**Part 1**

Rx UART

Main

Tx FIFO

Tx UART

Serial Port

Serial Port

New Data Semaphore

FIFO Put

FIFO Get

FIFO Interface:

typedef struct {

uint64\_t putI;

uint64\_t getI;

uint64\_t sizeInBytes;

} FIFO\_META\_DATA\_TYPE;

typedef struct {

FIFO\_META\_DATA\_TYPE metaData;

uint8\_t data[];

} FIFO\_TYPE;

typedef enum {

FIFO\_RESULT\_SUCCESS = 0,

FIFO\_RESULT\_FIFO\_FULL,

FIFO\_RESULT\_FIFO\_EMPTY,

} FIFO\_RESULT;

UART Interface:

FIFO\_TYPE\* FIFO\_Init(uint64\_t sizeInBytes);

FIFO\_RESULT FIFO\_Put(FIFO\_TYPE\* fifo, uint8\_t dataByte);

FIFO\_RESULT FIFO\_Get(FIFO\_TYPE\* fifo, uint8\_t\* dataByte);

/\*

\* uart.h

\*

\* Created on: Nov 22, 2018

\* Author: Sean

\*/

#ifndef INC\_UART\_H\_

#define INC\_UART\_H\_

#include "C:\Freescale\KSDK\_1.3.0\examples\frdmkl25z\demo\_apps\hello\_world\kds\inc\fifo.h"

void UART\_Init(void);

void UART\_BlockOutChar(uint8\_t dataByte);

void UART\_BlockInChar(uint8\_t\* dataByte);

//extern FIFO\_TYPE\* UARTRxFifoPtr;

extern FIFO\_TYPE\* UARTTxFifoPtr;

extern uint8\_t newDataSemaphore;

extern uint64\_t characterHistogram[256]; // statistics structure type

#endif /\* INC\_UART\_H\_ \*/

**Part 2**

## 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:**

1. - **Ability to report the number of elements in the buffer (1 point)**

**Part 3**

* No deliverables listed

**Part 4**

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, the CPU is toggling the GPIO pin. The GPIO pin will toggle extremely quickly while there is no UART activity, but will slow down UART activity begins because the single-core chip can only execute 1 command at a time.*

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.*

**Part 5**

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.*