**PROJECT – 2**

**Character Histogram**

**Principles of Embedded Software**

**By**

**Steve Antony and Swarupa**

**PROJECT REPORT**

PART - 2

**Q1) Is your implementation thread safe? Why or why not?**

Our Implementation is not thread safety because our code doesn’t contain only atomic operations.

There are many parts of the program which are not atomic eg: count++ .

For the implementation to be thread safe, all operations must be atomic or need to have locks or need to have variables as volatile.

**Q2) What potential issues exist if the same buffer is used by both interrupt and non-interrupt code? How can these issues be addressed?**

According to our code, when the same buffer is used simultaneously for both interrupt and non-interrupt code, say we get a data in UART0\_D, an interrupt will be raised, and the interrupt part of code will be executed and the data from UART0\_D will be fetched. The non-interrupt code will not be invoked as the interrupt will be given preference and as the UART0\_D data is fetched, flag RDRF will never be found as set during polling.

This issue can be addressed by disabling the polling method if the data is fetched through interrupt. So, the polling condition should also check a flag which we can set it in ISR. If that flag is set, it need not execute the non-interrupt code.

**Q3) How could you test these issues?**

These issues can be tested by enabling the interrupt modes of UART as well as having polling method

PART - 4

**Q1) For each implementation, what is the CPU doing when there are no characters waiting to be echoed? What is the behavior of the GPIO toggle in the non-blocking implementation?**

In the blocking mode, when it is not waiting to echo any character, it will be polling the flags TDRE and RDRF.

In the non-blocking mode, when there are no interrupts and while the control is in the while loop, the led toggles. The LED state is maintained when it exits from the while loop and servers the interrupt. If an interrupt occurs when the LED is ON, it will be ON until the interrupt completes. If an interrupt occurs when the LED is off, it is will off until the interrupt completes

**Q2) For each implementation trace the sequence of events that occur by listing, in order, the functions called from the point that a character sent to the FRDM board has been received until the point where the echoed character has been sent.**

Non- Blocking:

1. LED toggles
2. RX Interrupt received
3. State of the system saved
4. Goes to the RX ISR
5. Interrupt disabled
6. Checks whether the RDRF is set or TDRE set
7. Receive the character and store in the circular buffer
8. Pop individual character and update the frequency of the character in the array
9. Interrupt enabled
10. TX interrupt enabled when buffer is not empty
11. TX interrupt occured
12. State of the system saved
13. Goes to the TX ISR
14. Interrupt disabled
15. Check whether interrupt occurs due to TDRE or RDRF
16. Print the character on console
17. Interrupt enabled
18. Goes back to main thread

Blocking:

1. Polling for the RDRF flag and TDRE flag
2. When a character is typed on serial port, RDRF set
3. Receive data from UART0\_D to the circular buffer through polling RDRF
4. Polling for the TDRE flag for it to be empty and when the buffer is not empty
5. When TDRE is set, it calls the function to print on console through polling TDRE

**Q3) Comment on the interface presented to the main() application code for blocking vs. non-blocking variation. Which variation is easier to code to?**

Polling method is easier to implement the application as the implementation is quite easy when compared interrupt implementation. But many cycles of CPU time are wasted in polling

Part - 5

**Q1) What is the CPU doing after the last character has been received and while the report is being printed?**

After the last character is received, CPU returns to the main loop execution from the ISR. According to our code, report will be generated by Systick handler. So while the report is printed the main execution will be halted and serve the ISR of Systick.

**Q2) Baud rate aside, what limits the rate at which the application can process incoming characters? What happens when characters come in more quickly than they can be processed?**

Other than the baud rate, the size of the circular buffer also limits the rate at which the characters can be processed. However, in our case we are not storing the characters instead popping them out instantly and updating their frequency for the application concurrently. Thus, avoiding the situation where the buffer overflows.

**Q3) How does the size of the circular buffer affect report output behavior (especially during an onslaught)? What is an appropriate buffer size to use for this application? Why?**

We do not use circular buffer for output reporting. If a circular buffer is used, CB overflows and hence it must be resized as and when it overflows. Appropriate size would be a huge value around 500 so that frequent resizing can be avoided.