San José State University
Department of Computer Engineering
CMPE 146-03, Real-Time Embedded System Co-Design, Fall 2020

Lab Assignment 7

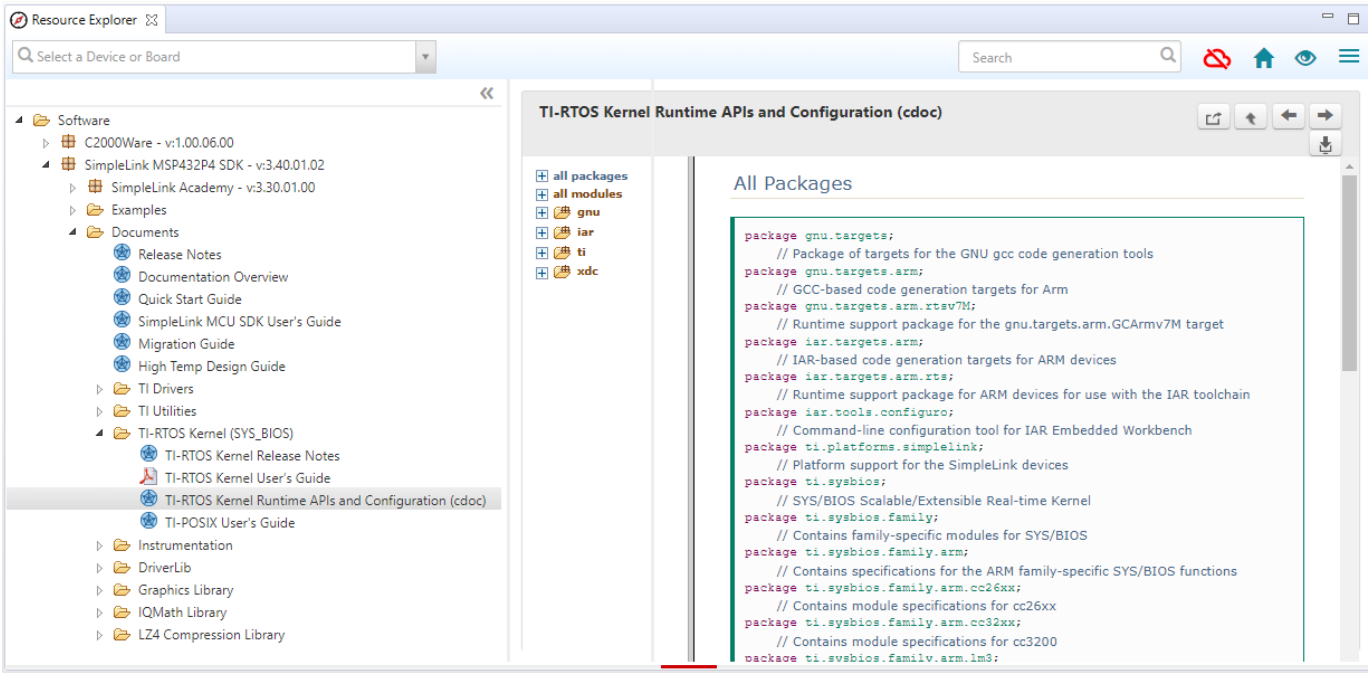
Due date: 11/22/2020, Sunday

1. Description

In this assignment, we will be familiarized with TI's real-time operating system, TI-RTOS. We will create tasks to do some work. We will also use the TI driver to communicate with an external terminal program through the MCU's UART.

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/ug/spruex3u/spruex3u.pdf.
Most of the documentation and API references are also available on Resources Explorer on CCS:



2. Exercise 1

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

We will import a "skeleton" project, /Software/SimpleLink MSP432P4 SDK…/Examples/Development Tools/MSP432P401R LaunchPad – Red 2.x (Red)/TI-RTOS Kernel (SYS_BIOS)/hello/TI-RTOS/CCS Compiler/hello, 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_Ex1_template.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.

In 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 purposes of *main()* are to initialize the hardware and create tasks to be executed by the RTOS.

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.

