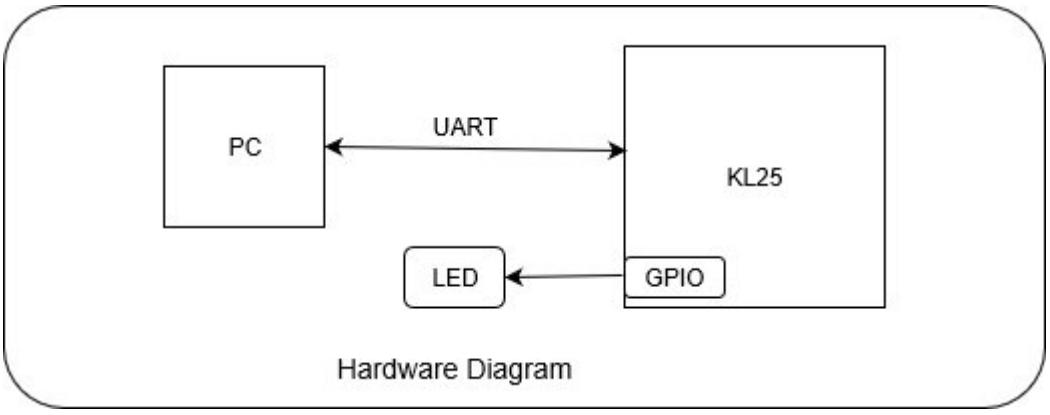
Project 2
Circular Buffers, UART and Interrupts

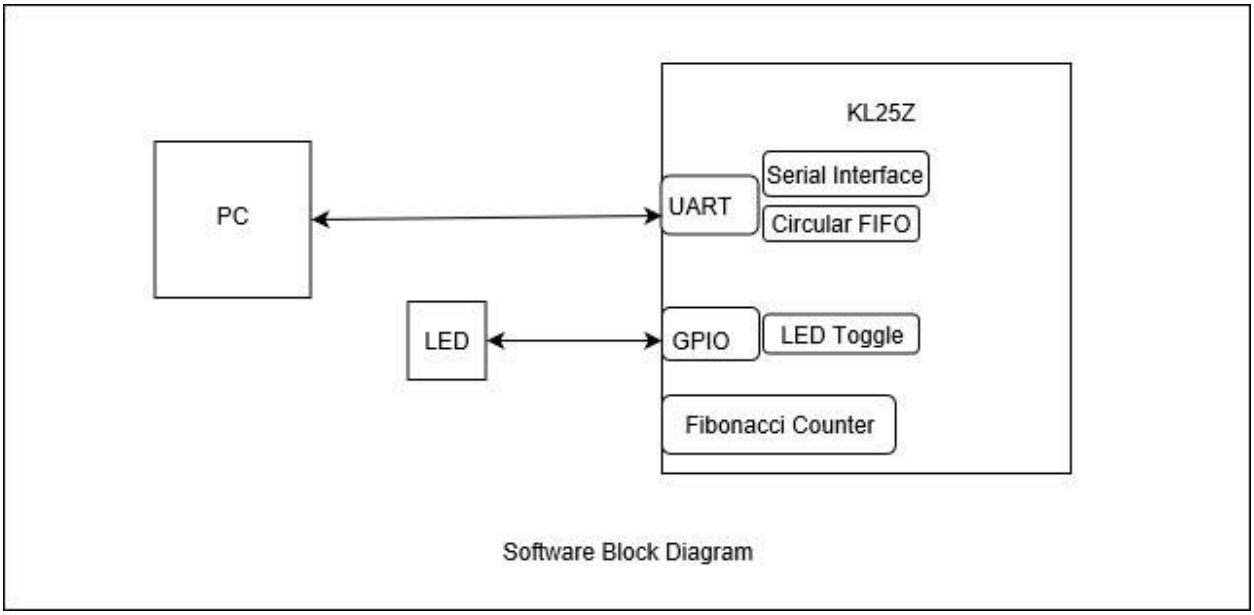
Vatsal Sheth & Sarthak Jain

Part1: Block Diagram and Architecture

Hardware component includes serial interface with computer and a LED. USB for serial interface and LED both are on board development board based on FRDM KL25Z.

