San José State University
Department of Computer Engineering
CMPE 146-01, Real-Time Embedded System Co-Design, Fall 2024

Lab Assignment 5

Due date: 11/03/2024, Sunday

1. Overview

In this assignment, we will implement a utility timing function for timing control. We will be familiarized with the GPIO's capacitive-touch capability and the Timer_A module on the MSP432 MCU. We will use the capacitive-touch capability as an input method to control a device.

2. Exercise 1 Delay function

In this exercise, we will build a small function to delay a specific number of milliseconds (ms). It will be a useful function for timing control purposes.

Exercise 1.1 Function implementation

Import the *empty* example project. Add the following line to the *include* file section for using *printf()*.

```
#include <stdio.h>
```

The delay function's prototype is as follows.

```
void delay_ms(uint32_t count);
```

We will use one of the 32-bit Timer32 timers in the function to implement the delay. For this exercise, we will use Timer32_0 (the same one we have used before). The input argument *count* is the number of ms to delay. Inside the function, use only the *MAP_Timer32_getValue()* and *MAP_CS_getMCLK()* DriverLib functions. Do not reset the timer or change the current value of the timer. Do not assume the system running at a particular frequency. Do not assume that the clock frequency is a nicely rounded number. Therefore, when the frequency is involved in the computation, be sure to preserve as much precision as possible. You may want to use 64-bit data types (like *uint64_t* or *long long*) for that purpose. To reduce computation overheads, do not use any floating point arithmetic, either explicitly or implicitly. Do not use interrupt. It is okay to use a loop to do the work.

For systems running at much higher clock frequencies than the default 3 MHz, we will quickly run into the situation that the timer wraps around back to 0xFFFFFFFF after counting down to zero. If no special handling is put in place to handle the situation, the amount of delay produced by the delay function will probably be incorrect. We will need to handle this wrap-around situation in the delay function.

To make the wrap-around situation happen more frequently for testing purpose, we will reduce the capacity of the timer from 32-bit to 16-bit. We change the one of the arguments to *MAP_Timer32_initModule()* from *TIMER32_32BIT* to *TIMER32_16BIT*. To set up the timer, we can use the following statements.

```
MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1, TIMER32_16BIT,
TIMER32_FREE_RUN_MODE);
MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
```

In the 16-bit mode, the timer still starts from 0xFFFFFFFF, but after it reaches 0xFFFF0000, it will return to 0xFFFFFFFF. The timer capacity is thus much reduced.

Use the timing method in the previous lab to measure if the delay function works properly. So, we can do something like this:

```
t0 = MAP_Timer32_getValue(TIMER32_1_BASE);
delay_ms(1000);
t1 = MAP_Timer32_getValue(TIMER32_1_BASE);
```

We will measure these delay values (in ms): 5000, 2000, 1000, 50, 20, 10, 5, 2, 1, 0. Use the order as listed; do not change it. To perform the time measurements for all the delay values, we will use the second timer in the Timer32 module, TIMER32_1. Set up the timer as follows.

```
MAP_Timer32_initModule(TIMER32_1_BASE, TIMER32_PRESCALER_1, TIMER32_32BIT,
TIMER32_FREE_RUN_MODE);
MAP_Timer32_startTimer(TIMER32_1_BASE, 0);
```

The usage is exactly the same as what we have done before; we just use a different timer ID when calling the DriverLib functions, i.e., we use *TIMER32_1_BASE* instead of *TIMER32_0_BASE*.

Print to the debug console the target delay time in ms, the difference of the two timer values (from TIMER32_1), the measured time in µs and error (difference between the measured and target delay) in µs. We can use an integer array to contain the list of delay values and use a *for* loop to go through all measurements. The measured times should be very close to the target delay values. If your errors are greater than 100 µs, you should examine the delay function carefully to reduce or hide unnecessary overheads in the delay function.

Setting Timer32_0 to 16-bit mode is for easier testing only (so that the wrap-around occurs very quickly). Your implementation shall work for both 32- and 16-bit modes.

Lab Report Submission

1. Show the delay function. Briefly explain how it works.
2. Show the debug console outputs that each line includes target delay time in ms, timer value difference, measured time in µs and error in µs.
3. Provide the program listing in the appendix.