The following are excerpts from the template file, *Lab7_Ex1_template.c*, to illustrate the steps to create the two tasks in the program.

```
25  Task_Struct task1Struct, task2Struct;
26  Char task1Stack[TASKSTACKSIZE], task2Stack[TASKSTACKSIZE];
27
28  int main()
29  {
30      /* Construct BIOS objects */
31      Task_Params taskParams;
32
33      /* Call driver init functions */
34      Board_init();
35
36      /* Construct task threads */
37      Task_Params_init(&taskParams);
38      taskParams.stackSize = TASKSTACKSIZE;
39      taskParams.stack = &task1Stack;
40      Task_construct(&task1Struct, (Task_FuncPtr)task1, &taskParams, NULL);
41
42      taskParams.stack = &task2Stack;
43      Task_construct(&task2Struct, (Task_FuncPtr)task2, &taskParams, NULL);
44
45      BIOS_start();    /* Does not return */
46      return(0);
47  }
48
49  Void task1(UArg arg0, UArg arg1)
50  {
51      printf("Task1\n");
52  }
53
54  Void task2(UArg arg0, UArg arg1)
55  {
56      printf("Task2\n");
57  }
58
```

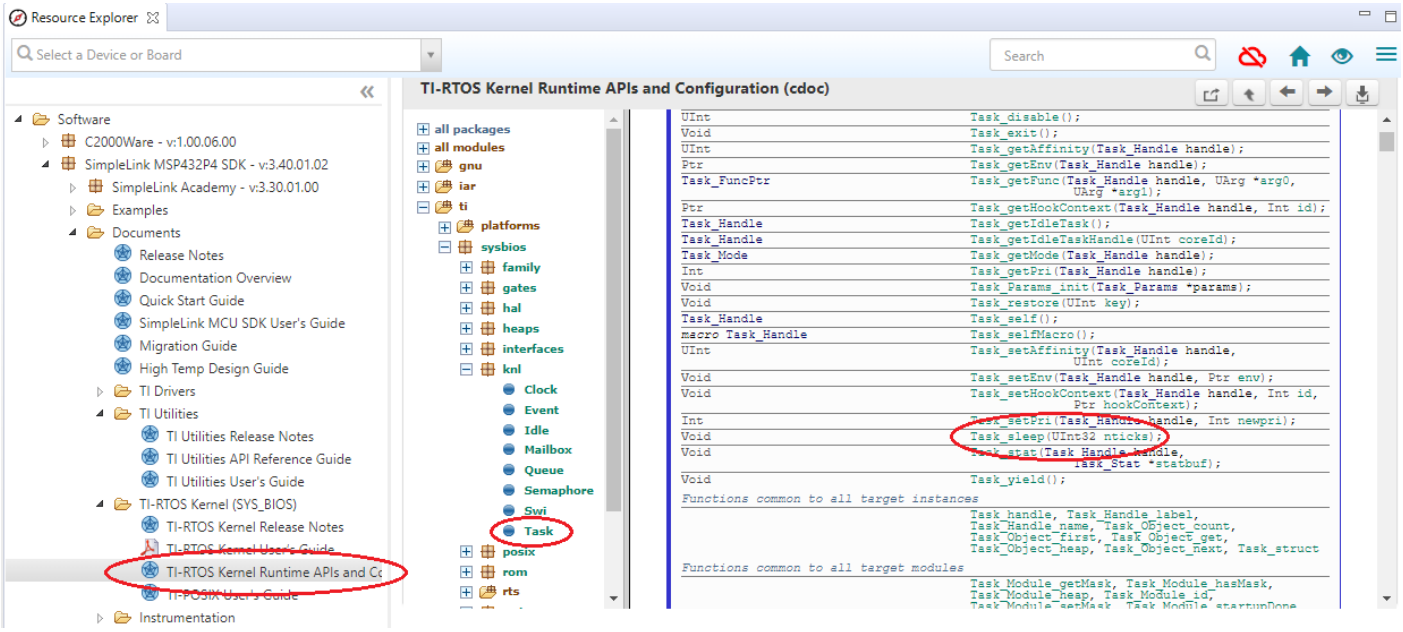
Line 25 declares two data structures for the tasks. The RTOS manages the tasks with the data structures. Each task has its own data structure.
Line 26 defines two stacks to run the tasks. Each task has its own stack. The stack size is 2048 bytes.
Line 31 declares a parameter data structure for creating a task. It contains the parameters the application conveys to the RTOS to create a task.
Line 37 initializes the parameter data structure to contain default values.
Lines 38-39 set up the parameters for the task we are going to create. We specify the stack size and where the stack is.
Line 40 creates the first task to be run later in the system. The task's entry point is the function *task1()*.
Line 42-43 creates the second task. The parameters are same as the first task's, except where the stack is and the entry point of the task.
Line 45, *BIOS_start()* jumps to start the operating system and it will not return to *main()*. The OS will schedule the tasks created and run them according to the scheduling algorithm.
Lines 49-57 are the task functions.

