ECE 315 Computer Interfacing

LAB 2: The Zynq-7000 UART Interface

Winter 2024

DATES

Date	Section	Time	Demo Due Date	Report Due Date
27-Feb-2024	D21	2:00 to 4:50 PM	12-Mar-2024	12-Mar-2024 11:00 PM
28-Feb-2024	D31		13-Mar-2024	13-Mar-2024 11:00 PM
29-Feb-2024	D41		14-Mar-2024	14-Mar-2024 11:00 PM
01-Mar-2024	D51		15-Mar-2024	15-Mar-2024 11:00 PM

LEARNING OBJECTIVES

This lab provides experience with FreeRTOS queues and UART interfacing using hashing and a simple proof-of-work mechanism. More specifically, the objectives of this lab exercise are:

- To implement UART interfacing for user input and system output via a serial terminal connection between the PC host and the Zybo Z7 board.
- To understand and apply FreeRTOS queues for inter-task communication.
- To gain experience with interfacing software with hardware using both polling and hardware interrupts.
- To gain experience using queues to decouple the execution of data producer and data consumer tasks.
- To learn the basics of hashing functions and their applications.

INTRODUCTION

In this lab you will use the Zynq-7000 Universal Asynchronous Receiver-Transmitter (UART) Interface in conjunction with the FreeRTOS operating system. This lab exercise is designed to provide a comprehensive understanding of UART communication, especially in the context of real-time operating systems.

The Zynq-7000 SoC (System on Chip) is a powerful platform that combines a high-performance ARM-based processing system with programmable logic. This flexible architecture makes it a convenient choice for a wide range of applications, including those requiring real-time processing and versatile communication capabilities. In this lab, we will focus on two primary subjects:

1. **Implementing UART Communication using Polling:**

In Part 1 of the lab, students will complete a text hashing application. This application employs the widely-used Secure Hashing Algorithm-256 (SHA-256) algorithm for hashing and utilizes the polling method for UART communication.

The focus of this part is on the implementation and comprehension of the polling-based communication using the UART interface in the Zynq-7000. Polling is a technique where the CPU repeatedly checks the status of an external device - a fundamental concept in the design of embedded systems.

2. Developing a UART Driver using Interrupts:

The second part of the lab involves implementing a more sophisticated UART driver using interrupts. This method leverages the interrupt capabilities of the Zynq-7000 to enhance the speed and efficiency of communication. This approach is more complex than polling but offers superior performance in real-time applications.

As with previous lab exercises, we will be utilizing FreeRTOS, a popular real-time operating system, known for its simplicity and robustness. FreeRTOS will provide the necessary multitasking environment to effectively manage the UART communication tasks, demonstrating how real-time systems can handle concurrent operations and resource management.

You will find designated sections marked for code insertion as shown in Figure 1 below. Please ensure that all of the code that you write as part of the exercise is added within these specified comment delimiters. This organization is crucial for maintaining the clarity and structure of the source code and for facilitating both development and subsequent evaluation.

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


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

Figure 1: Place Your Code Within These Markers

Winter 2024

ECE 315 Computer Interfacing

LAB 2: The Zynq-7000 UART Interface

PRE-LAB **Marks 10%**

Carefully read through the lab manual and review the source code files provided on eCLass. Sufficient documentation is provided within the description of the system's operation and the example code. Reviewing this documentation 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. To receive full marks the pre-lab work must include the following:

1. Draw a system architecture diagram for Parts 1 and 2, including the Zybo Z7 board peripherals, the SDK terminal and the FreeRTOS tasks and queues.
2. Create a flowchart for Part 1, illustrating the sequence from user input to output. Include decision points for hashing or verification and their respective outcomes. This flowchart should visually summarize the UART communication and hashing steps in the system.
3. What are the main differences between a polling-based interface and an interrupt-driven interface?

All pre-lab work is to be completed **individually**. Please submit your documents through the designated submission link on the eClass lab site. Ensure your submissions are clear, well-organized, and adhere to the guidelines provided to receive full credit.

ECE 315 Computer Interfacing

LAB 2: The Zynq-7000 UART Interface

Winter 2024

PART 1 - BASIC HASHING SYSTEM USING POLLING — Marks 30%

This exercise is designed to introduce students to UART (Universal Asynchronous Receiver-Transmitter) polling-based interfacing using the Zybo Z7 board. Specifically you will implement a text data hashing and verification system, implemented within FreeRTOS. This approach intended to provide a comprehensive learning experience, combining hardware interfacing with software algorithm design.

