PROJECT REPORT ON

CIRCULAR BUFFER, UART AND INTERRUPT IMPLEMENTATION

By

SIDDHANT JAJOO
SATYA MEHTA

UNDER THE GUIDANCE OF:

PROFESSOR KEVIN GROSS,
UNIVERSITY OF COLORADO BOULDER

DATED:11/26/2018.

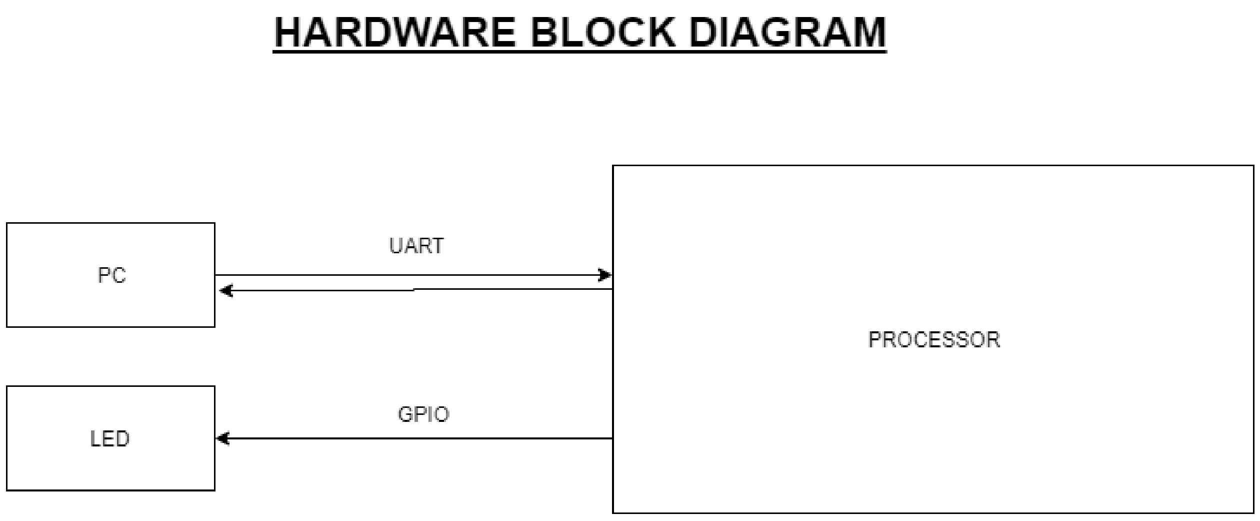
TABLE OF CONTENTS

DOCUMENTATION

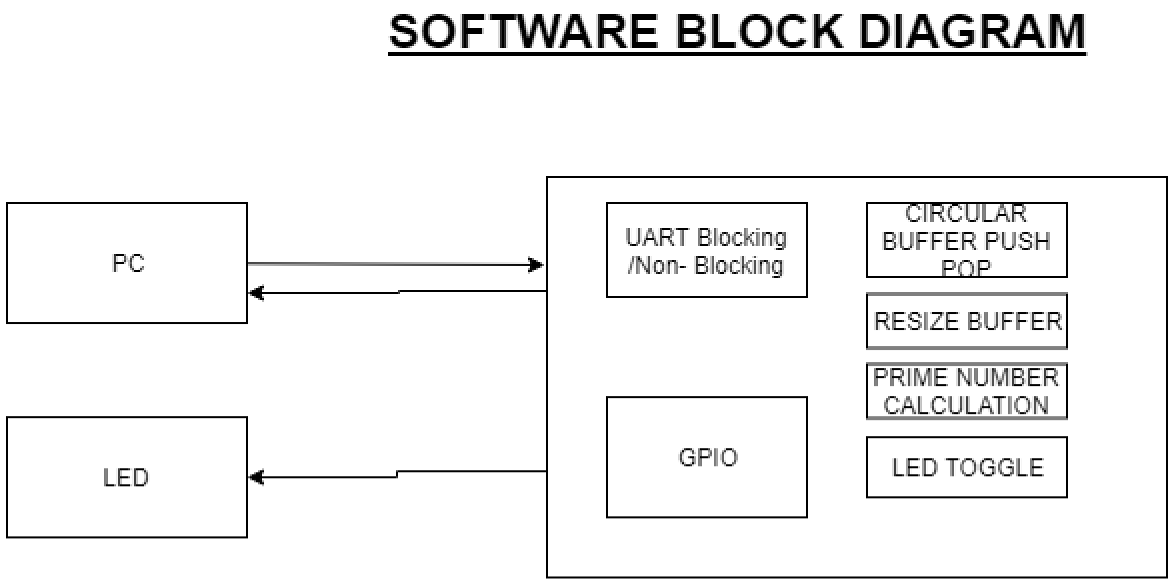
The following are the different modules used in the implementation of the Project

