ECE 315 Computer Interfacing

LAB 1: Interfacing to Input/Output Devices

Winter 2026

DATES

Date	Section	Time	Demo Due Date	Report Due date
03-Feb–2026	H21	2:00-4:50 PM	13-Feb-2025	24-Feb-2025
04-Feb–2026	H31		13-Feb-2025	25-Feb-2025
05-Feb–2026	H41		13-Feb-2025	26-Feb-2025
06-Feb–2026	H51		13-Feb-2025	27-Feb-2025

LEARNING OBJECTIVES

- To gain experience designing a FreeRTOS application with multiple tasks, delays and queues.
- To gain experience interfacing to operator inputs received from a keypad and pushbuttons.
- To gain experience interfacing to outputs sent to a 7-segment LED display on the Zybo Z7 and to the console window in the AMD Viitis IDE on the host PC.
- To implement a simple peripheral control system as an embedded system on the Zybo Z7.

INTRODUCTION

The hardware platform is a Digilent Zybo Z7 development board, which is based on an AMD Zynq-7010 System-on-Chip (SoC) silicon chip. The Zynq-7010 contains two 667-MHz ARM Cortex A9 32-bit CPUs, called CPU0 and CPU1, which can simultaneously execute two independent software systems. As in Lab 0, CPU0 will run the FreeRTOS operating system while the second processor, CPU1, will be left disabled.

The Zybo Z7 board has five Pmod expansion ports, which can be used to enhance the Zybo Z7 by connecting Pmod peripherals. In the lab system, two of the Pmod ports connect to a 2-digit 7-segment LED display (SSD) for output, and a third Pmod port is connected to a 16-button keypad for operator input. There are also four general-purpose pushbuttons on the Zybo Z7 board for input. Two adjacent Pmod headers, labeled JC and JD on the Zybo board, must be connected to the twin plugs of a Pmod 2-digit 7-segment LED display (SSD). The third header on the same edge of the Zybo board, labeled JE, must be connected to the plug of a Pmod 16-button hexadecimal (hex) keypad module. These three headers (JC, JD and JE) are assumed by the software to connect to certain general-purpose input/output signals coming from the Zynq-7010 SoC. Thus if the wrong headers are used to connect the Pmod modules, the software drivers will not work. The lab boards have already been connected properly for you.

Winter 2026

ECE 315 Computer Interfacing

LAB 1: Interfacing to Input/Output Devices

PRE-LAB **Marks 10%**

It is strongly recommended that you completely read this handout and answer the pre-lab questions before coming to your lab session. This will help ensure that you make the most effective use of your time in the lab session.

1. Imagine two different situations where the 7-segment LED display isn't functioning correctly:
    a. The digits are displayed on the incorrect side of the display.
    b. The digits are constantly flickering instead of being perceived as steadily lit.

Identify potential causes for these issues. What might be going wrong in each scenario? Assume that the hardware platform is correctly configured.

2. What are the steps required to establish an interface with a peripheral device on the Zybo Z7 board? This should be described assuming that the hardware platform is already in place and ready for configuration.

3. Create a system architecture diagram for Part 2, including the Zybo Z7 board peripherals, the serial terminal window and the FreeRTOS tasks and others. Be sure to use arrows to show the relationships between components in your diagram.

To answer the pre-lab questions, it is important that you carefully read through the lab manual and review the provided source code files. The answers and necessary understanding lie within the detailed description of the system's operation and the example code. This will not only help you to complete the pre-lab tasks, but also enhance your comprehension and readiness for the practical aspects of the lab.

The SoC hardware configuration and the initial source files must be downloaded from the Lab 1 section of the lab Canvas site. These source files contain the initial application code in C that you will be trying out and then modifying and upgrading to complete the lab exercises.

All parts of the lab must be executed by creating separate application components in Vitis. The provided Pmod keypad driver files, **pmodkypd.c** and **pmodkypd.h,** must be included along with the main source code file for all parts of the lab. All of the required files are downloadable from Canvas.

Note: *You must submit your Pre-Lab work using the appropriate submission link on the Canvas site. Skipping the pre-lab work is likely to prevent you from finishing the exercises within the 3-hour lab time.*

Winter 2026

ECE 315 Computer Interfacing

LAB 1: Interfacing to Input/Output Devices

PART 1 - INTERFACING WITH THE SEVEN SEGMENT DISPLAY AND KEYPAD — Marks 10%

Download the hardware configuration and software project files from the Lab 1 section of the Canvas site. Start Vivado and export the hardware configuration from Vivado to Vitis. Next, create a new platform component named **lab1_platform** and a new application component named **lab1_part1** including the provided **lab1_part1.c** and keypad driver files (**pmodkypd.c** and **pmodkypd.h**) as source files, and then open it.

