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

Lab Assignment 7

Due date: 12/01/2024, Sunday

1. Overview

In this assignment, we will be familiarized with the ADC (analog-to-digital converter) and the temperature sensor of the MCU. We will also do some multitask programming with TI's real-time operating system, TI-RTOS. We will create tasks to do some control work in parallel.

2. Exercise 1 ADC

There is a temperature sensor in the MCU. The output of the sensor can be routed to one of the ADC input channels. We will convert the sensor's analog voltage to temperature using the sensor calibration data stored in the ROM.

The temperature sensor resides in the voltage reference module of the MCU. It produces a voltage proportional to the internal temperature of the MCU. Software can monitor the temperature changes and takes necessary actions if warranted. For example, it can prevent the chip from overheating or compensate for certain sensitive components' electrical characteristics drift.

The sensor's output voltage can be digitized with one of the ADC channels in the MCU. The ADC has a maximum of 14 bits of resolution, providing 16,384 levels of voltage digitization. The reference voltage is provided by the voltage reference module in the MCU. The module provides three selections of voltages: 1.2, 1.45 and 2.5 V. For example, if 2.5 V is used, the ADC provides a resolution of 0.15 mV (2.5 / 16384).

Details of the ADC and the temperature sensor can be found in the MCU datasheet (Sections 5.25.7 and 6.9.8) and technical reference (Section 22.2.10).

Exercise 1.1 Digitize - software trigger

In this exercise, we are going to digitize, under software control, the temperature sensor's output voltage and display it on the debug console.

Create a new project (duplicate from the *empty* example project). Use the C file *Lab7_Ex1_template_adc.c* on Canvas that comes with this PDF. Replace the contents of the main file of the project with this C file's contents. This template program basically digitizes the sensor voltage once and displays the converted digital value.

There are three functional units of the MCU being used in this program. The voltage reference module (REF_A) contains the temperature sensor and provides the voltage reference for the analog-to-digital conversions. The ADC module (ADC14) contains a converter for digitization on a number of channels. The interrupt controller interrupts the processor when the digitization is completed.

To set up REF_A, we use the following functions:

```
REF_A_enableReferenceVoltage();
REF_A_enableTempSensor();
REF_A_setReferenceVoltage(REF_A_VREF1_2V);
```

REF_A_enableReferenceVoltage() enables the REF_A to be used. *REF_A_enableTempSensor()* enables the temperature sensor to be used. *REF_A_setReferenceVoltage()* selects the 1.2-V reference voltage to be used.

To enable ADC14 and route the sensor's output to one of the 32 conversion channels (only 24 channels are implemented on MSP432P401R), we do the following.

```
ADC14_enableModule();
ADC14_initModule(ADC_CLOCKSOURCE_MCLK, ADC_PREDIVIDER_1, ADC_DIVIDER_1,
ADC_TEMPSENSEMAP);
```

ADC14_enableModule() enables the module to be used. *ADC14_initModule()* sets up the ADC clock for digitization. For this exercise, we use the main system clock (*ADC_CLOCKSOURCE_MCLK*) and do not scale it down (both clock dividers are 1: *ADC_PREDIVIDER_1*, *ADC_DIVIDER_1*) for the conversion operations. The function also routes the sensor's output to a conversion channel (which can also be used for an external signal). The last argument routes the internal temperature sensor's output to the ADC.

The ADC supports several resolutions: 8, 10, 12 and 14 bits. To set to use 14-bit, we do the following.

```
ADC14_setResolution(ADC_14BIT);
```

For this exercise, we only do one single conversion at a time with ADC14. When the conversion is done, ADC14 places the digitized value (conversion result) in one of the 32 registers in the module. These are configured with the following function call.

```
ADC14_configureSingleSampleMode(ADC_MEM0, false);
```

The second argument of *ADC14_configureSingleSampleMode()* determines if we want to repeat the conversion or not. In this case, we set it to *false* to disable the repeat action. The digitized value is stored in register *ADC_MEM0*.

