Project 2

Circular buffer, UART and interrupts

ECEN 5813, Fall 2018

Prof. Kevin Gross

# Introduction

In this project we will write software for the FRDM development board. Software components will include the following:

1. A circular buffer implementation
2. Driver code for the UART
3. An application that responsive to input from the UART that reports statistics of said input in a formatted report to the UART output

# Guidelines

* Follow ESE coding guidelines
* Have boundary checks on arguments and NULL pointer checks if required.
* Use MACROS instead of hard coding values. Macro names should be followed by module name. For example, UART\_xx, CBUF\_xx.
* Work should be committed to the repository throughout development but at a minimum after completion of every part.

# Part 1: Block diagram and architecture

Create design documentation and submit one or more block diagram of your proposed solution. This is to help practice designing a complex software project and to understand the requirements of this project.

The diagram(s) should include both hardware and software components. Review White Ch. 2 for examples of these diagrams. Label each component in the diagram(s).

Your block diagram submission should include the following items:

* A diagram of what your software is and its interactions. Indicate modules and the communication interfaces etc.
* Documentation on needed software structures (for example circular buffer structure type, statistics structure type), in each module if relevant
* A list of the expected needed software interfaces and functions

# Part 2: Circular buffers

The circular buffer uses single, fixed-size buffer as if it were connected end-to-end. Data added to the circular buffer is removed in the same order it was added. For this project, groups implement a circular buffer of characters (int8\_t) and use a unit testing framework to test it separate from the rest of the project.

## Requirements

The following capabilities and interfaces must be present in your implementation:

1. Ability to operate on multiple buffers of different sizes using the same code
2. Add a new item to the buffer
3. Remove oldest item from the buffer
4. Bounds checking and reporting of errors when adding to a full buffer and removing from an empty one
5. Ability to accommodate all 256 legal values for the data it stores

These functions will be used in the Linux development environment for unit testing in part 3 of the project and in the KDS IDE with the UART driver in Part 5 of this project.

## Questions

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

*No, this circular buffer/FIFO implementation is not thread safe because there is a critical section involving the indices. If an interrupt were to occur during the read/modify/write sequence involved in updating either the “put” or “get” index and the interrupt service routine modified the same index for the same FIFO that was being updated, then the index could have the incorrect value at the end of the sequence. The simplest fix for this issue would be to disable indexes around those types of instructions, but this was not considered necessary for this project because the same indices are not being modified in multiple threads.*

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

*If the same buffer is used by both interrupt and non-interrupt code and both threads either write to or read from the buffer, it is possible for some modifications to the buffer indices or pointers to be lost due to the way in which processor updates variables (read value from memory into register, operate on value while in register, store value back to memory address) in a non-atomic fashion. This issue can be addressed by making modifications to variables like the buffer indices/pointers atomic (preventing interrupts from occurring during the modification).*

1. **How could you test these issues?**

*These issues can be tested in multiple ways.*

* *One fairly intrusive way to test these possible issues is to insert software-triggered interrupts (if supported by your processor) between every possible pair of assembly instructions (1 at a time) in your implementation and verify the behavior.*
* *Another somewhat less intrusive way to test these possible issues is to add code to the interrupt service routine to save the address that the processor will return to after completing the ISR to a buffer. Run several tests with several interrupts and verify the behavior of the buffer. Afterwards, examine the buffer with the return addresses to ensure that every possible interruptible location was tested.*
* *Finally, a non-intrusive way of testing this implementation is to return several tests with a high interrupt frequency over a long period of time and verify the behavior.*

## Extra credit

Implement additional capabilities:

1. Ability to resize an existing buffer (1 point)
2. **Ability to report the number of elements in the buffer (1 point)**
3. Implementation that does not reserve any unused buffer space when full (1 point)

# Part 3: Unit testing

Groups will set up a unit testing framework for testing their circular buffer implementation. You must use C-Unit test framework. You should perform unit testing on the Linux platform. Create a makefile to build your circular buffer implementation. Include a target in your makefile called ‘unittest’ which will run all the test cases and display the test results.

Your test cases must cover a variety of normal operation conditions as well as all reportable error conditions.

Your test cases should be in a source file or set of source files separate from the circular buffer implementation.