Exercise 1.2 Timing control

We will use the delay function to blink the red LED (LED1) precisely at 0.4 Hz.

Duplicate the project in the previous exercise. Replace the measurement code with the code to blink the red LED. Compute the required delay for timing control in the program. Use the basic code in the previous labs to set up and blink the LED. Do not include unrelated control operations, like *printfs* or measurements, in the control loop because they will produce a less accurate blinking frequency.

Once we have finished the implementation, debugging and testing, we should verify or characterize what we have built. We should reply on a reliable and well-proven method independent of the system that we just built. Use a stopwatch (an independent means) to verify that the LED is indeed blinking at the correct frequency. Take at least three measurements. In each measurement, measure how long it takes to blink 10 times or more. Provide the data and compute the blinking frequency. With multiple measurements, compute the average frequency and the error in percentage from the target frequency. With enough samples and averaging, the random error in measurements can be greatly reduced. The amount of reduction is equal to the square-root of the number of samples taken. The measured error should be within 1% (provided the system is implemented correctly). If not, it is quite likely that there are bugs in the program and/or the measurements are not done properly. Note that one blinking cycle involves both the LED being turned on and off.

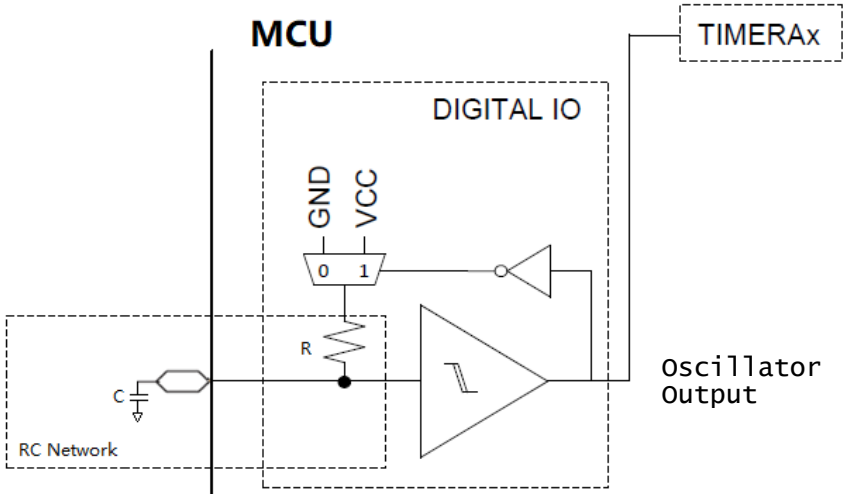
Lab Report Submission

1. Report the delay time used in the control loop. Briefly describe how the control loop works.
2. Briefly describe the measurement method. What parameters are used, for example, how many blinks do you use in the measurements?
3. Provide the data from your stopwatch measurements in a table form. Report the calculated frequencies, the final averaged frequency and error. You can use a spreadsheet for those purposes.
4. Provide the program listing in the appendix.

3. Exercise 2 GPIO oscillator

In this exercise, we will create an oscillator at a GPIO pin. We sample the state of the oscillator as fast as we can. We will also measure how fast we collect the data.

A GPIO input pin can be configured to form an oscillator, which consists of a simple RC network with a feedback loop. The resistance mainly comes from the built-in pull-up-pull-down resistor at the I/O pad. The capacitance does not come from a specific capacitor. Instead, it is a collection of parasitic capacitances from the MCU itself, connecting wires, circuit board, connector, etc.



To form an oscillator on the MCU, we'll need to enable the capacitive-touch feature at the GPIO port. For example, to create an oscillator at Port 4, Pin 1 (P4.1), we can execute the following statements to set up the control register of the feature:

```
CAPTIO0CTL = 0;          // Clear control register
CAPTIO0CTL |= 0x0100;    // Enable CAPTIO
CAPTIO0CTL |= 4 << 4;    // Select Port 4. Place port number in Bits 7-4
CAPTIO0CTL |= 1 << 1;    // Select Pin 1. Place pin number in Bits 3-1
```

To read the state of the oscillator's binary state output, we can read the control register and do something like this:

```
bool state = CAPTIO0CTL & 0x200;
```

Details of the capacitive touch control register can be found in Chapter 14 of the MCU technical reference. Section 14.3.1 explains how the control register is used. As Figure 14-3 in the technical reference (see below) shows, Bit 9 indicates the state of the oscillator, Bit 8 enables the capacitive touch feature, Bits 7-4 selects the port, and Bits 3-1 selects the pin.