The range of the reference voltage can be expressed with two end points: positive (higher) and negative (lower). We use an internal reference voltage source as the positive end and ground (V_{SS}) as the negative end (*ADC_VREFPOS_INTBUF_VREFNEG_VSS*). The channel to be used is number 22 (*ADC_INPUT_A22*). These are set up using the following function.

```
ADC14_configureConversionMemory(ADC_MEM0, ADC_VREFPOS_INTBUF_VREFNEG_VSS,
ADC_INPUT_A22, false);
```

The last argument determines whether to use differential mode at the input or not. In our case, the input is single-ended mode, so the argument is *false*.

The digitization process consists of two stages. The first stage is sample-and-hold and the second is conversion (quantize-and-encode). The sample-and-hold time allows the input signal to be captured and held stable before the actual conversion to digital form can be done. The sample-and-hold time is programmable and the actual length depends on the nature of the input signal. It is a successive approximation ADC, so the conversion time is fixed. For the 14-bit conversion that we are using, it takes 16 clock cycles. The digitization timing is determined by the following function calls.

```
ADC14_setSampleHoldTime(ADC_PULSE_WIDTH_192, ADC_PULSE_WIDTH_192);
ADC14_enableSampleTimer(ADC_MANUAL_ITERATION);
ADC14_enableConversion();
```

ADC14_setSampleHoldTime() sets the sample-and-hold time for all channels be 192 clock cycles. *ADC14_enableSampleTimer()* sets the digitization process to be under software (or manual) control; it can be all done automatically and repeatedly with additional internal circuitry. *ADC14_enableConversion()* enables the conversion to occur.

The end of digitization will be signaled by an interrupt. The following function calls set up ADC14 to generate such interrupt and allow it to be received by the processor.

```
ADC14_enableInterrupt(ADC_INT0);
Interrupt_enableInterrupt(INT_ADC14);
Interrupt_enableMaster();
```

Finally, we trigger the entire digitization process by writing to a control register with the following function call.

```
ADC14_toggleConversionTrigger();
```

When the conversion completes, the interrupt service routine *ADC14_IRQHandler()* will be invoked. It makes sure the interrupt indeed comes from the intended ADC source and then sets the global flag *adc_done* to notify *main()* that the conversion is done.

```
void ADC14_IRQHandler(void)
{
    uint64_t status;
    status = ADC14_getEnabledInterruptStatus();
    ADC14_clearInterruptFlag(status);

    if (status & ADC_INT0)
    {
        adc_done = true;
    }
}
```

For this exercise, add a loop in *main()* to continually digitize the sensor output voltage, display the digitized value and the corresponding voltage in mV. You may want to add some delay, like 500 ms, in the loop to slow down the display a bit.

Lab Report Submission

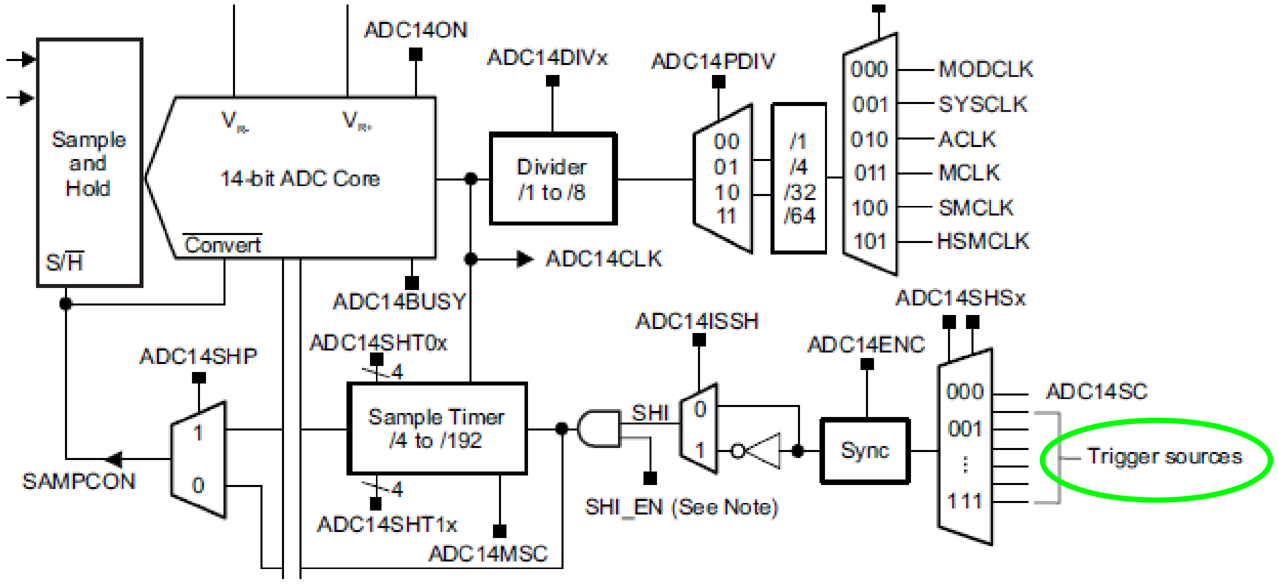
1. Provide a sample of the debug console outputs with the digitized values and voltages in mV.