In computer engineering, hashing systems are useful for efficient data management and security. Hashing systems use a one-way function that maps data blocks to equal-length binary hashes. Hashing systems are used in various applications such as:

- Secure password storage, where passwords are stored as hashes rather than as plain text.
- Data deduplication in storage systems, where hashing identifies redundant data.
- Implementing hash tables for quick data retrieval.
- Digital signatures and certificates in cybersecurity.
- Version Control Systems: In version control, hashing is used to track and manage changes in documents or code. Each version is assigned a unique hash, allowing for the efficient comparison, retrieval, and management of different versions of files. This is particularly important in software development, where tracking changes and maintaining historical versions of code is required.
- Implementing proofs of work (PoW): PoW is a mechanism that secures networks against fraudulent activities and service abuses. In PoW, participants are required to solve computationally intensive puzzles, verified easily through hashing. This process plays a critical role in achieving consensus in decentralized systems, making hashing indispensable in the architecture of modern digital infrastructures.

Polling is a method, sometimes used in UART communication where the CPU repeatedly checks the status of a peripheral device to see if it's ready for data transmission. Unlike interrupt-driven systems, where the device notifies the CPU when it's ready, polling requires the CPU to actively inquire the status of the device.

In our setup, polling is employed to manage data transmission over a UART interface which supports a serial communication protocol. You'll use specific functions like **XUartPs_ReadReg()** for receiving data and **XUartPs_WriteReg()** for sending data. These functions operate in polled mode, meaning they will block the calling task until the data is successfully received or sent. This method is particularly effective in real-time systems, where timely data processing is crucial.

In the lab, a Proof of Work (PoW) is the hash produced by adding a nonce (an arbitrary number that can be used just once in a cryptographic communication) to a specific string, such that the hash begins with a number of zeros exceeding a predetermined difficulty level. This process, demanding considerable computational effort, demonstrates the hashing function's probabilistic nature and the substantial work needed to find a qualifying hash.

System Operation:

- **Input Task:** This task uses repeated polling of the UART registers for capturing and forwarding user inputs which are the keystrokes on the keyboard that enter text into the terminal window of the SDK. The task captures the user's keyboard input and directs it to the appropriate task for further processing in the embedded system or display.

- **Hashing Task:** Receives data from the input task and executes various operations based on the user-selected options. It:
    - Applies a cryptographic hashing function (SHA-256) to the input and transmits the resulting hash to the Output Task for display.
    - Accepts a provided string and a corresponding hash, then checks to see if the computed hash matches the provided one. The Hashing Task subsequently relays the results to the Output Task, which displays a success message to the user if a match is confirmed, or an error message in the event of a mismatch.
    - Attempts to find a hash that meets a specified difficulty level by adjusting the nonce and iterating the process until successful.

- **Proof of Work Task:** Receives a string, a difficulty level, a nonce, and a hash from the Hashing Task. It validates whether the hash meets the set difficulty. If unsuccessful, the task increments the nonce and returns the data for rehashing. Upon finding a valid hash, it signals success.

- **Output Task:** This function is dedicated to managing the data received from the Input, Proof of Work and Hashing Tasks. It formats the data and sends it out through the UART.

After its initialization, the system offers the user three primary services: Hash Generation, Hash Verification and Generation of a Proof Of Work.

For the Hash Generation service, the user inputs text via the keyboard. The Input Task captures this input and relays it to the Hashing Task, which computes the hash of the received data. The computed hash is then forwarded to the output queue for display in the SDK terminal window.

For the Hash Verification service, the user is required to input a string followed by its corresponding hash. This input string is processed by the Hashing Task, which calculates the hash of the provided string and then compares it with the user-supplied hash. The system then displays a success message if the hashes match, or an error message in case of a mismatch.

For the Proof of Work service, the user submit a string and a difficulty level. The system, starting with a nonce at 0, attempts to generate a hash that has a number of leading zeros at least equal to the difficulty level. The Hashing Task sends this data to the Proof of Work Task, which checks if the hash meets the difficulty. If not, the nonce is incremented, and the process repeats until a valid hash is found, at which point a success message is displayed.

