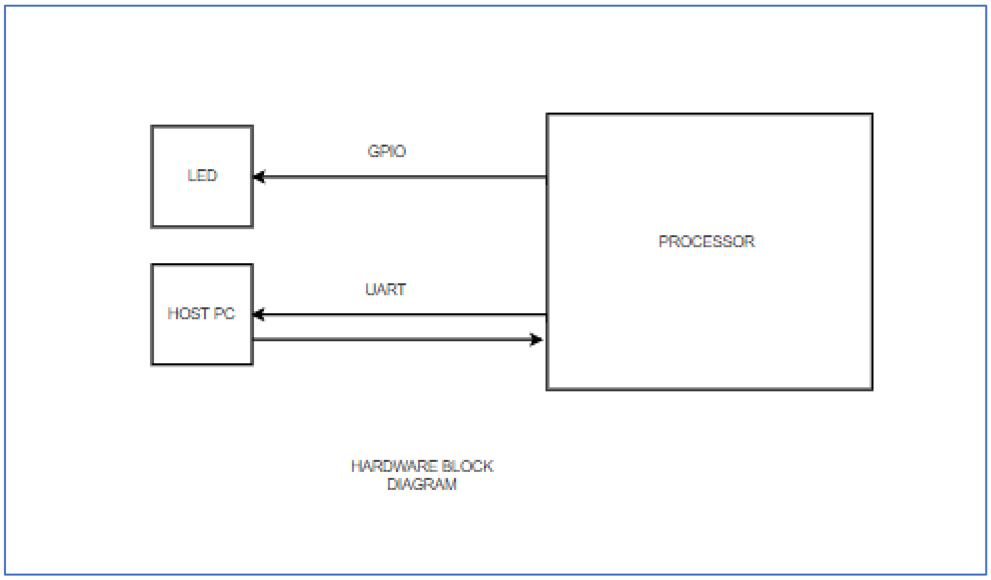
# Project 2

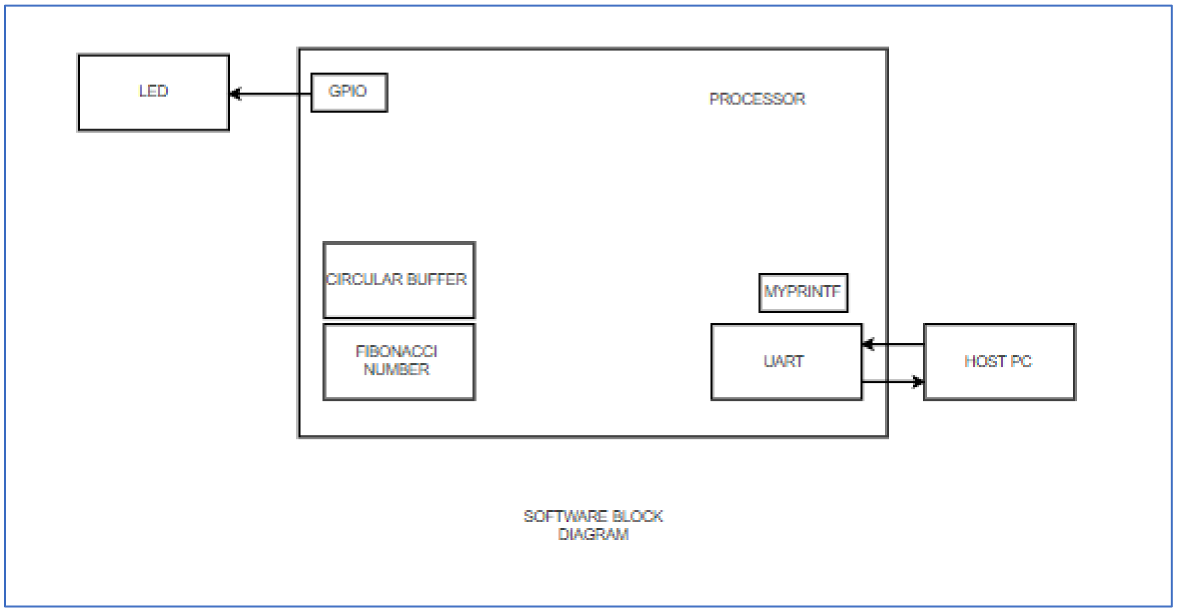
Circular buffer, UART and interrupts

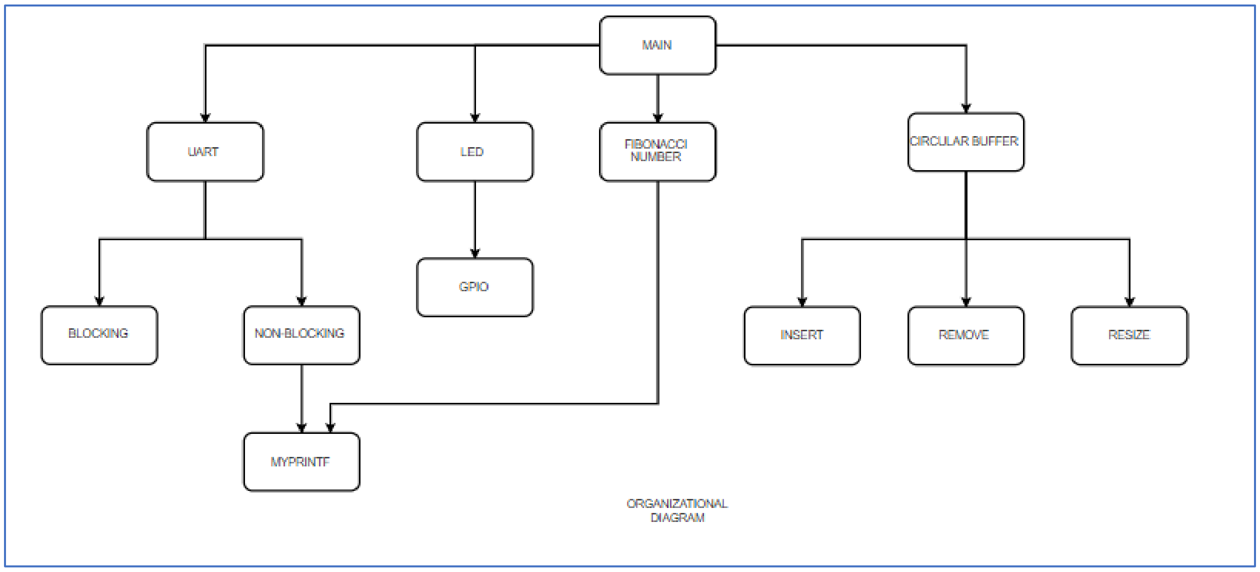
ECEN 5813
Principles of Embedded Software
Spring 2019
Prof. Kevin Gross

**Ayush Dhoot**
**Shubham Jaiswal**

## Part 1: Block diagram and architecture







## Part 2: Circular buffers

### Questions

1. Is your implementation thread safe? Why or why not?

Ans.

No, this implementation is not thread safe. As, if multiple threads are created, there is no way to give access of a particular data only to a single thread at any particular time. So, it may happen that these threads make changes to the same data simultaneously and thereby can hinder the working of the application.

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.

If same buffer is used by both interrupt and non-interrupt code, then issues such as loss of data, corruption of data and incorrect value of character count can arise. As the same buffer is used, it may happen that while non-interrupt code is inserting some character in buffer, at the same time an interrupt occurs and because before input-pointer of buffer could be incremented, the execution was transferred to interrupt handler and there again some character will be inserted into buffer into the same location where last character was entered, thereby over writing and corrupting the data of circular buffer.

3. How could you test these issues?

Ans.

These issues can be tested by enabling and using both the interrupt and non-interrupt codes in the implementation and passing a large file to the serial terminal.

## Part 3: Unit testing

We have implemented a unit testing using CUnit framework and used an automated long running randomized test cases for the application.

## Part 4: UART device driver

### Questions

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:

In case of the blocking mode implementation, the CPU is in wait state. The CPU polls continuously for a new transmission or reception of a new character.

While for the non-blocking mode, the CPU calculates the fibonacci number for the total number of characters received until we get a transmission or receive interrupt.

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. Wait for the character to be received using 'uart_read()'
2. Increase the character count for that particular character using a variable 'char_count'
3. Insert the received data in circular buffer.
4. Remove data from circular buffer. Pass it as argument to uart_write().
5. Now transmit the character back using 'uart_write()'

Non-Blocking mode:

1. Receive a character
2. Go to interrupt handler
3. Insert the character to the buffer
4. Increase count for that character in the array
5. Back in main, Remove character from buffer.
6. If Buffer is empty, then using my_print_irq(), print the report for the character and the fibonacci number for the total characters received.

3. Comment on the interface for sending and receiving characters presented to the main() application code for blocking vs. non-blocking variation. Which variation is easier to code to?