2. If we lower the resolution to 12 bits, what changes do you expect to see on the debug console outputs? Will the digitized values and voltages go up or down? Provide an explanation on your expectation.
3. Provide the program listing in the appendix.

Exercise 1.2 Digitize - timer trigger

In this exercise, we are going to do exactly what the previous exercise does, but the timing will be controlled by a timer, namely Timer_A. Quite often, we need to digitize an analog signal at a precise interval and at a fast rate. With a hardware timer, we can easily achieve that.

The following diagram (from Figure 22-1, Precision ADC Block Diagram, of the MSP432 technical reference), shows that the ADC can accept six "Trigger sources" outside the ADC module for triggering purpose.



The MSP432 datasheet documents what are implemented for the trigger sources in the MCU. The following table (part of Table 6-46, TA0 Signal Connections, in the datasheet) shows Timer_A A0's Capture/Compare Block 1's output is routed to the Trigger Source 1 of the ADC, and GPIO P2.4 as well.

DEVICE INPUT PIN OR INTERNAL SIGNAL	MODULE INPUT SIGNAL	MODULE BLOCK	MODULE OUTPUT SIGNAL	DEVICE OUTPUT PIN OR INTERNAL SIGNAL
P7.1/PM_C0OUT/PM_TA0CLK	TACLK	Timer	N/A	N/A
ACLK (internal)	ACLK			
SMCLK (internal)	SMCLK			
C0OUT (internal)	INCLK			
P7.3/PM_TA0.0	CCI0A	CCR0	TA0	P7.3/PM_TA0.0 TA0_C0 (internal)
DV_{SS}	CCI0B			
DV_{SS}	GND			
DV_{CC}	V_{CC}			
P2.4/PM_TA0.1	CCI1A	CCR1	TA1	P2.4/PM_TA0.1
ACLK (internal)	CCI1B			TA0_C1 (internal)
DV_{SS}	GND			Precision ADC (internal)
DV_{CC}	V_{CC}			ADC14SHSx = {1}

The example project,
C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\nortos\MSP_EXP432P401R\driverlib\adc14_single_conversion_repeat_timera_source\ccs, illustrates how to set up various modules to do the timer triggering. Basically, it sets up the A0 timer to count in Up mode and Compare register 1 (TA0CCR1) to generate a pulse train to trigger an ADC sampling on the rising edge of a pulse.

Use the previous project and the example project, and adapt accordingly, to produce the same results in the previous exercise. Note that the example project uses ADC input channel A0, but the temperature sensor uses A22. Define the sampling frequency to use with following statement:

```
#define SAMPLING_FREQUENCY 100       // Sampling frequency in Hz
```

Compute the necessary parameters in the program based on this constant and the current frequency of the system (do not hard-code the system clock frequency or its period). For more precise timing, use SMCLK (not ACLK) for the timer module setup. The duty cycle of the pulse train is not important here. So, use something reasonable.

Do not enter sleep mode in the control loop in *main()* (as it is done in the example project). There is no triggering action in the control loop because it is now all done by the timer. Since we are sampling at a much higher rate than the rate to print to the console, the ISR can just update a global variable to contain the latest digitized value, and the control loop just uses it when needed.

Before the control loop is entered, do a quick measurement on the sampling frequency to verify that the ADC is indeed sampling at the desired rate. For example, we can measure how long it takes to collect 10 samples with the Timer32 timer. Print the measured frequency to the debug console. If the measurement is implemented properly, the measured frequency should be very close to the defined macro *SAMPLING_FREQUENCY* because the timing is all controlled by hardware; the error should be less than 0.01 Hz. Note that there are two software "tasks" going on at the same time here. One is the ISR being invoked periodically to update the global variable containing the latest digitized value, and the other is *main()* doing the measurement. In order to measure the sampling rate accurately, we need to pay attention to when the digitization cycle starts or ends.

