CMPE-250 Assembly and Embedded Programming

Laboratory Exercise Seven

Circular FIFO Queue Operations

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students. Other than code provided by the instructor for this exercise, all code was developed by me.

Shubhang Mehrotra
Performed 11th March 2021.
Submitted 20th March 2021.

Lab Section 1
Instructor: Muhammad Shaaban
TA: Anthony Bacchetta
Aidan Trabuco
Sam Myers
Payton Burak

Lecture Section 1
Lecture Instructor: Dr. Roy Melton

Abstract

The purpose of this exercise was to design and implement a circular FIFO Queue in a Cortex-M0+ assembly language program. The scope of the activity was to write assembly code to handle the queue operations using multiple subroutines. The queue operations implemented in the exercise included enqueuing items, de-queuing items, printing the contents and the status of the queue, and a help command. The program utilizes the subroutines developed previously to handle the string I/O via GetCharSB and the PutCharSB subroutines, and interaction with the KL05Z board using the UART polling subroutines. The activity was successful in implementing a circular FIFO queue of a capacity of 4-character items.

Procedure

A “First-In First-out” data structure is important to handle tasks where the order of operations is integral to the functionality. The queue ensures that the data received first is utilized first, which is different from a simple array type data structure where data can be accessed at any index, or a stack which provides “Last-in First-out” data access. The queue functionality is obtained by maintaining two separate pointers: one for the entry in the queue and the other for the exit from the queue. In our program, the entry pointer is referred to as the In-pointer, and the exit pointer is called the Out-pointer. The queue implemented in the activity achieves circularity by having the two queue pointers wrap around the queue, as necessary.

Initially, both pointers point to the same address at the top of the queue, but as items are enqueued, In-pointer progresses, and as items are dequeued, the out-pointer increments. The queue implemented in the activity has a capacity of up to 4 alpha-numeric characters. Once the queue is full, the in-pointer cannot increment any further and waits for characters to be dequeued from the top to progress. Once an item is dequeued, the In-pointer then wraps around to the top of the queue to wait for the next enqueue command. Similarly, the out-pointer keeps track of its position with regards to the queue length and wraps to the front of the queue after dequeuing an item from the last index. The address of the index just after the end of queue capacity, is stored for quick comparison with the pointers.

The program implemented uses case-insensitive character commands to call a queue functionality. While characters “E” and “D” handle enqueuing and dequeuing operations, other characters such as “P,” handles printing the current queue, “S” prints out the current In-Pointer, Out-Pointer, and the number of items enqueued, and “H” outputs the list of valid commands. When the enqueue command is called, the Enqueue subroutine primarily checks for availability in the queue, after which it obtains the character input using the GetChar subroutine developed previously and stores it in the queue while incrementing the In-pointer. The Dequeue subroutine also looks at the number of items in the queue before dequeuing an item. After verifying queue contents, it pops the value at the Out-pointer address and outputs it to the terminal using the PutChar subroutine.

With each enqueue and dequeue operation, the In-pointer, Out-pointer, and the number of items enqueued are printed along with a status message of “Success” or “Failure.” Their output is handled by the PutNumHex and the PutNumUB subroutines, which output the correctly formatted numbers to the terminal screen. The same subroutines are also called when the status of the queue is requested by the user.

The Print command uses the PutChar subroutine to output each item in the list within “><” delimiters. It traverses between the In-pointer and the Out-pointer, wrapping around as needed, to ensure queue is printed in FIFO order.

Results

The code written for the 4-character circular FIFO queue operations was translated to machine code and then downloaded to the Freedom Board to ensure desired operation. A PuTTY environment was used to obtain the output of the run with various commands, as shown in Figure 1.