1. Circular Buffer
   a. Circular Buffer Push Pop Implementation. - Inserts and removes data from the buffer
   b. Circular Buffer Resize Implementation. - Changes the buffer size
   c. Circular Buffer Get Size Implementation - Gets the number of elements stored in buffer
2. UART
   a. UART Blocking Mode Implementation
   b. UART Non - Blocking Mode Implementation
   c. Printf function using Variadic Function.
3. Calculate Prime Numbers

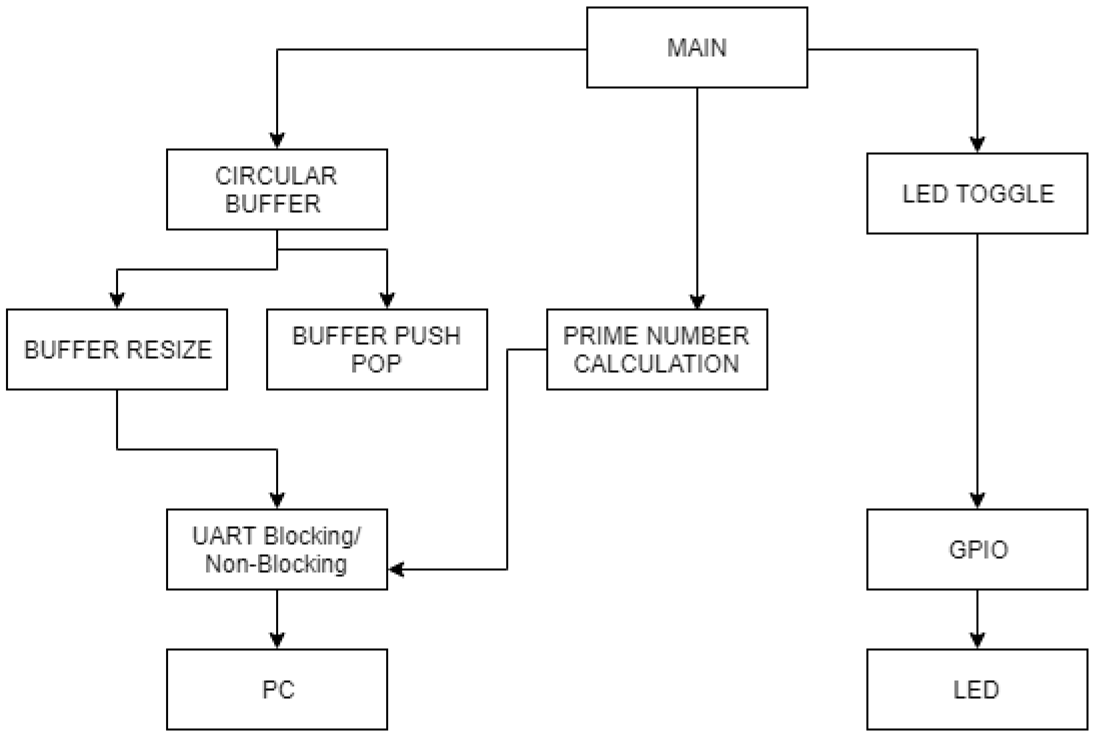
HARDWARE BLOCK DIAGRAM



SOFTWARE BLOCK DIAGRAM



ORGANIZATIONAL DIAGRAM



REPORT QUESTIONS

PART 1:

1) Is your implementation thread safe? Why or why not?
ANS: The Circular Buffer initialization uses malloc and returns the starting Memory address of the allocated Memory. Now if we use two threads we ought to create two object thus creating two buffers to work on. If we try running on the same buffer there is no provision to do so. There would be data races.