After the quick measurement is done, enter the forever control loop to show the digitized value and voltage at a regular interval. Note that the display frequency in the control loop (the foreground task) is much lower than the sampling (the background task) frequency.

Do not use the MAP_Interrupt_enableSleepOnIsrExit() call in your program as it is done in the example project so that the control loop in *main()* can continue to run after an interrupt is processed. Also do not use *MAP_CS_initClockSignal()* as it will change SMCLK to a much lower value.

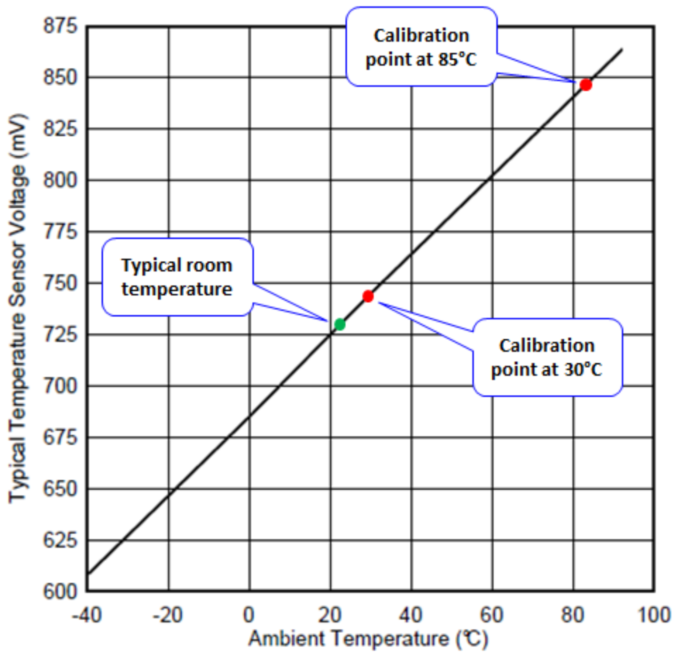
Lab Report Submission

1. Report the measured sampling frequency. Explain how the sampling frequency measurement is done precisely. How do you determine the beginning of a sampling operation?
2. Provide a sample of the debug console outputs with the digitized values and voltages in mV. They should match what you got in the previous exercise.
3. Provide the program listing in the appendix.

Exercise 1.3 Temperature

Once we learn how to convert an analog signal to a digital form, we can transform it to something that has a physical meaning. In this exercise, we are going to convert the digitized value to temperature in Fahrenheit using some calibration data stored in the ROM.

The picture below shows a typical linear relationship between the temperature and the temperature sensor's output voltage.



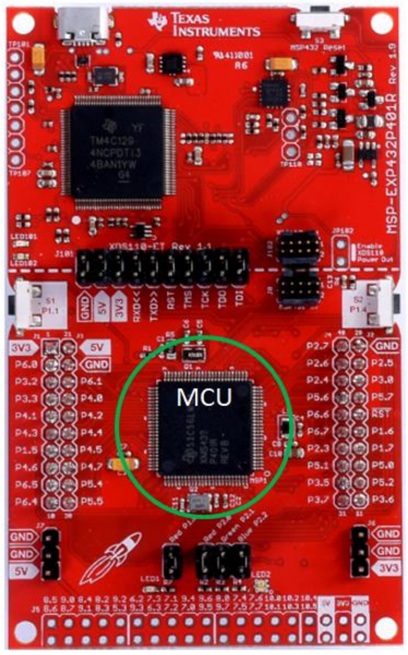
Such linear relationship is defined by two calibration points on the curve. Data of these two calibration points are stored in the Device Descriptor Table. The calibration data are stored in the "ADC14 Calibration" section of the table. The DriverLib function *SysCtl_getTempCalibrationConstant()* provides access to those calibration data.

Each calibration point is the temperature sensor output voltage's digitized value at a specific temperature with a specific reference voltage applied to the ADC. In the final stage of the MCU manufacturing process, the factory would place the chip in a constant temperature environment and record the digitized value at different reference voltages. The values are then burned to the ROM. As shown on the graph, two temperature points are used, one at 30°C and another one at 85°C.

