Lab 2: The Zynq-7000 UART Interface

Due date: March 13, 2024

Student Names: Parth Dadhania
Het Bharatkumar Patel

Course Section: B1

Lab Section: H31

Abstract:

In this lab, we embarked on an exploration of UART interfacing within the realm of embedded systems, employing the versatile Zynq-7000 platform. The lab's core objectives revolved around mastering UART communication for efficient user input and system output, leveraging the robust capabilities of FreeRTOS queues for seamless inter-task communication, and delving into the realms of both polling and hardware interrupts for hardware-software interfacing. The exercises were meticulously designed to provide hands-on experience with UART polling, where we developed a text hashing application using the SHA-256 algorithm. Further elevating our learning curve, we ventured into crafting a more advanced UART driver, utilizing interrupts to enhance communication efficiency and system responsiveness. Through these exercises, we not only honed our skills in interfacing and real-time system management but also gained valuable insights into the practical applications of hashing functions, thereby broadening our understanding of embedded system design and operation.

Design Section:

Part 1 - Basic Hashing System Using Polling:

In Part 1, we aimed to develop a system that leveraged the UART interface for text input and output within a FreeRTOS environment, with a focus on applying the SHA-256 hashing algorithm and exploring a foundational proof-of-work mechanism.

Queue Management and UART Communication:

Our design commenced with the establishment of three principal FreeRTOS queues: *xOutputQueue*, *xPoWInQueue*, and *xPoWOutQueue*. We utilized the *xOutputQueue* to manage strings destined for UART output, effectively separating the generation of output data from its transmission. This approach ensured that tasks remained non-blocking and improved system responsiveness. The Proof-of-Work (PoW) queues, *xPoWInQueue* and *xPoWOutQueue*, were pivotal in orchestrating data flow for PoW-related tasks, showcasing our application of queues for task synchronization and data interchange in real-time contexts.

Hashing and String Conversion:

We employed the *sha256String* function for the crucial task of SHA-256 hashing. This function accepted an input string, applied the SHA-256 algorithm, and stored the resultant hash in a byte array. Subsequently, the *hashToString* function converted this array into a hexadecimal string format, facilitating user readability and further processing, such as verification.

Output Handling and User Interaction:

Through the *printString* function, we demonstrated efficient inter-task communication by enqueuing individual characters for UART transmission via the *xOutputQueue*. This method allowed us to delineate data generation from transmission duties across different tasks, aligning with the real-time system design principle of task specialization for enhanced manageability and responsiveness.

We managed user interactions using the *receiveInput* and *printMenu* functions, which oversaw UART data reception and presented operational choices to the user, respectively. These functions highlighted our application of polling to detect incoming UART data and the integration of user inputs into the system's operational flow, enabling dynamic functionality based on user commands.

Proof-of-Work Implementation:

We explored a Proof-of-Work mechanism as an advanced extension to the hashing system. This feature illustrated the practicality of hashing algorithms in computational work verification scenarios, relevant in cryptocurrency and blockchain applications. The implementation entailed generating a hash with a specified number of leading zeros, necessitating adjustments to the input's nonce value until the target hash was achieved.

Task Design and Scheduling:

We crafted the system architecture around multiple FreeRTOS tasks, each dedicated to a distinct aspect of the application's functionality. This modular approach enhanced scalability and allowed for easy feature integration or modification with minimal impact on other components. The FreeRTOS kernel managed task scheduling and prioritization, ensuring optimal resource utilization to meet real-time operational demands.

In designing Part 1, we developed a multifaceted system composed of several interlinked tasks. We initiated with an Input Task, where we continuously polled the UART registers to capture and forward user inputs, effectively translating keystrokes into actionable data within our embedded system. Our Hashing Task took center stage by receiving this data and performing a variety of operations based on user selections, including applying SHA-256 hashing to input text and verifying provided hashes against computed ones, with outcomes relayed to the Output Task for user notification. We also ventured into a Proof of Work Task, tasked with assessing and iterating over nonces to meet a predetermined hash difficulty, a challenge we embraced to underscore the utility of hashes in secure computing. Finally, our Output Task seamlessly managed and formatted data from its counterparts, ensuring clear communication back to the user via UART, culminating in a cohesive system that adeptly handled input processing, secure hashing, and user interaction.