When you start, load the provided resource files into a project and run it using the system debugger configuration and connecting to SDK serial terminal. You should see a menu with three options, like in Figure 2 below

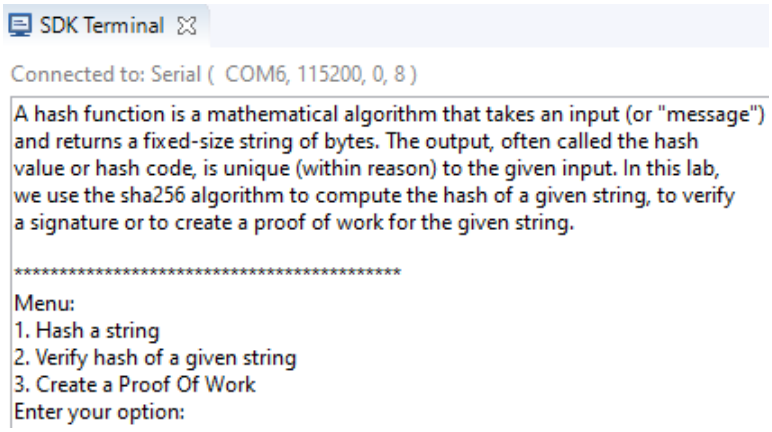

Figure 2: Initial view of the hashing app

However, you will notice that trying to execute any operation results in the app just printing the menu message. This is because the logic for the app is incomplete. It is your job to finish the code so that the app can function properly as described in the system operation.

Your main tasks for this part of the lab are as follows:

1. Declare a queue that will send data to the display task called **xOutputQueue.**
2. Write the body of **vOutputTask** which is to repeatedly poll the output queue for data and send any received bytes to the UART.
3. Write the body of the **printString** function which is to send data to the **outputQueue.**
4. Finish the **receiveInput** function, which is to poll the UART for data and use **printString** to display the chosen option.
5. Write the body of the **hashToString** function to read a BYTE array and turn it into its hexadecimal representation.
6. Use the **hashToString** function inside the hashing task to turn the computed hash into its hex representation.
7. Print the hash using the printString function.
8. Inside the hashing task's 'verify' branch, compare the two hash strings and print a message to inform the user of the result (success or error).
9. At this point your application should be able to support the Hashing and verification services.
10. Create the PoW input and output queues, named xPoWInQueue and xPoWOutQueue, ensuring each queue contains a specific number of elements, to be determined by you. Additionally, the size of each element within these queues should match the size of the ProofOfWork struct..
11. Use the newly implemented queues to add receiving and sending data capability to the Proof of Work task so that it receives data from the Hashing task and after verifying the result (if necessary) send the data back to the Hashing task.
12. Add sending and receive functions to the Hashing tasks so that it can exchange data with the Proof of Work task.
13. Create the PoWTask and give it a tskIDLE_PRIORITY.

Function name	purpose
XUartPs_ReadReg();	Reads a UART register. Returns the values read into an 8-bit variable.
XUartPs_WriteReg();	Writes 8-bit data to a UART register.
XUartPs_IsTransmitFull();	Determine if a byte of data can be sent with the transmitter.
XUartPs_IsReceiveData();	Return TRUE if there is receive data, FALSE otherwise.
xQueueSend();	Sends a message to a queue.
xQueueReceive();	Receives a message from a queue.

Table 1: Useful functions for completing Part 1

PART 2 - INTERRUPT DRIVEN INPUT AND OUTPUT USING THE UART — Marks 30%

Interrupts are a critical feature in real-time systems, allowing the processor to respond immediately to certain events. In UART communication, an interrupt is a signal that temporarily halts the currently executing software, allowing a special function, known as an Interrupt Service Routine (ISR), to execute. After the ISR executes, the system resumes executing the interrupted software. This mechanism is more efficient than polling as it allows the CPU to perform other tasks instead of continuously checking the status of the UART. When data is received or ready to be transmitted, the UART generates an interrupt signal, causing the CPU to execute the ISR to handle this data, thus ensuring real-time responsiveness and efficient CPU usage.

The driver's core responsibility includes managing two distinct queues: one for incoming data (the receive direction) and the other for outgoing data (the transmit direction). In the enhanced version of the driver, these queues and implementation details will be abstracted and concealed from the user tasks, emphasizing that certain implementation details are internal to the driver and need not be exposed externally.

In this part, you will complete and enhance an incomplete interrupt-driven driver for the Universal Asynchronous Receiver/Transmitter (UART) within the Zynq-7000 System on Chip (SoC). The primary goal of this exercise is to design an interrupt-driven architecture for effectively receiving and transmitting bytes via the UART interface. Specifically, this involves receiving bytes from the Software Development Kit (SDK) terminal through the UART and transmitting bytes back to the SDK terminal via the same channel.

Two driver functions, named **myReceiveFunction()** and **mySendFunction()**, are to be developed. Students will base their work on the initial code provided for these functions in the file **uart_driver.c**, which is accessible on the eClass site.

This exercise provides practical experience in driver development and data queue management, crucial skills in computer interfacing and embedded systems programming.

In this part of the laboratory, students are required to focus their work on two files: **lab_2_part_2_main.c** and **uart_driver.c**.