Figure 14-3. CAPTIOxCTL Register

15	14	13	12	11	10	9	8
Reserved						CAPTIO	CAPTIOEN
r0	r0	r0	r0	r0	r0	r-0	rw-0
7	6	5	4	3	2	1	0
CAPTIOPOSELx				CAPTIOPISELx			Reserved
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	r0

Create a project (import the *empty* example project and include *stdio.h*) to continually show the state of the oscillator. To see a stream of binary outputs, we can use *printf()* without a newline. Make sure to add a *fflush(stdout)* call after *printf()* so that the output can be seen immediately. Use a forever loop to show all the states on one line. Set the debug console to "Wrap Word" so that the output will go to the next line if the current line is full; otherwise, new outputs cannot be seen. Each output should be just one character, either '1' or '0'.

If we see a stream of random 1's and 0's, that means the oscillator is probably working properly. Now, we modify the program to sample the oscillator output as fast as we can in software. Declare a data array of 100 elements. (You should use a macro to define the size.) Use a *for* loop to read the oscillator output and store the data in the array. Make sure there is no unnecessary overheads, like *printf*, in the loop because they will slow down the sampling. After all the data are collected, print out the bit stream, like what we have done above.

Knowing the sampling rate on an input signal is quite useful when we perform some signal analysis. Add the measurement code (using a Timer32 timer that we have used in previous labs) to time how long the *for* loop takes. Given the time, convert it to sampling rate in Hz, which basically tells us how many data samples we would get per second. Print the computed rate to the debug console.

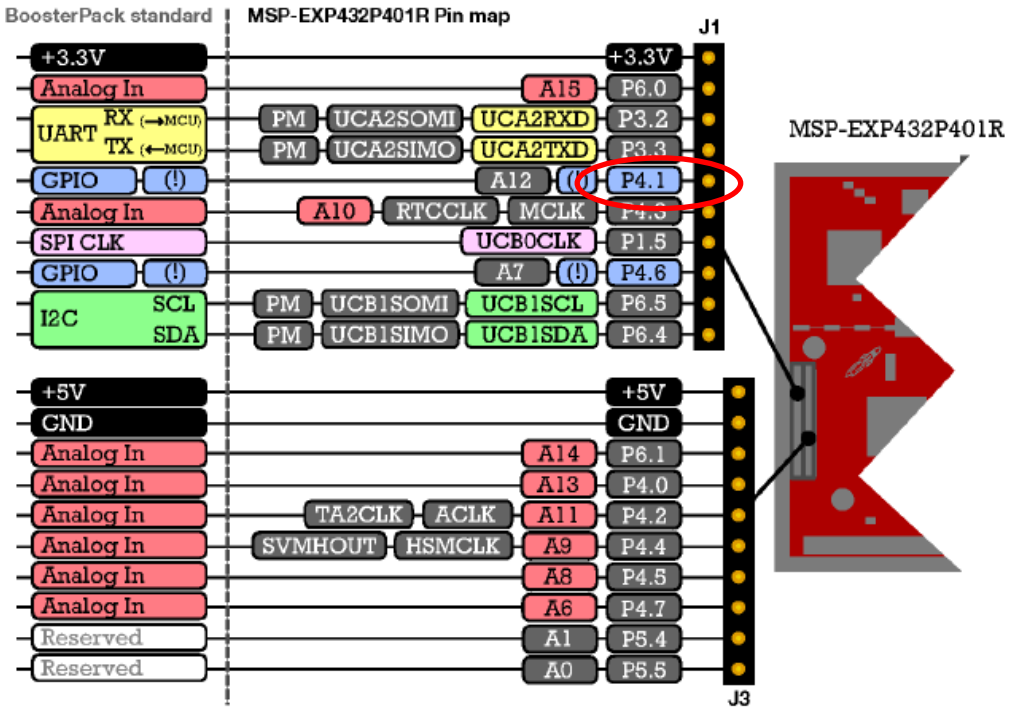
Lab Report Submission

1. Show the debug console outputs, which include the bit stream and the computed sampling rate.
2. With the proper code snippet, briefly explain how the data are collected and processed, and how the sampling rate is determined.
3. Provide the program listing in the appendix.