For this exercise, duplicate the project in Exercise 1.1 or 1.2 and expand it to do more work. Since the voltage reference being used is 1.2 V, pick out the calibration data for such voltage from the Device Descriptor Table. Use the calibration data and the newly digitized value to compute the temperature of chip, which should be close to the ambient temperature because the MCU is not very busy. Display the temperature measured in Fahrenheit repeatedly. Show the temperature value with two decimal places of precision. As in the previous exercise, you may want to add some delay to slow down the display.

The sensor is sensitive to its environment. If we touch the surface of the MCU (the circled chip in the figure below), we should get a higher reading because the finger's temperature is higher than the ambient. If we rub our finger for a few seconds before touching the chip, we should get an even higher reading.

In Lab 5, we measured the **no-touch** frequency of the oscillator formed at P4.1 when the capacitive touch feature was enabled. Investigate the effect of temperature on the oscillation frequency. We can run the program we built in Lab 5 to see how the frequency changes when we touch the MCU (do not touch P4.1) to change its temperature. Note that there is no need to modify the program for this exercise; we just re-run an old program.



Lab Report Submission

1. Provide a sample of the debug console outputs. Highlight the temperature readings of touch and no-touch conditions (on the top surface of the MCU chip). Alternatively, you can make a temperature plot to show those conditions using the data on the debug console. With the plot, we can also gain additional insights on how the temperature changes over time due to a better visualization of the data.
2. Report the relationship between the no-touch oscillation frequency at P4.1 and the MCU's temperature. Also provide some supporting data.
3. Provide the program listing in the appendix.

3. Exercise 2 TI-RTOS

TI-RTOS is a real-time operating system for TI MCUs. It consists of a real-time multitasking kernel and libraries for controlling on-chip devices and commonly used functions. It allows software developers to focus on the embedded application development, rather than the details of the underlying hardware and management of the system resources. It also makes programs developed for one TI MCU portable to other TI MCUs.

Some notable information resources for TI-RTOS:
TI-RTOS for MCUs: https://www.ti.com/tool/TI-RTOS-MCU.

TI-RTOS User’s Guide: https://www.ti.com/lit/ug/spruhd4m/spruhd4m.pdf.
TI-RTOS Kernel User’s Guide: https://www.ti.com/lit/pdf/SPRUEX3V.

CCS Version 12.8.0 does not support the MSP432P401R MCU to build applications to run under TI-RTOS. Install CCS Version 12.6.0. Both versions can co-exist, so there is no need to uninstall the existing one. And they can use the same workspace.

Exercise 2.1 Multitask programming to blink LEDs

In this exercise, we are going to create two tasks to control two LEDs independently.

We will import a “skeleton” project,
C:\ti\simplelink_msp432p4_sdk_3_40_01_02\examples\rtos\MSP_EXP432P401R\sysbios\hello\tirtos\ccs, and modify the main program file *hello.c* for this exercise. After importing the project, rename the project and main file to something meaningful. Use the C file, *Lab7_Ex2_template_rtos.c*, on Canvas that comes with this PDF. Replace the contents of the main file of the project with this C file’s contents. This template program basically creates two small tasks to print a simple message to the debug console.

On a Mac machine, the first time to build a new TI-RTOS project or rebuilding an existing project after cleaning it may fail. If CCS returns an “[build-1777670849-inproc] Error 1” message, retry building and it should succeed. Also, the project pathname cannot have a space in it; otherwise, building a TI-RTOS project will fail. Therefore, make sure there is no space in the CCS project workspace folder name.

In all the previous labs where we did not use a RTOS, we put most of the work to be done in *main()*. Under a real-time operating system (RTOS), things are done quite differently. Work to be done is organized as tasks. Each task is supposed to have its unique functionality. Under TI-RTOS, tasks are primarily defined as functions in the program. The main purpose of *main()* is to create the tasks to be executed by the RTOS. Specific device setup should be done in the task function that controls or uses the device.

We can think of a task as a somewhat independent mini-program within the application. It has its own stack for calling functions and storing local variables. It does share system resources with other tasks, including the processor, memory, timers, etc. However, the programmer can implement a task as if it is the only program running in the system; the RTOS switches from one task to another for execution without a task being aware of that.