The **uart_driver.c** file contains the driver software. This software is integral to the exercise as it includes the Interrupt Service Routine (ISR), which is fundamental to the operation of the driver in an interrupt-driven environment. Moreover, this file is where the students will define the two user-defined functions: **mySendFunction()** and

myReceiveFunction(). These functions represent the core of the driver software, facilitating improved handling and processing of data through the UART interface.

This structure ensures that students are exposed to both the application layer, via **lab_2_part_2_main.c**, and the driver layer, through **uart_driver.c**, providing a comprehensive understanding of the software stack and the interactions between different software layers within a real-time operating system environment.

Suggestions on how to successfully complete Part 2 of the lab:

1. Complete the code for **handleReceiveEvent()** for the receive direction interrupts in the **uart_driver.c** file.
2. Keep track of the number of transmit interrupts in the **handleSentEvent()** function.
3. Complete the **transmitDataFromQueue()** function code to receive data from the queue and send it through the UART interface.
4. Complete the logic of the **disableTxEmpty()** function to disable the **TEMPTY** interrupts.
5. Complete the user-defined function **myReceiveByte()**.
6. Complete the user-defined function **mySendByte()**.
7. Complete the **vBufferReceiveTask()** task inside **lab_2_part_2_main.c** source file.

Function name	purpose
XUartPs_ReadReg();	Reads a UART register. Returns the values read into an 8-bit variable.
XUartPs_WriteReg();	Writes 8-bit data to a UART register.
XUartPs_GetInterruptMask();	Returns the current interrupt mask.
XUartPs_SetInterruptMask();	Sets an Interrupt Mask. The second argument is the mask that contains interrupts to be enabled or disabled. A value of '1' enables the interrupt and '0' disables the interrupt.
xQueueSendToBackFromISR();	Send a message to the back of a queue from the Interrupt Service Routine (ISR).
xQueueReceiveFromISR();	Receives a message from a queue within the Interrupt Service Routine (ISR).

Table 2: Useful functions for completing Part 2

QUESTIONS

1. What are the implications for data integrity and system stability in the absence of protection for critical sections within the driver functions?
2. Is there a preferred order for processing interrupts? Briefly analyze how the sequence of handling transmit and receive interrupts in the interrupt service routine (ISR) impacts system performance or reliability. Provide a rationale for why one sequence may be more advantageous over the other in the context of efficient system operation.
3. Explain the importance of disabling transmit interrupts when no data remains to be transmitted. What are the potential effects on the system if transmit interrupts were to remain enabled in such circumstances?
4. Justify the number of interrupts that you found after sending three different sized messages to the Zybo Z7 board.
5. Discuss the importance of making queues and other implementation details private and inaccessible to the user in the context of the driver functions. Why is it important to hide implementation details from the users? How does encapsulating these details affect the overall system robustness and reliability?
6. Create a string composed of the names of all group partners, each name separated by a single space. Ensure that names are properly formatted with correct capitalization and no additional spaces before or after each name. For example, if your group consists of 'Alice Smith', 'Bob Jones', and 'Carol Liu', your string should be 'Alice Smith Bob Jones Carol Liu'. Calculate the hash of the string and create a proof of work with a difficulty of 10.
7. Find a Proof of work for the string “ECE315” with a difficulty of 12.
8. Verify that the following string message produce the corresponding hash.

Message	Hash
ECE 315 Lab 2	AD732C99EC35BF11212D1AF4D36E14CF292013CA1EB5EB6539C9A3914C9AF344
City of Champions	DF803B8BA97D9FCC794D8F1D5B4882AB989BF7AA3F540317C1BC37C726A915DA
University of Alberta	AE229AD3284D4503C73DCAAB20E25CEBE449F39E3381B953BF8803C5C00ED818
Hashing is fun!	52C5F6CCE8625209428FF6F1178E6059BD052541799B9BC48DB4837B6F348D41

Table 3: Hash table

Additionally, ensure that your main source code, along with the modified uart_driver.c file, is submitted as part of your laboratory report. This documentation is crucial for a comprehensive evaluation of your work.

ECE 315 Computer Interfacing

LAB 2: The Zynq-7000 UART Interface

Winter 2024

RESOURCES

- FreeRTOS Documentation.
- Zybo Z7 Reference Manual.
- Secure Hash Standard (SHS).

REPORT REQUIREMENTS

Cover Page:

- ➔ Include course and lab section numbers, lab number, and due date.
- ➔ List all student names in your lab team. Please omit ID numbers to respect privacy.

Abstract:

- ➔ 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, expected and actual results and a 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 PDF with 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 300 pixel resolution.
- ➔ Source code in your report should be syntax-highlighted to enhance readability. Syntax highlighting differentiates code elements such as keywords comments 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.

MARKING SCHEME

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