A Brief Look at Circular Buffers

Erin Payne

4/9/19

ECE 631 Lab 6

Introduction

For this lab we investigated the many uses of circular buffers and how to construct them. Much like a queue, buffers serve as a means to temporarily store sets of data while waiting for use or access to be read. Figure 1 depicts one of the most common fixed buffer structures. This kind of data storage can be very useful for synchronous operations like file reads and writes to allow quick access to large arrays of data without delays caused through the use of system interrupts. Although buffers offer possible increase to the overall runtime efficiency of a particular system, one of its most common constraints is its limited length. With fixed buffers, a system must consider the bounds of the buffer and sequentially what to do in the case of possible overflow. To avoid the corruptions and overwrites that a buffer overflow could create, we can instead utilize circular buffers.

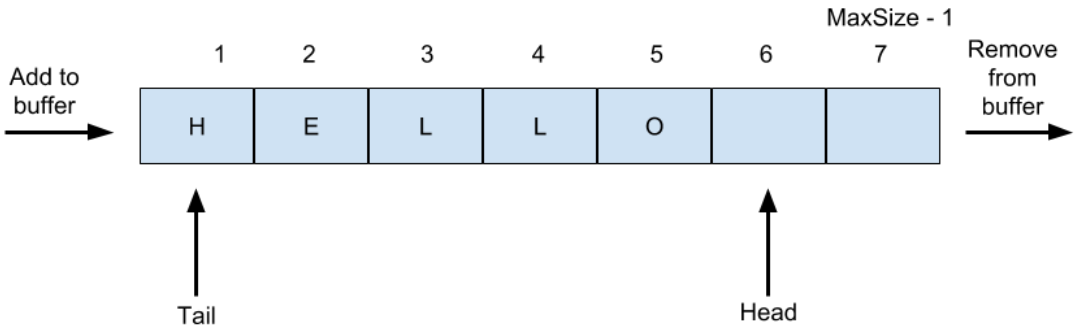

Figure 1.) Conceptual example of a buffer structure with its various parts containing the string “hello”.

Rather than continually adding new data to the end of the buffer until data overflows into neighboring memory spaces, a circular buffer acts as though it is a circle - adding data to the end of the buffer until the maximum length is reached. Once the maximum length is met, the buffer head then returns to start to write the remaining information to the beginning of the buffer. This can be seen visually in figure 2 below. Utilizing continuous looping to write data rather than the original linear approach, we avoid the challenges of buffer overflow along with its corresponding overhead. Furthermore, a circular approach offers solution to the complexities that could be

created in attempting to extend and expand the initial buffer size to meet incoming data write requests.

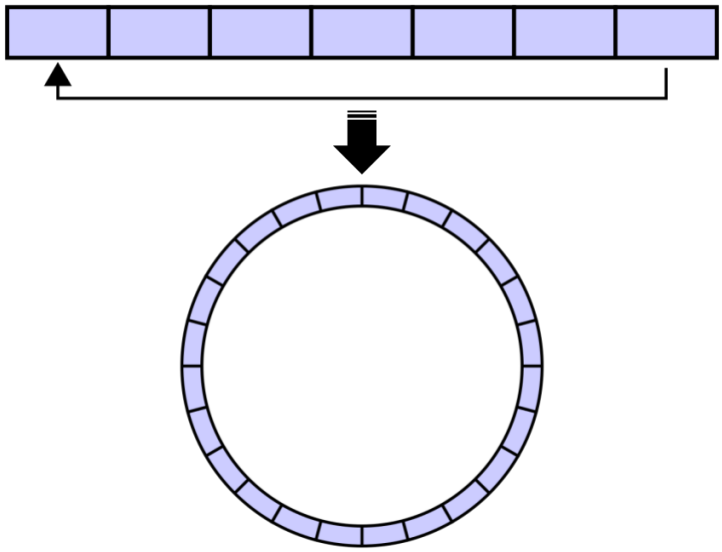

Figure 2.) Visual comparison of how continuous fixed buffer (uppermost) demonstrates the concept of circular buffers (bottomost).