Part 2 - Interrupt Driven Input and Output Using the UART:

In Part 2, we expanded on our UART interfacing knowledge by implementing an interrupt driven UART communication system. Our focus was on enhancing the efficiency and responsiveness of the UART communication by transitioning from polling to interrupt mode for receiving data, while maintaining polling for data transmission.

Interrupt-Driven Reception:

We integrated an interrupt service routine (ISR) to handle UART data reception. This approach allowed us to respond immediately to incoming data without continuously polling the UART status registers. Upon receiving data, the ISR was triggered, and the received bytes were queued into *xRxQueue* for further processing. This method significantly improved the system's ability to handle incoming data promptly and efficiently.

Queue Management:

The use of FreeRTOS queues, *xTxQueue* for transmission and *xRxQueue* for reception, played a pivotal role in our design. These queues facilitated a smooth and controlled exchange of data between different parts of the system. For instance, when the ISR received data, it placed the bytes in *xRxQueue*, from where they were processed by the main application tasks. Similarly, data to be transmitted was enqueued into *xTxQueue*, from which it was sent out when the UART was ready.

myReceiveFunction():

The *myReceiveFunction*() plays a crucial role in the interrupt-driven UART communication system by handling the reception of data from the UART interface. Functionally, this method checks for incoming data in the receive queue (*xRxQueue*) and retrieves it for further processing by the application. The implementation leverages the FreeRTOS queue management system, where the interrupt service routine (ISR), upon receiving data, places it into *xRxQueue*. *myReceiveFunction*() then polls this queue to see if any data is waiting. If data is present, it is dequeued and returned to the caller, allowing for the processing or manipulation of this data as per the application's requirements. This design encapsulates the data reception process within a function, abstracting the complexity of ISR and queue management from the main application logic, thereby adhering to the design principles of modularity and encapsulation.

mySendFunction():

The *mySendFunction*(), on the other hand, complements the data reception mechanism by facilitating the transmission of data over the UART interface. In line with the lab's objectives to enhance UART communication efficiency, this function dynamically assesses the UART's readiness to send data and ensures that data transmission does not block or impede the execution of other critical tasks. The function first checks if the transmit queue (*xTxQueue*) has space available; if so, it enqueues the data to be transmitted. It then triggers the UART to send this data by enabling the relevant interrupt, which is handled by the ISR to transmit the data byte-by-byte from *xTxQueue*. This approach of using a transmit queue and interrupt-driven transmission ensures that data can be queued for transmission at any time and will be sent out as the UART becomes available, without necessitating constant polling by the application. This design not only improves system responsiveness but also aligns with real-time operating system principles by efficiently managing task priorities and execution.

Capitalization Logic:

A key feature of our design was the ability to alter the capitalization of received text before transmitting it back. In the receive task, we implemented logic to check each character's case and convert lowercase letters to uppercase and vice versa. This feature was not only a demonstration of string manipulation but also served as a practical exercise in real-time data processing within an embedded system.

Special Command Sequences:

We introduced special command sequences, such as *"\r#\r"* and *"\r%\r"*, to interact with the system in a more sophisticated manner. For example, the sequence *"\r#\r"* triggered the system to display interrupt statistics, providing insight into system performance. The sequence *"\r%\r"* reset the interrupt and byte counters, aiding in system diagnostics, and debugging.

Display and Button Integration:

To enhance user interaction, we incorporated a Seven Segment Display (SSD) and button inputs. The SSD was used to display various system statistics, such as interrupt counts, and the number of bytes processed. The buttons allowed users to select the specific statistics to display on the SSD, adding an interactive element to the system.