Build and run the template program. We should see two messages 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 1 Hz and the green one at 0.25 Hz.

Both tasks should look almost identical, except for different parameters for LEDs. 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. Use DriverLib functions to set up and control the LEDs, just like what we have done in other labs.

For timing purpose, use the RTOS delay function, *Task_sleep(time_ms)*. Do not use your own function or others'. For example, *Task_sleep(1000)* will allow the RTOS to suspend the task for 1000 ms. During the sleep time, the RTOS will use it to run other tasks. After the sleep time expires, the suspended task will resume execution. Details of the RTOS APIs can be found on the Resource Explorer. For example, the following picture shows the path to the *Task_sleep()* function.



Lab Report Submission

1. List the task functions.
2. Explain how you control the blinking and how you compute the arguments for the delay function.
3. Provide the program listing in the appendix.

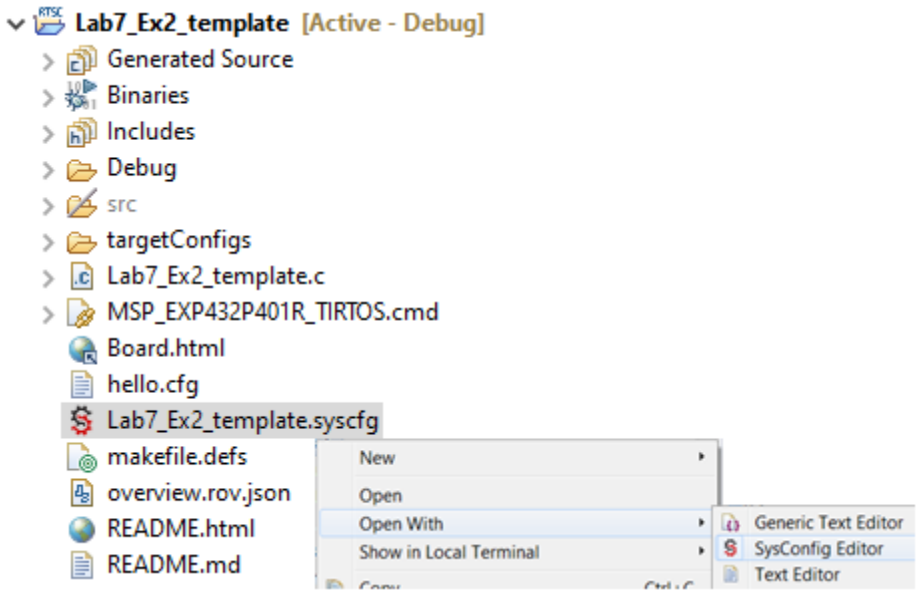
3. Exercise 2

In this exercise, we are going to display the state of an input push button on a terminal window. The program will sample the state of the input periodically and send a message to the terminal through a UART.