In order to complete the execution of this lab, the main hardware we used was the STM32L475 IOT Node equipped with ARM Cortex M4 core and USB to TTL serial breakout connector cable. Using the included sensors on STM32 IOT Node and the connection made to our personal computers, we were able to perform a variety of small tests to demonstrate our code. To edit and manipulate the board we used the provided OpenSTM32 System Workbench IDE and provided “circularBuffer.h” code provided on Canvas we were equipped to construct our own personal code to implement the circular buffer, TXE and UART4 interrupt handlers. Furthermore we were able to track any and all progress made to edit our code on GitLab online services.

Design Partitioning

After learning the basics of the proper setup of the circular buffer and reviewing the main goals of this lab, I decided to partition my code to focus on the two main components of data

transaction. The two major categories I subdivided my circular buffer to was the transmitting and receiving aspects of message translation. Based upon the code given to us in the .h file provided on Canvas and these two overarching themes, I aimed to edit each of the operational functions given to focus on the reading and writing accesses of data within the circular buffer. After I was able to identify these two partitions, writing the internal code of each function seemed to fall well into place.

One of the first steps I took to subdividing my code into their proper subsections, was to concentrate my attention on the action verbs used within the foundational reading and writing functions of "circularBuffer.c". Take for example the functions "getChar" and "putChar". Notice how one function begins with "get" and the other one with "put". I found these two action verbs to be very telling of which category they belonged to after recalling the main roles of the transmitting and receiving processes. In the case of the circular buffer, if we were to focus solely on the transmitting aspects of data transfer, we know that in order to transmit any message for delivery we would first need to begin with "putting" the requested string of characters into the buffer. Therefore, by this logic, I knew we could dedicate the "putChar" function to focus its internal processes towards the transmitting side of the buffer. Conversely when looking at the receiving end of the buffer, we know that in order to store a message we would first need to "get" the requested message to put into the buffer for later transmission. From this I knew to construct the "getChar" function to revolve around the needs of the receiving buffer - manipulating the head and tail pointers to intake incoming messages.

Design Issues\Limitations

By separating my design into the two major partitions above I was better able to determine the need to create two buffers - one for receiving and one for transmitting. Through this division of message correspondence between two buffers to transmit to the STM IOT node I was better able to analyze each process and their manipulations of the head and tail pointers. Originally I had considered designing my code to only implement one buffer that acted as both the receiver and transmitter for the STM IOT node, however I soon realized that in order to

execute the steps necessary to implement UART to the board, two buffers would be vital to maintaining data exchanges. The diagram depicted in figure 3 below demonstrates the main transaction levels processed from the STM IOT board.

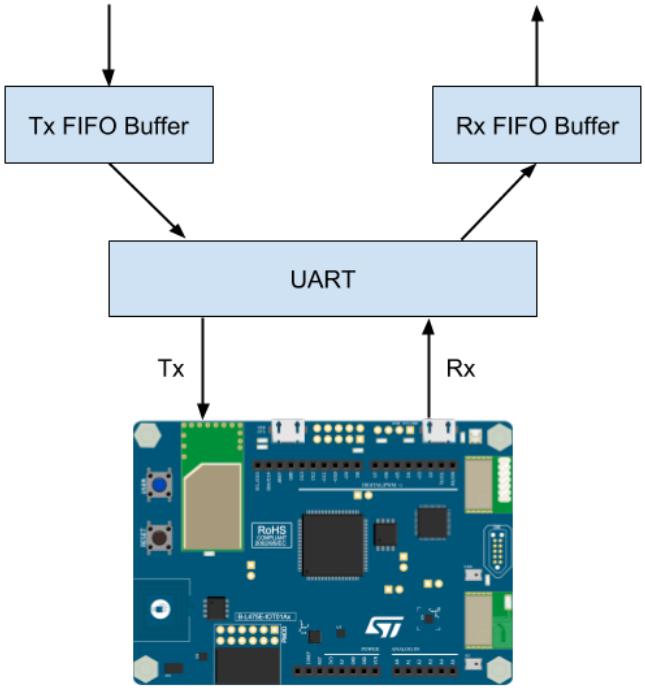

Figure 3.) Levels of data exchange for receiver and transmitter on the STM32 IOT Node.