In developing Part 2, we fully leveraged the capabilities of interrupts within real-time systems for our UART communication, significantly enhancing system responsiveness by implementing an Interrupt Service Routine (ISR) that efficiently managed data reception and transmission, thereby negating the need for continuous polling. This strategic approach allowed the CPU to focus on additional tasks, optimizing overall system performance. We meticulously designed the driver to manage incoming and outgoing data through two distinct queues, encapsulating the complexities of interrupt handling and queue management within the driver, thus abstracting these details from the user-level tasks. Our primary focus was on refining an interrupt-driven UART driver to adeptly handle byte transfers via the UART interface, centering our efforts on two essential functions, *"myReceiveFunction()"* and *"mySendFunction()"*, which were developed from a foundational codebase. This task extended beyond mere driver development; it encompassed a deep exploration of data queue management, a critical aspect of computer interfacing and embedded system programming.

Testing Suite:

Part 1:

Test Case	Description	Expected Result	Actual Result	Rationale
1	Hashing a String: ECE 315 Lab 2	SHA-256 value of String "ECE 315 Lab 2"	SHA-256 value of String "ECE 315 Lab 2"	Ensure correct mapping of String to it's SHA-256 hash value
OUTPUT: SHA256 Hash of ECE 315 Lab 2 is: **ad732c99ec35bf11212d1af4d36e14cf292013ca1eb5eb6539c9a3914c9af344**				
2	Hashing a String: City of Champions	SHA-256 value of String "City of Champions"	SHA-256 value of String "City of Champions"	Ensure correct mapping of String to it's SHA-256 hash value
OUTPUT: SHA256 Hash of City of Champions is: **df803b8ba97d9fcc794d8f1d5b4882ab989bf7aa3f540317c1bc37c726a915da**				
3	Hashing a String: University of Alberta	SHA-256 value of String "University of Alberta"	SHA-256 value of String "University of Alberta"	Ensure correct mapping of String to it's SHA-256 hash value
OUTPUT: SHA256 Hash of University of Alberta is: **ae229ad3284d4503c73dcaab20e25cebe449f39e3381b953bf3803c5c00ed818**				
4	Hashing a String: Hashing is fun!	SHA-256 value of String "Hashing is fun!"	SHA-256 value of String "Hashing is fun!"	Ensure correct mapping of String to it's SHA-256 hash value
OUTPUT: SHA256 Hash of Hashing is fun! is: **52c5f6cce8625209428ff6f1178e6059bd052541799b9bc48db4837b6f348d41**				
5	Verifying hash of a given string: ECE 315 Lab 2	Verification Successful	Verification Successful	Ensure successful verification if pre-computed hash matches with the hash of a given string
INPUT: Precomputed hash of ECE 315 Lab 2: **ad732c99ec35bf11212d1af4d36e14cf292013ca1eb5eb6539c9a3914c9af344**				
6	Verifying hash of a given string: ECE 315 Lab 2	Verification Failed	Verification Failed	Ensure verification fails if pre-computed hash doesn't match with the hash of a given string

INPUT: Precomputed hash of ECE 315 Lab 2: **df803b8ba97d9fcc794d8f1d5b4882ab989bf7aa3f540317c1bc37c726a915da**				
7	Creating a Proof of Work of a given string for a difficulty level	Proof of Work of a given string for a difficulty level	Proof of Work of a given string for a difficulty level	Ensure correct generation of Proof of Work of a given string for a difficulty level

Part 2:

Test Case	Description	Expected Result	Actual Result	Rationale
1	Sending lowercase characters as user input: “abcde”	“ABCDE” as the output	“ABCDE” as the output	Ensures the change in capitalization of letters
OUTPUT on SSD: Byte count (BTN0) = 5, Rx interrupts (BTN1) = 12, Tx interrupts (BTN2) = 5				
2	Sending “/r#/r” as user input	Printing Byte count, Rx interrupts and Tx interrupts	Printing Byte count, Rx interrupts and Tx interrupts	Ensure that the counts are correct
OUTPUT in terminal: Byte count = 9, Rx interrupts = 20, Tx interrupts = 10				
3	Sending uppercase characters as user input: “QWERT”	“qwert” as the output	“qwert” as the output	Ensures the change in capitalization of letters
OUTPUT on SSD: Byte count (BTN0) = 10, Rx interrupts (BTN1) = 24, Tx interrupts (BTN2)=10				
4	Sending “/r#/r” as user input	Printing Byte count, Rx interrupts and Tx interrupts	Printing Byte count, Rx interrupts and Tx interrupts	Ensure that the counts are correct
OUTPUT in terminal: Byte count = 19, Rx interrupts = 40, Tx interrupts = 20				
5	Pressing BTN3 on the Keypad	Reset interrupts and byte count to 0	Reset interrupts and byte count to 0	Ensures interrupts and byte count are set to 0
OUTPUT on SSD: Byte count (BTN0) = 0, Rx interrupts (BTN1) = 0, Tx interrupts (BTN2) = 0				
6	Sending “/r%/r” as user input	Reset interrupts and byte count to 0	Reset interrupts and byte count to 0	Ensures interrupts and byte count are set to 0
Byte Count, and interrupt counters set to zero				

Discussion:

Answers to the questions from Lab Manual:

1. In the absence of protection for critical sections within the driver functions in *uart_driver_student.c*, particularly when interacting with shared resources like UART queues, several significant implications for data integrity and system stability arise. Firstly, without adequate protection, there's a heightened risk of data corruption. This could happen when multiple tasks or interrupts simultaneously attempt to access or modify the queues, leading to incomplete or overwritten data entries. Such corruption can cause erroneous data transmission or reception, directly impacting the reliability of UART communication.
   Secondly, system stability can be compromised due to race conditions, where the execution sequence of tasks leads to unpredictable states. For instance, if one task is in the middle of enqueueing or dequeuing data while another task or ISR pre-empts and tries to perform a similar operation, it could lead to deadlocks or crashes if the queue state becomes inconsistent or if buffer overflows/underflows occur. Given the real-time nature of embedded systems used in the lab, such instability could lead to system failures or erratic behavior, severely impacting the application's performance and reliability. Implementing mechanisms like FreeRTOS's *taskENTER_CRITICAL()* and *taskEXIT_CRITICAL()*, or using queue API functions designed for ISR contexts, are crucial in safeguarding against these issues, ensuring that data handling within the driver remains robust and the system operates reliably.

2. In the context of UART communication for Lab 2 Part 2, handled by *uart_driver_student.c*, the sequence of processing interrupts—specifically, whether to prioritize transmit or receive interrupts in the Interrupt Service Routine (ISR)—can significantly affect system performance and reliability. Prioritizing receive interrupts is often advantageous, especially in systems where the integrity and timely handling of incoming data are crucial. This approach ensures that new data is quickly read from the UART, reducing the risk of data loss due to buffer overflows, and maintaining high data integrity. It's particularly beneficial in applications requiring prompt responses to incoming information, as it ensures that incoming commands or data are processed immediately, keeping the system responsive and reliable.
   Conversely, there might be scenarios where prioritizing transmit interrupts could be more beneficial, especially in control systems or applications where sending timely responses is critical. However, this can lead to the neglect of incoming data, potentially causing buffer overflows or unprocessed inputs. Generally, the decision on interrupt processing order should be tailored to the application's specific needs, with a bias towards receive interrupts in many cases to safeguard against data loss and ensure

system stability. The choice should consider factors like the expected data flow, the importance of timely outbound vs. inbound communication, and the consequences of delayed data processing.