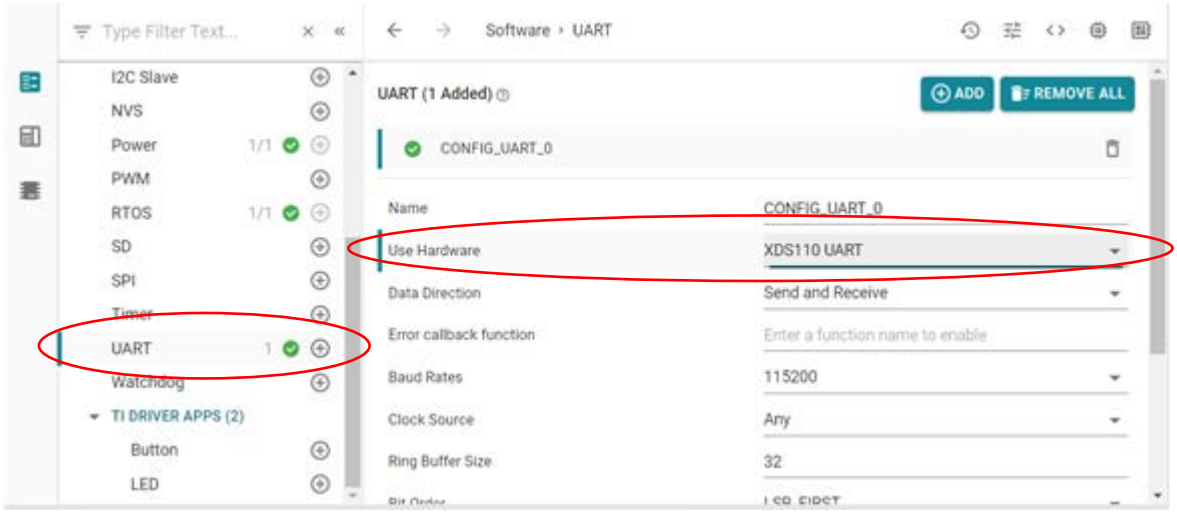
Install a terminal program on your laptop or PC if you have not done so before. There are many such tools available from the Web. For example, you can download PuTTY from https://www.puttygen.com/download-putty. (CCS has a terminal window function, too. But it is more convenient and easier to use a standalone program. Do not use it for this exercise.)

Duplicate the *hello* example project as in the previous exercise. Use the C file, *Lab7_Ex2_template.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 sends a number to the UART periodically.

For this exercise, instead of DriverLib, we will use TI-RTOS's UART driver for the communication with the outside world. Before we can use the driver, we need to add it to the project. On the project pane, locate the file with extension *.syscfg*. Right-click on it and open it with SysConfig Editor.



Scroll to the UART section on the left pane. Click on the circled + symbol. Set "Use Hardware" to XDS110 UART.

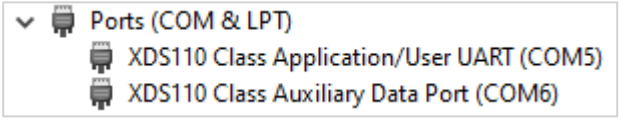


This template C file is very similar to the one used in Exercise 1. We just have Task 1 to handle the UART communication.

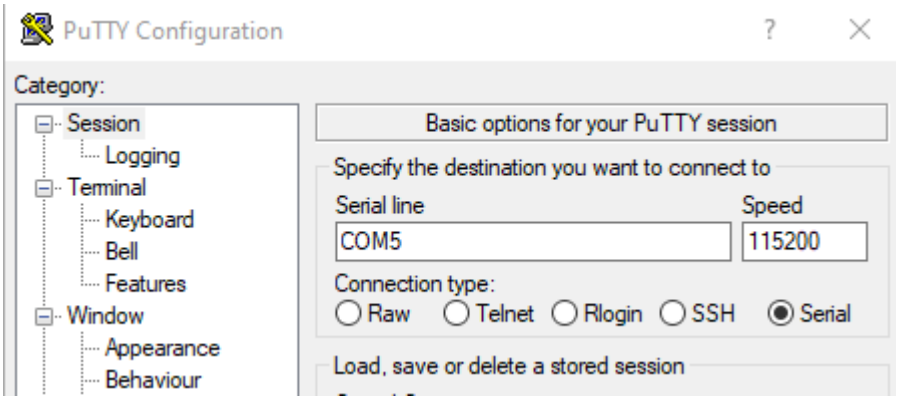
```
52  Void task1(UArg arg0, UArg arg1)
53  {
54      printf("Task1\n");
55      UART_Handle uart;
56      UART_Params uartParams;
57
58      UART_init();    // Driver init
59
60      // Set up communication parameters and open the device
61      UART_Params_init(&uartParams);
62      uartParams.readEcho = UART_ECHO_OFF;
63      uart = UART_open(CONFIG_UART_0, &uartParams);
64
65      if (uart == NULL) {
66          printf("Failed to open UART.\n");
67          while (1);
68      }
69
70      unsigned int count = 0;
71      char buffer[10];
72      while (1) {
73          sprintf(buffer, "%d ", count++);
74          UART_write(uart, buffer, strlen(buffer));
75          Task_sleep(1000);
76      }
77  }
```