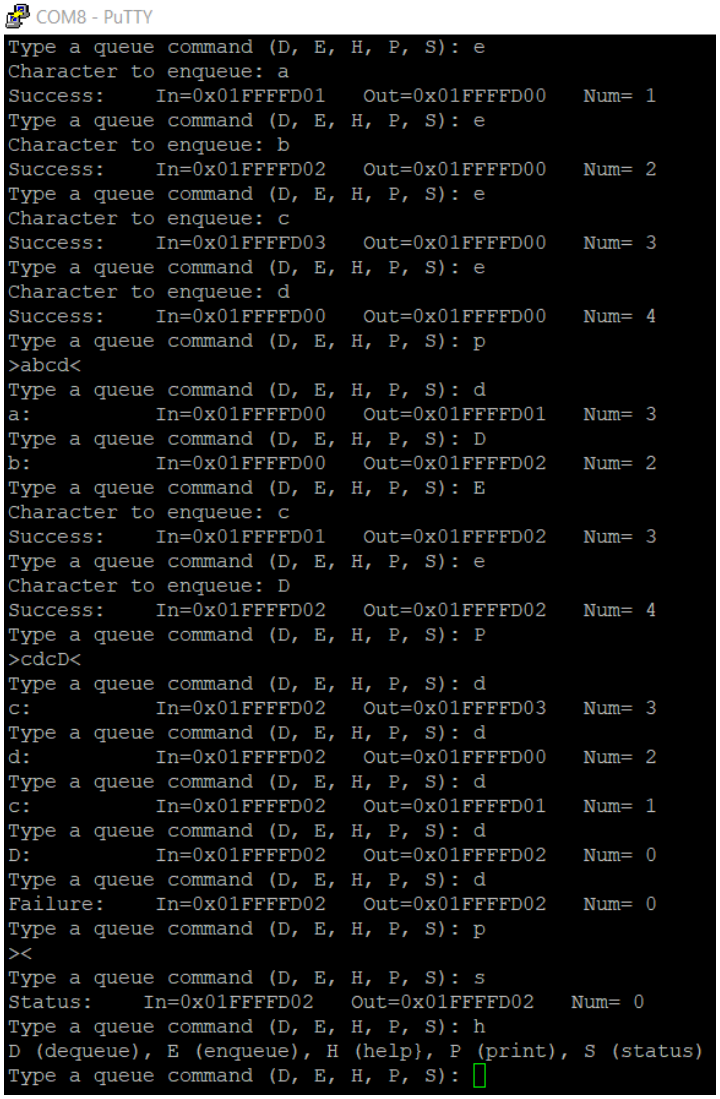

Figure 1. Output

The code was tested against a variety of inputs to ensure maximum functionality. From the first row onwards, it is evident that the enqueue operations works for 4 characters and then fails when a fifth character is tried to be enqueued. This is the desired result as the queue has a capacity for only 4 items. With each enqueue operation, the In-pointer, and the number enqueued can be seen incrementing by one and at the 4^{th} command the in-pointer wraps back to the first index, hence ensuring circularity in the queue. Next, we see that the print command outputs the contents of the queue in the order they were input, thus signifying a successful implementation.

The Dequeue command is also successfully implemented as with each call, it outputs the item at the top of the queue, decreases the number of items enqueued and increments the out-pointer to point to the next valid address. The circularity is maintained here as well, which can be seen through the wrapping of the Out-pointer.

The status of queue with each operation is printed in the desired manner as well, with the addresses in the preferred Hexadecimal format and the number of items enqueued in the correct decimal output. As well as the Help command lists the commands in the proper manner.

The translation of the code generates a listing file for the assembly code, and building it generates the map file. The contents of the two files were analyzed to obtain the memory addresses of various objects in the code. For instance, the map file lists the execution addresses in the ROM for each entity in code. The execution address, along with the size of the entity give information about the exact address range used by a particular object. The data obtained for each object is listed in Table 1.

Table 1. Memory ranges

Object	Memory Address Start	Memory Address End (Start + Size - 1)	Size
MyCode AREA	0x00000410	0x000008CF	0x000004C0
Constants in ROM	0x000001C4	0x0000028B	0x000000C8
Queue Record structure in RAM	0x1FFFFD04	0x1FFFFD15	0x00000012
Queue Buffer in RAM	0x1FFFFD00	0x1FFFFD03	0x00000004

From Table 1, it can be seen that the instructions written for the Main program began at memory address 0x00000410 and went all the way up to address 0x000008CF, taking 1216 bytes in the memory. The constants defined in the program occupied 200 bits starting from address 0x000001C4 and going up to address 0x0000028B.

The variables used by the program occupy space in a much more volatile system of the RAM. The Queue Record structure implemented in the program occupied 18 bytes in RAM and the Queue buffer occupied only 4 bytes.

Conclusion

The activity was successful in designing and implementing a circular FIFO queue data structure for a KL05Z board using assembly language. The subroutines defined in the program functioned as expected in providing targeted functionality to each queue command. While the queue produced was successful for 4 characters, it can be easily scaled with minimal changes to the code to provide the required capacity to the queue. The output to terminal was handled by subroutines defined in previous activities, which also demonstrates their robustness and usability in future programs.