3. Disabling transmit interrupts when there are no data left to be transmitted is crucial for maintaining system efficiency and preventing unnecessary processor interruptions. In the *uart_driver_student.c* file, the function *disableTxEmpty()* is designed for this purpose. When transmit interrupts remain enabled despite the absence of data, the UART hardware continuously signals the processor for attention, leading to ISR invocations without any productive outcome. This not only wastes CPU cycles that could be allocated to other critical tasks but also increases the risk of the system becoming overwhelmed with spurious interrupts, thereby degrading overall performance.
   If transmit interrupts were allowed to remain enabled without pending data, it could lead to a situation where the system's responsiveness to other, more critical interrupts might be delayed or compromised. This constant interruption without cause disrupts the normal execution flow, potentially affecting time-sensitive operations within the system. Moreover, in a scenario where interrupt priority is not optimally managed, these unnecessary interrupts could pre-empt more crucial service routines, leading to latency issues and, in extreme cases, system instability or failure due to missed deadlines in real-time applications. Therefore, judiciously managing the enabling and disabling of transmit interrupts is essential for ensuring efficient and reliable system operation.

4. After sending three different sized messages to the Zybo Z7 board, the number of interrupts observed corresponds to the behavior and design of the UART communication and the interrupt-driven system implemented in the *uart_driver_student.c* file. The UART interface generates interrupts for various events, such as data reception, data transmission readiness, and buffer states (empty or full). The number of interrupts is influenced by the size of the messages due to the way data is processed and moved between the UART hardware and the software queues.
   For smaller messages, the number of interrupts might be relatively low, primarily because the entire message can be quickly received by the UART, causing a series of receive interrupts, and then promptly processed by the ISR. As the message size increases, the number of interrupts increases proportionally due to the need to handle each byte (or chunk of bytes, depending on the hardware FIFO settings) of data as it is received or transmitted. Larger messages require more data transactions, each potentially triggering an interrupt when the UART is ready to receive more data or when the transmit buffer becomes empty.

5. Making queues and other implementation details private and inaccessible to users within the context of driver functions, as seen in the *uart_driver_student.c* and *lab_2_2.c* files, is crucial for maintaining the abstraction layer between the hardware interface and the application logic. This encapsulation ensures that users interact with the UART functionality through a well-defined API, without needing to understand the underlying complexities of interrupt handling, queue management, or data synchronization. By hiding these details, the driver maintains control over the data flow and interrupt management, reducing the likelihood of user-induced errors, such as improper access to shared resources, which could lead to data corruption or system instability.
   Encapsulation of these details enhances system robustness and reliability by providing a controlled environment where data integrity and access synchronization are managed internally within the driver. This approach minimizes the risk of concurrent access issues and ensures that the system's critical sections are protected, especially in a multitasking environment where tasks may pre-empt each other. Keeping implementation details abstracted away from the user also facilitates easier updates or modifications to the driver, as changes can be made internally without affecting the application code that relies on the driver's API.

6. String composed of the names of all group partners: "**Het Bharatkumar Patel Parth Dadhania**".
   Hash of the above given string:
   bd8d13bdc7e56e70e2066c4abf2b0b7955fe6f32127682318e633ec6be281d59

Conclusion:

The objectives set forth for Lab 2 were successfully met, culminating in a comprehensive understanding and practical application of UART interfacing within embedded systems using the Zynq-7000 platform. The exercises facilitated a deep dive into both polling and interrupt-driven communication, enabling us to experience firsthand the nuances and efficiencies of each method. The successful implementation of a text hashing application and the development of an advanced UART driver underscored the lab's success in achieving its educational goals. Notably, the lab underscored the pivotal role of FreeRTOS queues in managing inter-task communications, a critical component in the architecture of real-time systems. Throughout the lab, we encountered no significant issues that hindered our progress, thanks to the well-structured exercises and the robustness of the Zynq-7000 platform. This lab experience not only reinforced our theoretical knowledge but also provided a solid foundation for future explorations in the field of computer interfacing and embedded systems.