2) What potential issues exist if the same buffer is used by both interrupt and non-interrupt code? How can these issues be addressed?
ANS: In case of non-interrupt code, we use the blocking method so virtually we won't require any buffer or simply a buffer of length 1 because we poll for the receive and transmit flags and service only one data at a time. On the other hand , in case of interrupt implementation , an interrupt is generated depending on the interrupts flag enabled during initialization. In such case the data is not lost if the receiving rate is more than the transmitting rate (i.e push data rate more than pop data rate) as the elements are serviced afterwards. The data is lost in this scenario which is in case of blocking Implementation. An important point here is that the data is lost in non blocking implementation if the data exceeds the buffer size.

3) How could you test these issues?
ANS:This issue can be tested by sending a large block of data together. The basic idea is to have a push data rate to be more than the pop data rate so that the elements are stored in the buffer. This can be done by inputting a text file.
To check if the data looses or not, just input a text file larger than the buffer size.

PART2:

1) 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?
ANS: Blocking Mode:
The CPU waits for an incoming Character in a while loop.

Non-Blocking Mode:
The CPU calculates all the prime numbers till the total number of characters that have been input. The GPIO toggle can be implemented using a timer or a for loop to create a delay.

2) 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.

ANS: BLOCKING MODE:

1) Start
2) Initialize hardware clock , uart and circular buffer.
3) Wait for a character to be inputted . Function used is uart_rx()
4) Echo the received character using uart_tx(data)
5) Increase count of received character using lookup table
6) Print Report
7) Goto step 3
8) Stop

NON-BLOCKING MODE:

1) Start
2) Initialize hardware clocks, uart and circular buffer
3) Continuously Monitor for UART interrupts , if receive interrupt set goto step 5
4) Calculate Prime Numbers till total number of Characters, goto step 3
5) If pressed ESC , resize buffer
6) Else while (buffer_size != 0), print report and count total characters , goto step 3
7) Stop

3) Comment on the interface presented to the main() application code for blocking vs. non-blocking variation. Which variation is easier to code to?
ANS: The Blocking Mode is easier to code because it does not require any buffer.

<u>PART3</u>:

1) What is the CPU doing after the last character has been received and while the report is being printed?
ANS: If a text file has been inserted and the last character has been received , the CPU will pop all the elements of the buffer one by one and print it in the report using the interrupt routine.
The CPU will calculate the prime numbers in between interrupts.
If a text file is not inserted , after receiving the character the CPU will print the character with its count and then resume calculating the number of prime numbers.

2) 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?
ANS: Baud rate aside , the processor frequency limits the rate at which it can process the incoming characters. If the processor frequency is much more than the baud rate , then the buffer may not get full and data losses may not occur. Also, it depends on your instructions in your code. Lesser the instructions, lesser clock cycles will be utilized for processing instructions. If the characters come in more quickly than it can process the incoming characters may get lost.

3) 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?
ANS:To avoid any data losses , it is favourable to have a size which can accomodate a large number of characters. Having a small buffer size will lead to data losses if the buffer overflows.
The buffer size really depends on the application: if large text files are to be serviced then a large buffer size is needed.

APPENDIX

This report has comprehensively covered all the modules , functions , their description, the different modes implemented in UART and answers to relevant questions.

The project covers circular buffer implementation with error handling stating if it is full or empty ,the ability to resize the buffer at any point of time, ability to return the number of elements in the buffer and also the ability to reset the buffer . The circular Buffer has been implemented in such a way that it does not reserve any unused byte of data when the buffer is full. Any number of buffers can be implemented by just using the circbuff_init function.

A C-Unit Test framework has also been implemented. Unit testing is performed on the Linux platform. The test cases cover a variety of normal operational conditions as well as all reportable error conditions. An automated long-running randomized test case has also been implemented.

The Project also successfully implements UART driver with a blocking and non-blocking Mode having two separate working code files respectively. The interrupt based UART calculates all the prime numbers upto the total number of characters when the ISR is not being serviced and displays it along with the report. In addition to this, a **myprintf** function has also been implemented using variadic function. The LED toggles in case of blocking Mode implementation in the main loop. There is no report output corruption (due to circular buffer overflow) regardless of input character rate. This has been tested by sending a large text file. The code for finding all the prime numbers was referenced from:
https://www.tutorialgateway.org/c-program-to-print-prime-numbers-from-1-to-100/