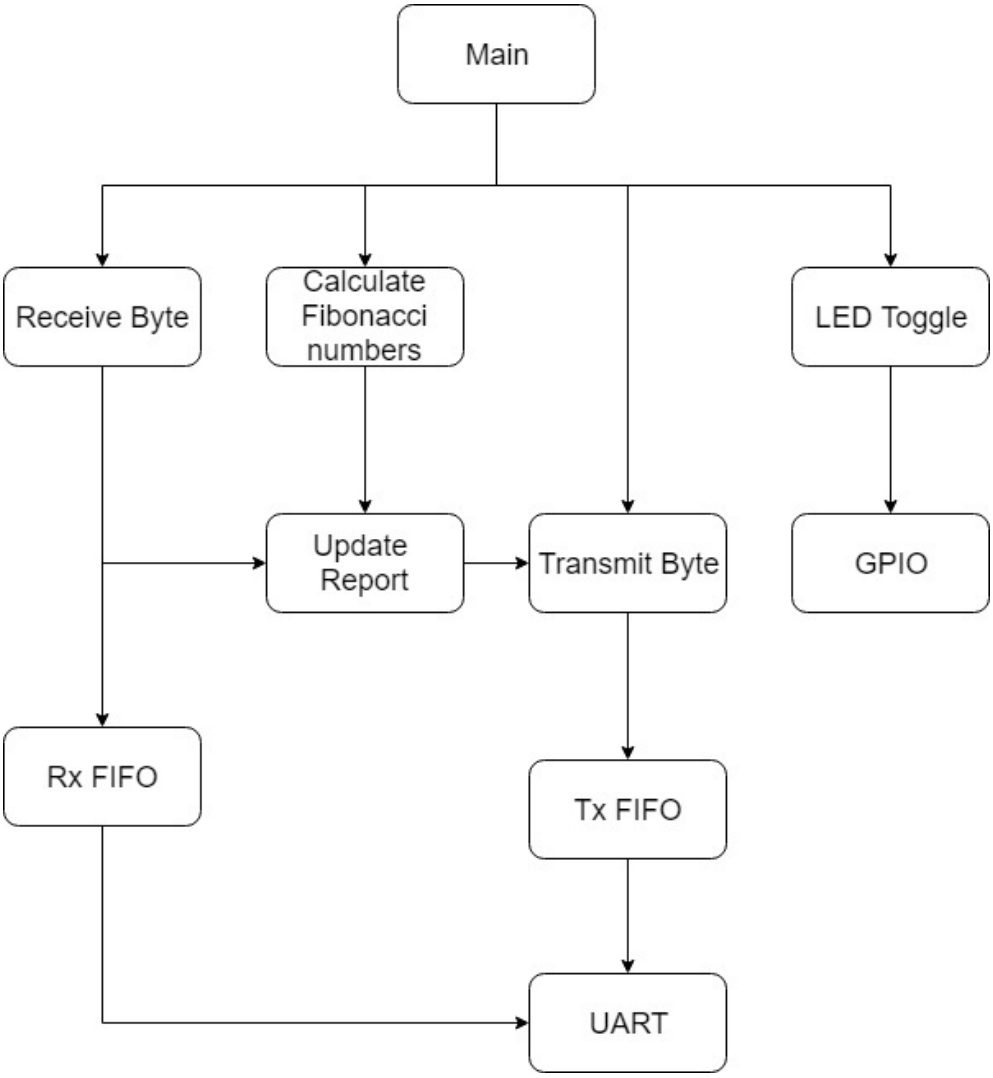


Software component of this project includes UART drivers to communicate serially with computer. Transmit and receive of data is interfaced through Tx and Rx Circular FIFO, which acts as single point of entry and exit of data. Above that serial interface manages the user interface. Next component is LED

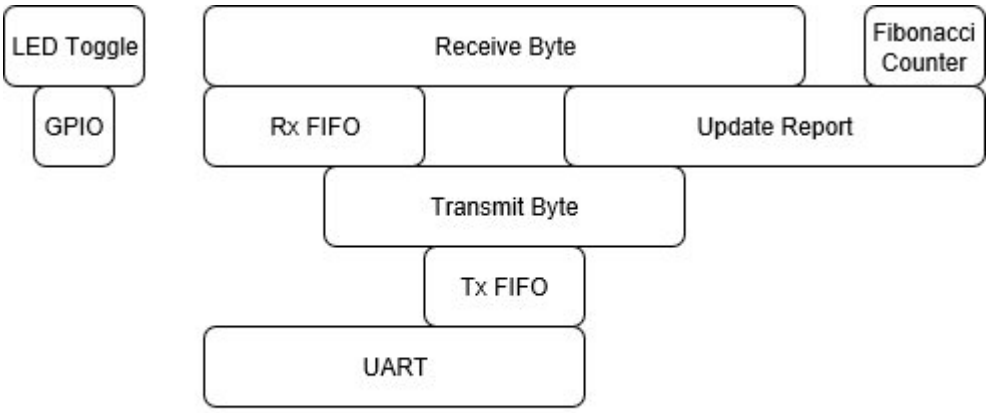


toggle which is implemented through GPIO drivers and Low power timer ISR. Also Fibonacci counter is implemented which runs independently and in non-blocking way with other functions.

Organizational Diagram tells the flow of code. LED toggle is controlled through timer ISR. Main code takes care of processing content of Rx FIFO, Transmitting from Tx FIFO and calculating Fibonacci series.