## Folder structure for Linux project

1. src: should have all .c files
2. inc: should have all .h files
3. Unittest: should have unittest.c file
4. Makefile

# Extra credit

Implement an automated long-running randomized test case (2 points)

The circular buffer project in linux should be affiliated with makefile such that it can support native compilation for HOST as well as cross compilation for KL25Z and Beaglebone. (The makefile must only generate executable, It won’t be executed on KL25Z and beaglebone)

(3 points)

# Part 4: UART device driver

Instead of using the existing library functions (printf(), scanf(), etc.) You will be creating your own UART software driver to transmit data to and from a test terminal. To do this on the FRDM board, you will be using the same USB connector as labeled for the debug interface. This connector has an internal UART to USB converter in the onboard OpenSDA implementation. After downloading code to the target via KDS, you will use a terminal emulator​ ​(like​ ​​ ​Putty​ ​or​ ​Realterm)​ ​to​ ​interact​ ​with​ ​the​ ​board​ ​and​ ​your​ ​code.

This module will support methods to configure and transmit and receive data using the UART peripheral on the Kinetis processor. Implementation will be done in two stages" blocking, using polling, and then non-blocking using interrupts.

## Blocking

In a blocking implementation, your code inquires or polls the UART hardware

Your blocking version must contain the following functionality:

1. Hardware initialization including pin MUX control, clock configuration, baud rate and serial format set up
2. Function to check whether the transmitter is available to accept a new character for transmission
3. Function to transmit a character assuming transmitter is available.
4. Function written in terms of the first two (2 and 3 above) wait (block) for the transmitter to be available and then once available transmits a character and returns
5. Complementary functions to 2, 3 and 4 for receiving a character using the UART
6. A main() function that uses the above functions to echo received characters one at a time back to the sender

## Non-Blocking

In this implementation, you will be using interrupts, instead of polling, to detect the completion of a sending sending or receiving operation. You are encouraged to reuse the code you developed for the blocking implementation for the non-blocking version. Ideally the same code can be used without modification for both implementations such that the non-blocking implementation is simply an improvement on the prior work.

Your non-blocking implementation must contain the following functionality:

1. Hardware initialization as your per blocking implementation with addition of interrupt configuration and enable
2. An interrupt service routine (ISR) function to handle the interrupts. This function should handle both transmit and receive interrupts. Hint: check status register flags to determine the type of interrupt.
3. Function to check whether the transmitter is available to accept a new character for transmission
4. Function to transmit a character assuming transmitter is available
5. Function called by the ISR to supply a next character on completion of the previous
6. Function called by the ISR for receiving a character
7. A main() function that uses the above functions to echo received characters one at a time back to the sender. The function should rapidly toggle a GPIO pin when it is not busy echoing characters.

## 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?**

*In the blocking implementation, while there are no characters waiting to be echoed, the CPU is continuously polling the RDRF bit of the UART0 status register, waiting for a character to be received.*

*In the non-blocking implementation,*

1. **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.**

*Blocking Implementation:*

* *Most likely, when the character is received, the processor was spinning in UART\_BlockInChar() with continuous calls to UART\_RxReady().*
* *Then, UART\_Receive() is called and loads the received character from the UART data register into a memory location.*
* *Then UART\_BlockOutChar() is called with the value that was just received. Within that function, the processor spins on the UART\_TxReady() function until the hardware is ready to transmit another character (unlikely to take very long). Then, UART\_Transmit() is called to complete the echoing process.*

*Non-blocking Implementation:*

* *When the character is received, an interrupt is triggered which calls triggers the execution of the UART0\_IRQHandler(). In this function, UART\_RxReady() is called to verify that the hardware is prepared to receive another character to transmit. If it is, UART\_Receive() is called to load the data from the register to memory.*
* *Afterwards, the flow of execution returns to the place it was interrupted from, which was most likely the main function in which it was toggling the GPIO pin. Now that the new data semaphore indicates that there is new data to be displayed, the main thread will put the character to echo into the output FIFO buffer and enable interrupts form the transmission UART.*
* *Then, the UART\_IRQHandler() will be triggered again (the receive and transmit interrupts use the same ISR). This time, FIFO\_NumElements() and UART\_TxReady() will indicate that there is a data ready to be transmitted and the UART hardware is ready to transmit more data, so UART\_Transmit() will be called to complete the echoing process.*