Build and run the template program. We should see two text strings on the debug console.

Modify the task functions (do not change anything else) so that one task blinks the red LED (LED1 on the LaunchPad) and the other task blinks the green LED (part of LED2 on the LaunchPad). Blink the red LED at a rate of 2 Hz and the green one at 0.5 Hz.

Both tasks should look almost identical, except for the different GPIO parameters for the LEDs and control parameters. In each task function, set up the GPIO port at the beginning and enter a forever loop to turn the LED on and off repeatedly. Do not place the device setup code in *main()* because its main purpose is to set things up for the RTOS to run tasks, not to do any device-specific tasks. Use DriverLib functions to set up and control the LEDs, just like what we have done in previous labs.

For timing purpose, use the RTOS delay function, *Task_sleep(time_ms)*. Do not use *__delay_cycles()*, your own delay function or any others'. Do not use any hardware timer/counter, the way you may have done in previous labs. Instead, use services provided by the RTOS. For example, *Task_sleep(1000)* will allow the RTOS to suspend the task for 1000 ms. During the sleep time, the RTOS can run other tasks. After the sleep time expires, the suspended task will resume execution.

Lab Report Submission

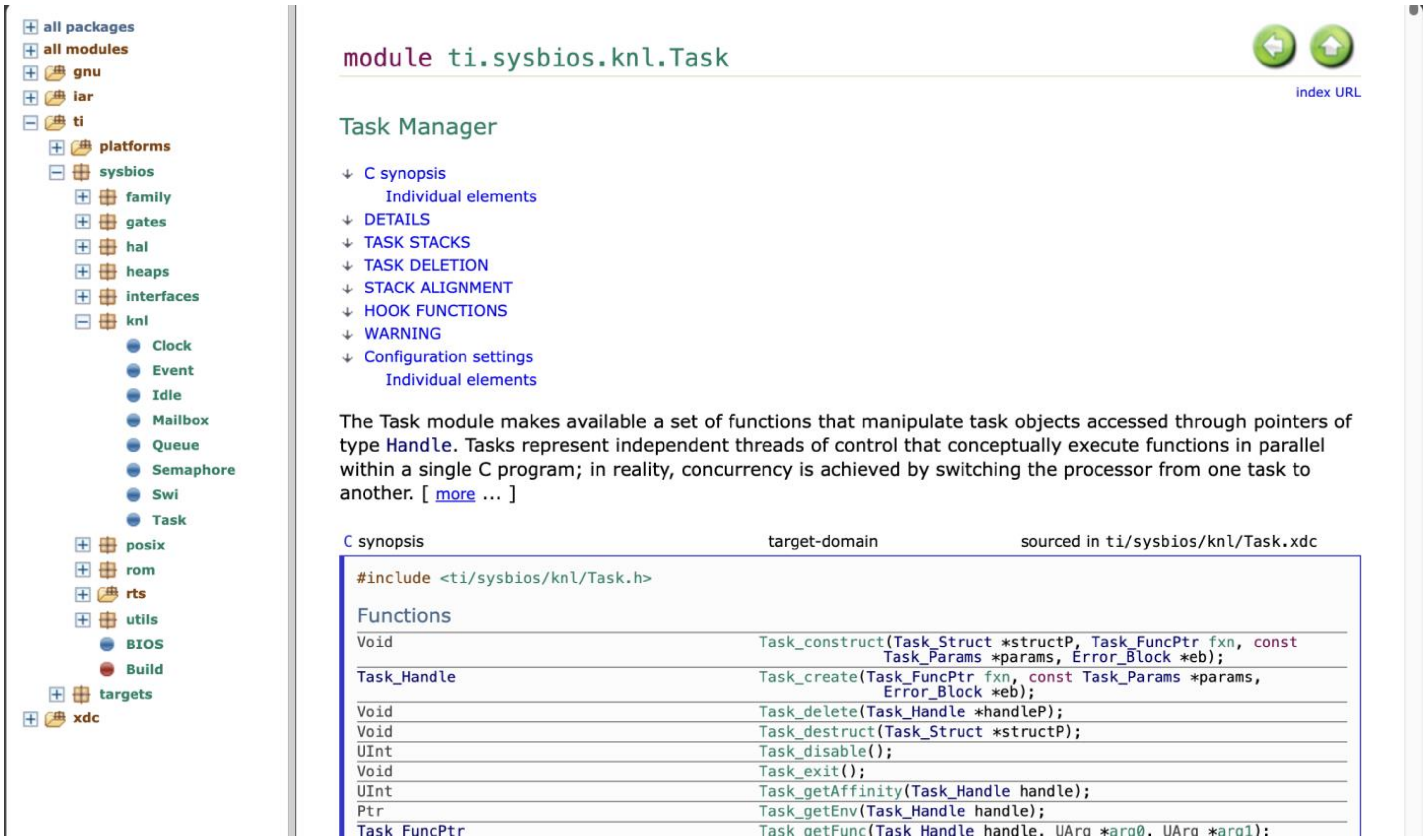
1. Report the delay value used for each LED.
2. With the code of a task function, explain how you control the blinking.
3. Provide the program listing in the appendix.