Examine the given template carefully before making changes to the code. You must include your code in the section that is enclosed in between the following comment line delineators:

```
/********** Enter your code here***********/
    // TODO:
/******************************************/
```

Figure 1: Place Your Code Within These Markers

In this part of the lab you will interface with the seven segment display so that it outputs any digits pressed on the keypad and you will also determine experimentally the minimum delay that eliminates flickering due to an overly low switching frequency between the two digits displays. Your tasks for this part of the lab are:

1. Update the board support package to set the tick rate to 1000 ticks per second:
   a. Select the **lab1_platform** in the Vitis explorer.
   b. Open the platform settings by either clicking the gear icon in the toolbar or opening the **vitis-comp.json** file inside the settings folder inside the platform.
   c. Navigate to: **ps7_cortexa9_0→freertos_ps7_cortexa9_0→Board Support Package→freertos**
   d. Scroll through the FreeRTOS configuration parameters until you locate the parameter labeled **freertos_tick_rate**.
   e. Change the value to 1000 and go back to the **Board Support Package** section of the explorer and click **Regenerate BSP**.
   f. Rebuild the platform component to ensure the modified BSP settings are applied.
2. Declare a macro to return the seven-segment display (SSD) base address.
3. Declare and initialize the SSD device.
4. Define a constant of type **TickType_t** named **xDelay** and initialize it with the value 100 for the SSD delay.
5. Update the values of **previous_key** and **current_key** every time a new key is pressed.

6. Use the **ssd_value** variable to save the binary representation of the pressed key using the provided **SSD_decode** function.
7. Only one digit can be displayed at a time on the seven segment display, but two digits can be effectively displayed if each digit is lit rapidly enough in succession alternating between the left and right digits. In order to do this, every time a new key is pressed, send the currently pressed key **current_key** to the right SSD digit and the previous key **previous_key** to the left digit, in other words the previously entered digit will be shifted to the left digit position. To do this you will need to write **current_key** to the right SSD, pause the execution of the task for a specific amount of time (**xDelay**), then write **previous_key** value to the left SSD and execute another pause.
8. Adjust **xDelay** to determine the minimum delay that eliminates visible flickering on the display. Begin with a large delay value and gradually decrease it while observing the display, identifying the smallest delay at which the display appears stable. Repeat this procedure for both lab partners, as perception of flicker due to persistence of vision may vary between individuals, and record the minimum delay found where flicker is unnoticed.
9. Add a print statement (using the **xil_printf** function) to display the value of the variable status each time that it changes.
10. Show your results to the LI or TA.

Note: 'B' will be displayed as **'b'** on the 7-seg display and **'D'** will be displayed as **'d'**. These segment patterns are used to avoid appearing like **'8'** and **'0'**, respectively, on the 7-segment display.

PART 2 - INTERFACING WITH THE RGB LED AND PUSHBUTTONS — Marks 20%

Create a new application component named **lab1_part2** and copy the code from **lab1_part1.c** as a source file, renaming it to **lab1_part2.c** and open it.

Examine the given template carefully before making changes to the code. You must include your code in the section that is enclosed in between the following comment line delineators:

```
/********** Enter your code here ***********/


/*******************************************/
```

Figure 1: Place Your Code Within These Markers

In this part of the lab, you will learn how Pulse Width Modulation (PWM) signaling can be used to control the brightness of an LED. PWM operates by rapidly toggling the LED on and off at a fixed frequency. The ratio of the on-time to the total signal period, known as the duty cycle, determines the LED's perceived brightness. A greater duty cycle means the LED remains on for a longer portion of each period, causing it to appear brighter. By gradually adjusting the duty cycle (Figure 2), smooth transitions in brightness can be achieved. This principle is fundamental to applications such as fading light effects, controlling the speed of DC motors, and positioning servo motors.