4. Exercise 3 Frequency measurement

In this exercise, we will use Timer_A to measure the frequency of the oscillator that we created in the previous exercise. We will change the frequency by touching the GPIO pin with our finger.

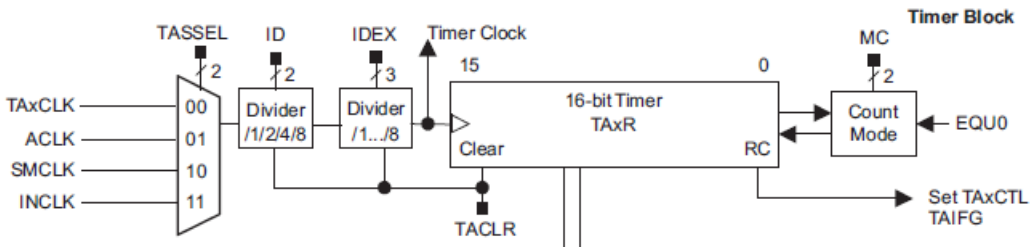
There are many GPIO pins we can get access to on the LaunchPad's connectors. For example, Port 4, Pin 1 is connected to Connector J1.



The P4.1 (refers to Port 4 and Pin 1) oscillator output can be routed to the A2 counter/timer in the Timer_A module. We will use the 16-bit counter to sense the change of the capacitance at the GPIO pin. The P4.1 oscillator provides a clock signal to the counter. We measure how many clock pulses it receives in a specific time window. If the count changes significantly, it is most likely caused by a change in the capacitance at the pin induced by a human's finger touch on the connector pin.

The more surface area a finger touches the pin, the more change of capacitance it would introduce. You can attach a wire to the J1 pin to increase the touch surface area. Alternatively, you can just insert a metal paper clip on the J1 connector on the backside of the board to make the same connection. If the surface area is large enough, placing the finger close to the wire but not touching it should introduce enough capacitance change to be detected.

Each counter in the Timer_A module accepts one of the four input clock signals. The P4.1 oscillator output is routed to the module as the INCLK signal.



We can use the Timer_A DriverLib functions to set up the counter to count the incoming pulses. The following statements configure the counter to count in continuous mode and start up the counter to count.

```
Timer_A_ContinuousModeConfig timer_continuous_obj;
timer_continuous_obj.clockSource = TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK;
timer_continuous_obj.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
timer_continuous_obj.timerInterruptEnable_TAIE =
TIMER_A_TAIE_INTERRUPT_DISABLE;
timer_continuous_obj.timerClear = TIMER_A_DO_CLEAR;
MAP_Timer_A_configureContinuousMode(TIMER_A2_BASE, &timer_continuous_obj);
MAP_Timer_A_startCounter(TIMER_A2_BASE, TIMER_A_CONTINUOUS_MODE);
```

Note that TIMER_A_CLOCKSOURCE_INVERTED_EXTERNAL_TXCLK refers to the INCLK input clock in the schematic. We set the clock divider to divide by 1, essentially not dividing because we want to be able to measure with the highest precision. We do not use interrupt. We clear the counter and disable counting when we configure it.

Duplicate the project in the previous exercise. Modify the main C file so that it would count how many counter ticks will be recorded in 3 ms (the measurement window). Basically, before the measurement, we clear the counter with a DriverLib function *MAP_Timer_A_clearTimer(TIMER_A2_BASE)* so that it counts up from 0 again. Delay for 3 ms. (We can use the delay function from Exercise 1.) Then read the counter register to get the count by calling *MAP_Timer_A_getCounterValue(TIMER_A2_BASE)*. Convert the counter value to frequency (in Hz) and print it to the debug console. The value tells us the current frequency of the oscillator.

In the program, repeat the measurement with a forever loop. Touch the J1 connector where P4.1 is connected to. Observe how the counter value and frequency change. What would you do to change the frequency slightly and drastically?