Exercise 2.2 Single task function

In this exercise, we are going to create multiple tasks with the same task function. Basically, we create the same application as in the previous exercise. Instead of using two separate task functions, we will use one single task function for both tasks. Tasks running under the RTOS can share code in the memory. In the previous exercise, the device to be controlled and the blinking frequency are hard-coded in the task functions. That is no longer the case in this exercise. We will have two tasks executing exactly the same instructions.

When *main()* creates the tasks, it will need to provide the device information (LED's port number and pin number) and the control parameter (blinking frequency or delay) to the task function. When the task function starts up, it knows which device to set up and the specified timing to act on it. The data structure *Task_Params* (of TI-RTOS) provides two fields for passing "arguments" to the new task to be created. Consult the task-creation API on how to use them. Both arguments are basically 32-bit words. We may not need to use both of them for this exercise.

The TI-RTOS documentation and API (Application Programming Interface) references are available in the locally installed SDK package. We can use a web browser to bring up this webpage (on Windows) C:\ti\simplelink_msp432p4_sdk_3_40_01_02\docs\tirtos\sysbios\docs\cdoc\index.html or (on macOS) /Applications/ti/simplelink_msp432p4_sdk_3_40_01_02/docs/tirtos/sysbios/docs/cdoc/index.html to examine the API. To find the information about task management, expand the tree on the left pane to Task as shown in figure below. We can click on the function or data-structure name to get more details.



module ti.sysbios.knl.Task

Task Manager

- C synopsis
  - Individual elements
- DETAILS
- TASK STACKS
- TASK DELETION
- STACK ALIGNMENT
- HOOK FUNCTIONS
- WARNING
- Configuration settings
  - Individual elements

The Task module makes available a set of functions that manipulate task objects accessed through pointers of type `Handle`. Tasks represent independent threads of control that conceptually execute functions in parallel within a single C program; in reality, concurrency is achieved by switching the processor from one task to another. [more ...]

C synopsis target-domain sourced in ti/sysbios/knl/Task.xdc

```
#include <ti/sysbios/knl/Task.h>
```

Functions

Void	Task_construct(Task_Struct *structP, Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb);
Task_Handle	Task_create(Task_FuncPtr fxn, const Task_Params *params, Error_Block *eb);
Void	Task_delete(Task_Handle *handleP);
Void	Task_destruct(Task_Struct *structP);
UInt	Task_disable();
Void	Task_exit();
UInt	Task_getAffinity(Task_Handle handle);
Ptr	Task_getEnv(Task_Handle handle);
Task_FuncPtr	Task_getFunc(Task_Handle handle, UArg *arg0, UArg *arg1);

Duplicate the project created in the previous exercise. Replace the two task functions with one common function. Modify how tasks are created in *main()*. The common task function shall not contain (be hard-coded with) any device or control information. There are many methods to pass those information to the task function when a task is being created. We can create a data structure in the global memory that has all the information necessary for both tasks. Then we pass a unique reference to the data structure during task creation. In the task function, before the forever control loop is entered, print to the console the device and control information for verification purpose. Also print the reference information that we can identify a specific task.

Lab Report Submission

1. List the common task function and the relevant code snippet of task creation in main(). With the code, explain how a task gets the necessary information to control a LED.
2. Provide the console outputs.
3. Provide the program listing in the appendix.

4. 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.

5. 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.