Line 58 initializes the UART driver.
Line 61 initializes the parameter data structure. It sets up the default values, something that we can modify with the SysConfig Editor.
Line 62 turns off the automatic read-echo feature so that when we type something on the terminal, the driver will not send back whatever we typed.
Line 63 gets a "handle" of the UART from the driver so that we can use it to access the UART. A handle is basically a control object's ID maintained by the RTOS. From that point on, all accesses to the UART are done with this handle.
Line 74 sends out a message to the UART. We have a forever *while* loop in the task to send out a number once every second.

Start up the external terminal program. Use the baud rate of 115,200. On a Windows machine, the COM port is probably COM5. Consult Device Manager if you want to be sure or the communication does not work.



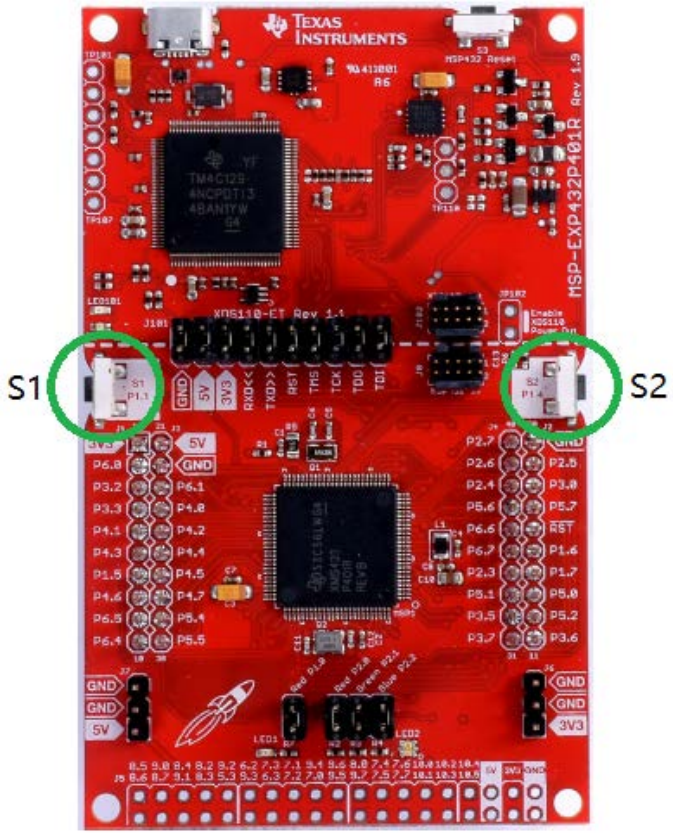
If you use PuTTY, the basic setup configuration screen may look like this:



Build and run the program. You should see a series of numbers being displayed on the terminal window.

Exercise 2.1

In this exercise, we are going to read and display the state of the left push button (S1) on the LaunchPad.



Modify the template C file. At the beginning of *task1()*, set up the GPIO port to be an input port with pull-up resistor. Use DriverLib functions only to do the setup and reading the push button's state. When we press the button, we should read a '0'; it is a '1' when not pressed. Read the state every 100 ms. Same as the previous exercise, only use *Task_sleep()* to control the timing. If the switch is pressed, send out a string "S10 "; otherwise, send out "S11 ". "S1" refers to the name of the button.