Make the program to take measurements for about 10 seconds. Use the Timer32 timer for timing the duration. Make sure there are not any unnecessary overheads in the loop so that it can make as many measurements as possible. Print the measured frequency to the debug console in the measurement loop. After the measurements, you can exit the program or enter a forever dummy loop (so that you can restart the program quickly if necessary).

Take a set of measurements without touching the pin. With the data from the console outputs, make a graph (with Excel, for example) to show how the frequency changes with time. Make sure you adjust (if necessary) the scale of the vertical axis so that you can easily see the small changes. Run the program again. Touch the pin a few times so that you see small and large changes in the frequency outputs. Make another graph to show the results.

Lab Report Submission

1. Show the debug console outputs.
2. Explain, with the code you have written, how you measure the oscillation frequency from the counter value.
3. Show the two graphs, the first one without touching and the second one with some touching. Make sure the graphs are labeled properly on both axes. Describe your observations from the graphs.
4. Explain the method (why it works) you used to make the frequency change slightly and drastically.
5. Provide the program listing in the appendix.

5. Exercise 4 LED control

In this exercise, we will use the capacitive touch feature to turn on and off an LED.

Based on what you have done in the previous exercise, write a program to control the red LED (LED1 on the LaunchPad). It will be similar to control a desk lamp: When we touch a switch, the lamp is turned on, and when we touch the switch again, the lamp is turned off. For this exercise, initially, the LED is off; when we touch the J1 connector where P4.1 is connected to, the program just toggles the state of the LED. Note that if we touch the pin and do not lift your finger, the state of the LED shall not change after it is turned on or off.

We can make the control decision based on the frequency of the oscillator that the program continually measures. For example, if the frequency is below some frequency threshold, we can assume P4.1 is touched. Whatever you do, you cannot hard-code the threshold parameter in the program for the control purpose because it may not work properly when the system's operating conditions change. To get reliable control parameters, which are dynamic, we should do some kind of calibration when the program starts up, and before the control loop is entered. We can safely assume that there is no human touch on the J1 connector during calibration. We take at least a few measurements under such condition to determine the control parameter(s). Once the parameter(s) is determined in the program, print it (or them) to the debug console.

In the control loop, do not print the measurement results, so as not to increase the response time to the human actions. The system needs to be able to respond to the human actions in real time. That is, when the touch is made, the LED should be turned on or off immediately so that the user won't see any lag of the response to the touch.

Take a short video clip of the control of the LED, demonstrating the real-time performance. Place the video file where it can be accessed remotely.

Lab Report Submission

1. Explain, with the code you have written, the mechanism you use to determine the control parameter(s) and the control of the LED.
2. Show the debug console outputs, which include at least the control parameter(s) determined.
3. Include a link to the video clip. Do not submit the video file to Canvas. Make sure the video file can be opened by just clicking on the link.
4. Provide the program listing in the appendix.

6. Submission Requirements

Write a formal report that contains at least what are required to be submitted in the exercises. Submit the report in PDF to Canvas by the deadline. Include program source code in your report. Do not use screenshots

to show your codes. In the report body, if requested, list the relevant codes or screenshots only for the exercise you are discussing. Put the entire program listing in the appendix. Use single-column format for the report.

In your report, show the debug console outputs (when requested) in text format; copy and paste them to your report. In general, do not use screenshot to show any text outputs or data; otherwise, points will be deducted.

When presenting a screenshot in the report, only present the relevant information. You may need to crop them from the initial screenshot. Please use proper scale so that the contents can be read easily.

Your report should have several sections. The first section is the information header that contains at least your name, course name and lab assignment number. Each exercise should have its own section where you discuss the results of the exercise. At the end, if requested, there should be the appendix that contains the complete source code for each exercise. The source code may be used to reproduce your reported results during grading. So, make sure they do produce the results you reported.

7. Grading Policy

The lab's grade is primarily determined by the report. If you miss the submission deadline for the report, you will get zero point for the lab assignment.

If your program (listed in the appendix) fails to compile, you will get zero point for the exercise.

If your program's outputs do not match what you reported, you will get zero point for the exercise.

Points may be deducted for incorrect statements, typos, non-readable texts on screenshots, lack of description on the graphs/pictures shown, etc. So, proofread your report before submission.

Pay attention to the required program behaviors described in the assignment. Points will be deducted for not producing the required outcomes.