Layer Diagram is important as objects size shows its complexity and it's dependencies. Firstly, LED Toggle is an independent block. Fibonacci counter is also independent but its current value is displayed along with report so it has a dependency with Update Report block. Received byte is pushed to Rx FIFO which operates through UART. Each received byte is echoed which uses Tx FIFO which in turns uses UART. Also each received byte is processed and reports are updated and updated report is sent to be transmitted.



Layer Diagram

Part 2: Circular Buffers

1) Buffer implementation is not thread safe. Firstly, KL25Z has Cortex M0 core which is single core, that means threads will be managed by OS in time slices and basically it will be sequential. So resource coherency will be an issue. If my serial interface uses threading using my circular buffers, then in that case suppose one thread is made to echo the character and other thread is used to update report then it is possible that while popping a character from Rx buffer one of the thread may loose a character depending on which executes pop instruction first. So it may lead to a scenario where either the character is echoed on terminal but its count is not updated in report or vice versa. To resolve this resource synchronization need to be implemented in buffer code.
2) Same circular FIFO can be used for both interrupt and non-interrupt code. But using FIFO for non-interrupt code doesn't makes sense, as in blocking version even if we add a character to Rx FIFO on receiving we will have to pop it immediately to process it. Whereas in in interrupt version it is required because on Rx interrupt we will push character in FIFO and use it in main one by one. Same goes for transmit side, so buffers are required for holding data coming from ISR till the time when it is processed. If fast serial data is given in then in blocking case buffers will overflow relatively faster compared to non-blocking version as with interrupts data can be transmitted along with receiving.
3) These issues can be tested with unit testing, here constraint randomization can be used to generate a random length buffer, use assertions to check for 0 length error and malloc fail errors. Then push random data over entire length and check for overflow, then pop entire length till underflow. Now check for valid data at correct locations in buffer. This can be done by pushing known sequence of data over entire length and the popping it to check against the sequence. Also buffer resizing can be tested by resizing it with a random length.

Part 4: UART Device Driver

1) In Blocking implementation if no character is received then it waits till character is received. Whereas in non-blocking implementation if no character is received then also it keeps on toggling

LED, calculating Fibonacci series and transmitting data if anything is left to send. GPIO toggle is not done in non-blocking implementation but anything written in infinite loop in main will stop till the next character is received.

2) Blocking Version: Code waits for the character to be received, once received it echoes the character then updates the report and prints it. Then it again waits for next character and process continues.

   Non-blocking Version: Receive ISR loads data in Rx FIFO. In main, if Rx FIFO is not empty then it's a pop's a character from it and pushes it in Tx FIFO, then it computes report and its printings characters are also pushed in Tx FIFO. In main it also checks for Tx FIFO not empty, and in that case it pop's a character and sends it on UART. Whenever Tx interrupt is generated its flag is just cleared. So the receive process goes on and keeps on adding data to be printed in Tx FIFO and transmits process keeps on sending data till Tx FIFO is not empty and UART is ready to accept new byte.
3) There is just a minute difference in user interface for blocking and non-blocking implementation. In blocking it ask's user at start for receive and transmit buffer size whereas in non-blocking case it starts with a default buffer size, as in this case buffers are single point of entry and exit for serial data so no welcome message can also be printed without buffer. Thus they are initialized with default values in this case. Blocking version is more easy to code.

Part 5: Application

1) CPU computes Fibonacci series and toggles LED when it is not processing received byte and has issued transmit data and wait till Tx interrupt so that it can send next byte. Same goes for last received character also.
2) If we don't consider baud rate then buffer size limits transmission speed. As for each received byte we need to echo it, report of this and all previous characters input and Fibonacci value and formatting characters like line feed and carriage return. So basically for each received character there is around 30-50 transmit characters. So this results in Tx FIFO overflowing and loss of data. So buffer size limits the rate. This issue is taken care in my implementation, since my both Rx and Tx works on interrupts and till my buffers are not empty, I just ensure that my Tx FIFO never overflows by skipping receive processing. Thus my received data is secure in my Rx FIFO and I ensure not to overflow my Tx FIFO, so my code is loss free. Only scenario where data loss can come is if Rx FIFO overflows, but that can be taken care by user using buffer resize feature. Also this is NOT achieve at the cost of interface speed, as Tx FIFO always has data so it can keep on sending that and it doesn't have to wait for data to be sent.
3) In my implementation size of Tx buffer has NO effect on report printing as I have checks to ensure that my Tx buffer doesn't overflow. Rx FIFO size should ideally be equal to maximum data that you expect will be received quickly ideally continuously like through file or something till end of file. And Tx FIFO of similar size as Rx FIFO will work with my implementation as anyways it will never overflow. So I have used 500 bytes as default size for both Rx and Tx FIFO.