(CTIMER_MCR_MR0R_MASK | CTIMER_MCR_MR0S_MASK | CTIMER_MCR_MR0I_MASK);

       // Reset the counter when match on channel 3
       // *********************************
       // MCR bit MR3R gets set to 1
       // *********************************
       CTIMER0->MCR |= CTIMER_MCR_MR3R_MASK;       // MR3R bit to 1

       // Match on channel 3 will define the PWM period
       // If we use the 12MHz internal clock, then 12000000 will make it 1 Hz.
       // *********************************
```

```
        CTIMER0->MR[3] = CTIMER_FREQ;              //

        // This will define the PWM pulse off time.
        // If this is 75% of MR[3] value you'll get 25 Duty Cycle.
        // Make this freq * (1-PWM_DUTY_FACTOR).
        CTIMER0->MR[0] = CTIMER_MATCH;             // DUTY FACTOR is 0.75


    // Start the timer:
    // ************************************************
    // 1. CTIMER0->TCR CEN bit (bit 0) got set to ENABLED
    // 2. The timer counter (TC) will contain values... CTIMER0->TC
    // ************************************************
    CTIMER0->TCR |= CTIMER_TCR_CEN_MASK;  // 0 is disabled. 1 is enabled.
    // at this point, the TCR ENable bit gets set to 1
    // and then, automatically, we should see the timer TCVAL value change.
    // that should confirm that the timer is actually running.

    /* Enter an infinite loop, just incrementing a counter. */
    while(1) {

                    // nothing
                    asm("NOP");
    }
    return 0 ;
}
```

Show the TA this working code.

There's no PWM out. Now what?

This happens. Get out your debugger (and maybe an oscilloscope if you have one) and dive in the registers. First, run your program and pause it in the infinite while loop. Check out the counter value. Does it update each time you halt the code? If yes, then that tells you that your timer is working, even if the signal isn't making it to the outside world. Check your Switch Matrix. Did you configure it to connect the timer (PINASSIGN4 values) to the correct pin? Each set of 8 bits in the PINASSIGN4 register controls a different timer channel. Did you choose Channel 0 (7:0), Channel 1 (15:8), Channel 2 (23:16) or Channel 3 (31:24)? Is the right value assigned within?

Communication, Inter-Group (informal, not graded)

With others in the lab, discuss the following. What happens when you want to use this display in general? You should consider a strategy for creating a function that operates like "printf" for the 7 segment LCD. Can you think of a way that a function could accept a sequence of four digits in the input parameter and then convert that to a set of commands to the LEDs of the 7-segment display? Would using an array help?

Reflection on Learning *(informal self-assessment, not graded)*

After the formal lab, reflect on your experience. Did your program work perfectly the first time you tried to run it on the student learning kit? If not, did you have to add design features you didn't think of, or did problems arise because you did not use the correct instructions to implement the design? Could you have foreseen the problems and included the fix in the design phase? If so, what would you have had to know to be able to do this?