$$duty\_cycle = \frac{Ton}{Ton + Toff}$$

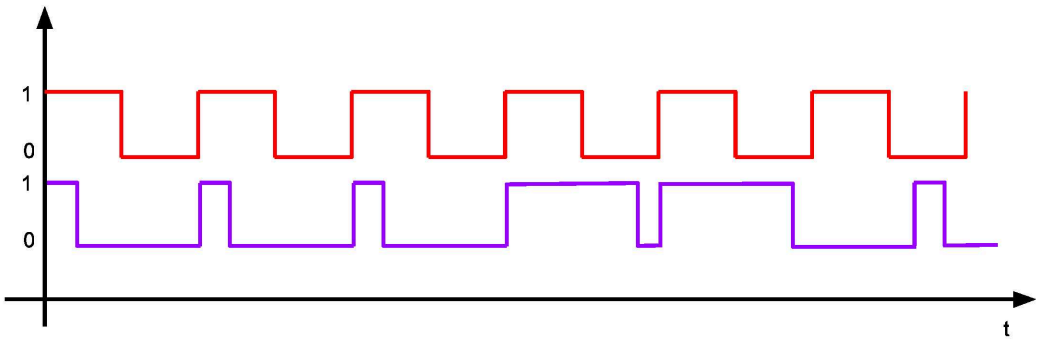

Figure 2: visual representation of different duty cycles

Your tasks for this part of the lab are:

1. Include the file **rgb_led.h** in the source code. This header provides the required macros for RGB color selection, as well as the RGB LED device configuration parameters, including the base address and channel definitions.
2. Declare an instance of type **XGpio** to represent the RGB LED device. This will be used to initialize, configure, and control the RGB LED.
3. Copy the **vRgbTask** from **vRgbTask.c** function into your **lab1_part2c** source file as provided. This function generates a simple square-wave signal on the RGB LED and will serve as the basis for implementing PWM-based brightness control.
4. Declare and initialize the RGB LED device in **main()** and configure its GPIO data direction to output.
5. Use **xTaskCreate** to create a FreeRTOS task for **vRgbTask**.
6. Run the application. Verify that the keypad and seven-segment display operate as in Part 1, and confirm that the RGB LED now blinks as generated by **vRgbTask**.
7. Declare the pushbuttons as an **XGpio** input device. Initialize the device and configure its data direction as input. The base address and channel number can be found in **xparameters.h**.
8. In **vRgbTask**, read the pushbutton input value. If the value is 8, increment the LED period; if the value is 1, decrement the period. This will allow manual control of the blinking frequency. Move the **xil_printf** statement inside these conditional blocks so the period is printed only when it changes. After updating the period, insert a short delay to throttle the adjustment so holding a button does not change the period too rapidly. Ensure the period value cannot become negative.
9. With this implementation, the user can control the on/off period of the RGB LED. Run the application to verify correct operation. Determine the smallest period value for which no visible flickering occurs and record this value. Present your result to the TA or LI for verification.

Up to this point, the RGB LED has been controlled using a fixed on/off timing, resulting in a simple blinking behavior. This functionality will be extended by introducing Pulse Width Modulation (PWM), allowing the perceived brightness of the RGB LED to be controlled by adjusting the duty cycle while keeping the overall signal period constant.

1. Set the PWM period to the value identified in the previous part as the minimum period that avoids visible flickering.
2. Modify **vRgbTask** so that the pushbuttons adjust the duty cycle rather than the overall period. Keep the period constant and use $T = Ton + Toff$; therefore, each button press must update both Ton and $Toff$. Declare **xOnDelay** and **xOffDelay** to store these values and use them in the task in place of the existing xDelay. Handle edge cases to ensure **xOnDelay** and **xOffDelay** are always positive.

3. Run the application and verify that pressing the pushbuttons changes the PWM duty cycle and, consequently, the perceived brightness of the RGB LED. Demonstrate correct operation to the TA or LI.

PART 3 - APPLICATION ARCHITECTURE — Marks 30%

In this part of the lab, the application developed in Part 2 will be refactored to improve structure, modularity, and scalability. Instead of having individual tasks manage multiple peripherals, each peripheral will be handled by its own dedicated task. This approach simplifies maintenance and makes it easier to extend the system with additional functionality. Because tasks will no longer directly access all peripherals, inter-task communication must be introduced. FreeRTOS provides several mechanisms for this purpose; in this lab, queues will be used to safely and efficiently exchange data between tasks.