There are two push buttons on the LaunchPad. The port number and pin number of the push buttons can be found in the LaunchPad's schematic. There are example projects in Resource Explorer that demonstrate how to get access to the buttons.

Lab Report Submission

1. List the task function.
2. Show a screenshot of the terminal window, showing the pressed and not-pressed messages.
3. Provide the program listing in the appendix.

Exercise 2.2

Most of the time, the button's state remains unchanged. In this exercise, we are going to do some optimization on the communication. We only send out messages when really needed. Here are the additional requirements:

1. In order to avoid sending the same message again and again (wasting communication bandwidth), if the new state is same as the old one, do not send the message.
2. Make an exception to #1 above. It must send the message if nothing has been sent in 5 seconds. This is to make sure that the board is still alive and the communication channel is still okay.

Expand what you have done in the previous exercise. The update to the terminal should be much less frequent now. If the push button is pressed or released, the state is immediately updated to the terminal. Otherwise, the update only occurs once every 5 seconds.

Lab Report Submission

1. List the task function.
2. Show a screenshot of the terminal window, showing messages from different conditions: button pressed, button released, long duration (> 5 s) with no state change.
3. Provide the program listing in the appendix.

Exercise 2.3

In this exercise, we are going to do same thing with the right push button (S2) on the LaunchPad.

Expand the previous exercise. Set up *task2()* to handle S2. When sending out message, use "S20 " or "S21 " to notify the source is from S2.

Both tasks need to get access to the same device, namely the UART. Note that setting up the UART can only be done once, i.e., UART_open() cannot be called multiple times with the same parameters. Some special handling needs to be put in place to resolve the issue.

Lab Report Submission

1. List the task functions.
2. Show a screenshot of the terminal window, showing messages from different conditions: button pressed, button released, long duration (> 5 s) with no state change. Explain how you resolved the issue of UART initialization.

3. Provide the program listing in the appendix.

Exercise 2.4

In this exercise, we are going to control the blue LED (part of LED2 on the LaunchPad) from the terminal program.

Expand the program from the previous exercise. Create a new task, *task3()*, to read from the UART and take appropriate actions according to what is received. If we receive a '1', we turn on the blue LED; if we receive a '0', we turn it off; otherwise, we do nothing. To read from the UART, use the function *UART_read()*.

Use the same way to set up the port to control the LED as in Exercise 1. If the keyboard key '1' is pressed on the terminal window, the LED should be turned on; the '0' key should turn it off. Also, make sure all three tasks work properly together.

Lab Report Submission

1. List the *task3* function.
2. 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 source code in your report. As for the report contents, do not use screenshots to show your codes. In the report body, 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.

5. Grading

Lab 7 Rubric

Criteria	Ratings			Pts
Exercise 1	**3 pts** **3** Correct implementation of entire program. Complete program listing in appendix.	**2 pts** **2** Correct implementation of task functions. Provided proper explanations on control and computation of delay.	**0 pts** **0** Not attempted or reported.	3 pts
Exercise 2.1	**2 pts** **2** Correct implementation of entire program. Complete program listing in appendix.	**1 pts** **1** Showed proper outputs on standalone terminal program when button is pressed and released. Correct implementation of task functions.	**0 pts** **0** Not attempted or reported.	2 pts
Exercise 2.2	**2 pts** **2** Correct implementation of entire program. Complete program listing in appendix.	**1 pts** **1** Showed proper outputs on standalone terminal program when button is pressed and released. Correct implementation of task functions.	**0 pts** **0** Not attempted or reported.	2 pts
Exercise 2.3	**1 pts** **1** Correct implementation of entire program. Provided explanation on solution of UART init. Complete program listing in appendix.	**0.5 pts** **0.5** Showed proper outputs on standalone terminal program when buttons are pressed and released. Correct implementation of task functions.	**0 pts** **0** Not attempted or reported.	1 pts
Exercise 2.4	**2 pts** **2** Correct implementation of entire program. Complete program listing in appendix.	**1 pts** **1** Correct implementation of task functions.	**0 pts** **0** Not attempted or reported.	2 pts
				Total Points: 10

Grading Policy

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