By dividing the working load of the STM IOT node into two buffers we can then use the UART to asynchronously signal when to begin transmitting and receiving to and from the node. This was a beneficial key to the use of the UART as it ran by a system of interrupts to signal transitions between the receiver and transmitter. With my original design, the lack of separation between the two signals caused an unbreakable loop of sorts that prevented the buffer from transmitting any received messages larger than the original buffer size. This issue was caused as consequence of wrap around by the circular buffer, therefore rewriting over prior data until the head pointer returned to the position of the tail pointer. If in the case the head pointer met a newline character before returning to the tail pointer, then the currently received message would return as trash as it would be unable to be stored without transmission of the prior message

beforehand. With the separation of the receiver and transmitter to two separate buffers this problem was no longer of concern.

Testing

In order to test my constructed code I utilized the suggested testing instructions within the lab write up posted on Canvas. By using the Arduino serial monitor I was able to echo the following commands in figure 4 and determine from the returned code the validity of my design.

Send from serial terminal to STM32L475	Echoed back from STM32L475
Push STM32L475 Reset button	
A	A
B	B
C	C
D	D
E	E
F	F
Push STM32L475 Reset button	
123456	1234
A	A
abc	abc
xy	xy

Figure 4.) Listed tests from provided lab instructions for composed code.

In addition to the brief tests provided in figure 4, I performed a variety of personal tests with an assortment of string messages to test the structure of my circular buffer. Using the included debugging tools in STM Eclipse IDE I was able to follow the individual characters of each string and their transfer between buffers with and without circular wrap dependent around the length of the string.

Furthermore, with each of these personal tests I was able to demonstrate the some of the boundary conditions of my code. One of these error tests was the case of multiple messages received to the receiving buffer without transmission. This case lead to the rare condition when the head pointer reaches a newline character before the tail pointer, signifying that a message has not been transmitted before a new message overwrite. In this circumstance my code would need to properly terminate the incoming message and trash any prior characters from the message saved as rewrites to the buffer. Testing this performance of my design confirmed the necessary precautions to prevent the opportunity for infinite rewrites over unsent messages.

Conclusion

With the addition of circular buffers to the STM32 IOT node, the only improvements I viewed to be fitting would be in the execution of my buffers in the boundary conditions. If possible I would like to find another means to transmit unsent messages from the transmitting buffer to the STM board upon receival of every new message to avoid any garbage returns on the chances rewrites are about to be made to unsent data. One solution I believe would be considerable to bypass this issue is a possible erase of buffer space after transmission with a prior check of the amount of empty slots compared to the amount of incoming characters being received. If the quantity of incoming characters exceeds the quantity of open spaces then we expand the receiving buffer to hold the incoming message until enough space is made available in the transmitting buffer.

Appendix

Link to code in course GitLab: https://gitlab.beocat.ksu.edu/enpayne/lab-6.git

Resource references:

"Basics of UART Communication." Circuit Basics, 11 Apr. 2017, www.circuitbasics.com/basics-uart-communication/.

"Buffer Overflow Attack with Example." *GeeksforGeeks*, 29 May 2017, www.geeksforgeeks.org/buffer-overflow-attack-with-example/.

"Circular Buffer." Wikipedia, Wikimedia Foundation, 21 Mar. 2019, en.wikipedia.org/wiki/Circular_buffer.

"How to Use UART Multitasking ?" AVR Freaks, 14 Jan. 2018, www.avrfreaks.net/forum/how-use-uart-multitasking.