1. Create a new application component named **lab1_part3** and copy the code from **lab1_part2.c** as a source file, renaming it to **lab1_part3.c** and open it.
2. Create two new task functions named **vButtonsTask()** and **vDisplayTask()**. Do not create the FreeRTOS tasks for them yet.
3. Declare two FreeRTOS queue handles to support inter-task communication using queues. One queue will be used to send data from **vKeypadTask** to **vDisplayTask**, and the other will be used to send data from **vButtonsTask** to **vRgbLedTask**. Name the queue handles appropriately to clearly reflect the source and destination tasks.
4. Inside **main()** create the queues as single-element queues. Select an appropriate data type for the queue element based on the information being exchanged, and restrict the queue length to one element to simplify the design and behavior.
5. Move all code in **vKeypadTask** that is related to updating or controlling the seven-segment display into **vDisplayTask**.
6. Use **xQueueOverwrite** to write the most recent value into the queue, and use **xQueueReceive** in the receiving task to read from it. Use the return value of **xQueueReceive** (i.e. check for **pdTRUE**) to handle the case when a new value has been received.
7. Repeat Steps 5 and 6 for the RGB LED and pushbuttons: move all pushbutton-handling code out of **vRgbLedTask** into **vButtonsTask**, and use **xQueueOverwrite** and **xQueueReceive** to pass data from **vButtonsTask** to **vRgbLedTask** (checking for **pdTRUE** on receive).
8. Draw a system architecture diagram to represent the current configuration of the embedded system. Include the hardware components, software layers, and their interactions (FreeRTOS Tasks, queues, peripherals, etc.) and ask the Teaching Assistants or Lab Instructor for feedback.

REPORT REQUIREMENTS

Cover Page:

- ➔ Include the course and lab section numbers, the lab number, and the due date.
- ➔ List all of the student names in your lab team.

Abstract:

- ➔ Briefly summarize the lab's objectives, the hardware used and the exercises in your own words.

Design Section:

- ➔ Outline your solutions with a concise explanation of your code for each exercise.
- ➔ Include task flow diagrams. Avoid big sections of code snapshots.

Testing Suite:

- ➔ Provide a table with test descriptions, the expected and actual results and the rationale for each test case.

Conclusion:

- ➔ Assess if the given objectives were met and summarize the lab's findings.
- ➔ Discuss any issues if the lab was incomplete or unsuccessful. What were the cause(s) of the issue and how could you resolve them if you had more time?

Formatting and Submission:

- ➔ Submit the report as a single PDF with both the lab number and student names in the filename.
- ➔ Ensure correct spelling and grammar.
- ➔ Use a readable font size (larger than size 10).
- ➔ Diagrams may be hand-drawn with a straightedge and scanned at a 300 pixel resolution.
- ➔ Source code in your report should be syntax-highlighted to enhance readability. Syntax highlighting differentiates code elements such as keywords and strings.
- ➔ Adopt a professional format with consistent headings and subheadings.
- ➔ Only submit your modified source code files in one zip folder. Do not zip the entire workspace.
- ➔ Adhere to the submission deadlines as indicated on the first page of the lab handout.

ECE 315 Computer Interfacing

LAB 1: Interfacing to Input/Output Devices

MARKING SCHEME

Students will be graded based on the form and general quality of the report (clarity, organization, tidiness, spelling and grammar) for the lab. The Pre-lab work is worth 10%, lab demonstrations (Parts 1, 2 and 3) are worth 60%, and the report itself is worth 30%.

Winter 2026

ECE 315 Computer Interfacing

LAB 1: Interfacing to Input/Output Devices

REFERENCES

- Zybo Z7 Reference Manual.
- FreeRTOS Documentation.
- PmodKYPD Reference Manual.
- PMOD Seven Segment display Reference Manual.
- Lab 1 hardware platform and software environment.

APPENDIX

Function name	Purpose
XGpio_Initialize();	Initialize the GPIO device.
XGpio_SetDataDirection();	Set the input/output direction for this specific GPIO channel.
XGpio_DiscreteWrite();	Write the value for this GPIO channel to be displayed.
XGpio_DiscreteRead();	Read the value from the device attached to this GPIO channel.

Table 1: Useful functions for controlling peripherals

Function name	Purpose
xQueueCreate();	Create a queue for inter-task communication
xQueueOverwrite();	Write an item to a queue, overwriting the existing item if it exists
xQueueReceive();	Read an item from a queue; return pdTRUE if an item was received

Table 2: Useful queue functions

Function name	Purpose
xTaskCreate();	Create a FreeRTOS task
pdMS_TO_TICKS()	Convert milliseconds to ticks
xTaskGetTickCount();	The count of ticks since vTaskStartScheduler was called.
vTaskDelay();	When called it specifies a time delay after which the task wishes to unblock relative to the time at which vTaskDelay() is called.

Table 3: Useful task functions