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

*In the blocking implementation, the interface to the UART driver was very simple. It consisted of the initialization (UART\_Init()), receive (UART\_BlockInChar()), and transmit (UART\_BlockOutChar()).*

*The non-blocking interface was also very simple as it only required the initialization (UART\_Init()), and, adding data to the transmission buffer FIFO*

*Only considering the interface to the main() application code, the non-blocking implementation is easier.*

## Extra credit

Your KDS project makes use of your own custom Makefile (5 points)

# Part 5: Application

Write an application that receives characters from the UART and and keeps track of the number of occurrences of each of the 256 possible characters and sends an updated ASCII formatted report back out the UART for display on a terminal. You do not need to echo the input characters to the output.

The format of the report can be as simple as a two column listing of character and number of occurrences. Characters with no occurrences should be omitted from the report. You may use library functions such as sprintf() to format your report. The final report after typing "Characters" to the terminal could be as simple as follows:

C - 1

a - 2

c - 1

e - 1

h - 1

r - 2

s - 1

t - 1

The application should attempt to update the report after each character received. When characters are being received rapidly, it will, however, be unable to do so. Nevertheless it should still be able to accurately count occurrences of received characters and once the onslaught has subsided, should be able to print an accurate report. You can test your program's ability to gracefully handle an onslaught by pasting a large text file into the terminal emulator.

## Requirements

Your application must be implemented in accordance with the following requirements:

1. You **must** use the non-blocking version of your UART driver developed in part 4 of this project.
2. You **must** use the circular buffer developed in part 2 to hold characters for the report awaiting transmission.
3. You **may** use a circular buffer to hold received characters awaiting tabulation.

## Questions

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

*After the last character has been received, the CPU can finish adding characters to the transmission buffer FIFO and then toggle the GPIO pin while the TDRE interrupt triggers after each character is sent until the last element is removed from the transmission buffer FIFO.*

1. **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 rate at which the application can process incoming characters is the length of time it takes to execute the interrupt service routine. In this system, because the hardware does not contain its own FIFO buffer, data must be copied from the data register (which is done in the ISR) before the next data element is received or it will be lost forever.*

1. **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?**

*The size of the circular buffer determines how “far behind” the output can be with the input. For example, in this application, the input of 1 character can trigger the output of hundreds of characters (a line for every printable character, a hypen, the number of occurrences of that printable character, a new line, and a carriage return (assuming the number of occurrences is at least 1)). Therefore, elements can be added to the transmission buffer several times faster than they can be received through the UART. If the transmission FIFO buffer fills up, then characters will be lost, and the output will be incorrect. The appropriate buffer size for this application will vary depending on how fast the input will be. If the intention is to always wait until the output is finished printing before typing a new character, the size of the buffer only needs to be the product of the number of printable characters supported, and the number of bytes transmitted per line of the report. For my testing purposes, I tested up to 62 different characters at 5 characters per line, so I needed at least 310 bytes in my FIFO ( I rounded up to 350 for a safety factor).However, that number would again need to be multiplied by the number of inputs that can be given while the transmission is being made.*

*In order to alleviate this problem (and a possible solution to the Extra Credit #1), the system could use a timer to limit the frequency at which it attempts to write the report to the screen (instead of generating a report with every new key stroke), therefore limiting the stress on the transmission buffer.*

## Extra credit

1. Avoid any report output corruption (due to circular buffer overflow) regardless of input character rate. (3 points)
2. Do something "useful" in the main loop of your application (e.g. compute digits of pi, fibonacci numbers) and print the results as part of the report. (2 points)

# What to turn in

1. Block diagram and architecture description
2. Code dump report in PDF format
3. Answers to questions posed here in the project description
4. Github repo link, should be tagged.
5. KDS zipped project.

# Grading rubric

1. Block Diagram: 05 points
2. Circular buffer functions: 15 points
3. Unit testing: 10 points
4. UART driver: 20 points
5. Final application: 20 points
6. Answering questions: 10 points
7. Demo: 20 points
8. Extra credits(max): 18 points