Ans:

In blocking mode, the sending and receiving of characters is more simpler as it can be used without a buffer or using a single element buffer. Meanwhile, for non-blocking mode, the sending and receiving of characters are more complex as we need to have an interrupt handler which will add or remove the characters using the additional implementation of a circular buffer.

The blocking mode variation is much easier to code as there are no interrupts and interrupt handling required. It can be done using just by polling till we complete either the transmission or receive of the character. Also, we do not require to store the characters in a buffer which waits for an interrupt to print the report.

## Part 5: Application

### Questions

1. What is the CPU doing after the last character has been received and while the report is being printed?

Ans:

The CPU is calculating the Fibonacci number for the total number of the characters received in the time when the last character has been received and while the report is being printed.

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:

The clock speed(operating frequency) of the processor being used limits the rate at which the application can process incoming characters. Also, the type of instructions used and number of clock cycles it takes to complete an instruction also limits the processing rate of incoming characters. If the size of circular buffer is not large enough and if the characters keep coming at a rate higher than the rate at which they can be processed, then data losses and corruption and occur.

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:

The size of the circular buffer should be appropriate for the type of application it is used for. If we are to input a single character or a small string, we might not need a large buffer but if we are to place a large text file, we might need a larger buffer. If we have a small buffer of size 1 which will work if we have a single character but if there is a string of characters, the output of the report would not be accurate as it will not be able to fit in all characters giving us a data loss.

Also, we can have a functionality to resize the buffer every time the buffer is full. This will be a good approach which will act for each application and result in no data loss.

## 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. 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 ring_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 the Fibonacci number for the total number of characters when the ISR is not being serviced and displays it along with the report. In addition to this, a my_printf function has also been implemented using variadic function. The code for writing the printf function was referenced from http://www.firmcodes.com/write-printf-function-c/. 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 calculating Fibonacci number was referenced from: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/