Introduction

For our sixth assignment, we will continue working on the Nucleo-64 board. This assignment will focus on developing a full command interpreter running over USART2, using interrupts and queues. You should be able to base part of your solution on code you developed originally in Assignments 1 and 2, although you will probably need to make modifications.

For this assignment we are going to cast aside USART2 initialization code that was available in earlier projects – instead you will be writing this code based on the example provided by Dean. The clock information is already available in the main.c and the clocks have been initialized and setup.

A template to start your project is provided in the Github classroom project.

As before, you will develop a proper GitHub project for this assignment, complete with code, documentation, and README. Your final code should be tagged "ready-for-grading" before the due date.
Details of the assignment are given below.

Assignment Background

To allow your company's development and QA teams to more fully exercise a new board under development, you have been given the task of setting up an initial command processor running over the serial line.

You will need to develop the following:

1) **A circular buffer queue**. You will need two circular buffer queues, one each for the transmit and receive directions. You should reuse the implementation from Assignment 2 and generalize it to use two static buffers of size **128 bytes** each.

2) **Test code to exercise your circular buffer queues**. You may wish to adapt the automated tests you created for Assignment 2. This code should run at startup if the DEBUG define is set in your code, in order to give you confidence that your queue buffers are solid.

3) **Code to configure USART2 and send and receive data over it**. Parameters for this assignment are specified below. Your implementation should be fully interrupt based[1]. The USART2 solution should be built atop your queue implementation.

4) **Glue code that ties your USART2 communication code into the standard C library functions**. After this glue code is working, a call to `printf()` or `putchar()` on the device

[1] You may wish to implement a poll-based approach at first as you are developing the code, because it is simpler. You should remove the poll-based approach once you get the interrupt method working.

should result in characters being sent over UART2 to the PC, and a call to `getchar()` should result in reading a character that the user typed on the PC.

5) **A command processor** that can accept some very simple interactive commands (specified below) and take action on the device.

Interactive Commands

The command processor running on your device should start by printing a double arrows ("$$ "). Note the space after the "$$". It should then wait for a command typed by the user, and then respond to that command.

As each character is received by your command processor loop, you will need to echo that character back so that the user can see it printed in his or her terminal window. You will need to handle the backspace and delete keys but do not worry about arrow keys.

Each user command should be terminated by a newline.

Command	Arguments	Description
Echo	String of chars	Echoes the same string back in upper case but removes any extra spaces between words/tokens. See example below. For missing parameters, it just echoes an empty line.
LED	ON/OFF	Turns the User LED ON or OFF on the Nucelo-64 board. This is not the external LED.
HEXDUMP	Start, Len	Prints a hexdump of the memory requested, with up to 8 bytes per line of output. See below for the required output, which is very similar to that used in Assignment 1. Start should always be interpreted in hexadecimal. Start will not have a 0x prefix in our grading. Len should be interpreted in decimal, unless it begins with the characters 0x, in which case it should be interpreted in hexadecimal. Valid values for Len range from 0 to 640 decimal, inclusive. It will be possible for a user of the program to crash the program, by providing an illegal address for Start. Your code does not need to catch this error.

All commands and arguments should be processed in a case-insensitive manner, including any hex characters that the user enters. Additionally, the command processor should be forgiving about whitespace; the user should be able to have leading or trailing whitespace around the command, and multiple consecutive whitespace characters should be treated as one for the purposes of separating tokens.

If the command processor receives any command it doesn't understand, it should print the string "Unknown command(<command as received>)". For any errors in input parameters or execution errors, print out the error string of your choice. The command processors should read the entire input line, check for errors, execute the command and should give an error output as described if necessary. Thereafter, the command processor prints the prompt("$$ ") for the next input line and then start the execution of the command.

You should leverage the LED code, hexdump and cbfifo code from previous assignments with some modification.

You don't need to allow any new command input when the current command execution is in process.

Here is an example of a valid command session:

```
$$ Welcome to SerialIO!
$$ LED ON                                    // Turns on the user led
$$ Echo The       Sky            is blue
THE SKY IS BLUE
$$ LED OFF                                    // Turns off the user led
$$    Garbage command
Unknown command(Garbage)                  // Unknown command
$$ HEXDUMP 0 64
0000_0000  00 30 00 20 D5 00 00 00
0000_0008  43 01 00 00 31 27 00 00
0000_0010  00 00 00 00 00 00 00 00
0000_0018  00 00 00 00 00 00 00 00
0000_0020  00 00 00 00 00 00 00 00
0000_0028  00 00 00 00 47 01 00 00
0000_0030  00 00 00 00 00 00 00 00
0000_0038  49 01 00 00 4B 01 00 00
$$ Hexdump a0 0x20
0000_00A0  0F 02 00 00 17 02 00 00
0000_00A8  1F 02 00 00 27 02 00 00
0000_00B0  2F 02 00 00 37 02 00 00
0000_00B8  3F 02 00 00 47 02 00 00
```

Implementation Details

For the circular buffer queues, you may wish to reuse both your Assignment 2 implementation and your Assignment 2 test code, although you will need to enhance this solution to add two circular buffers, which are statically allocated. You will also need to handle race conditions in the circular buffer updates as it is getting written and read and may cause a race condition giving erroneous output.

In order to make the code a bit different from that given as an example in Dean Ch8, we will use the following serial parameters:

Baud rate	19200
Data size (Terminal setting)	8
Parity	Odd
Stop Bits	1

You should also be aware that the system clock parameters we have provided are different than what Dean's solution is based upon. Please go through this code to understand the clocks.

All four of the above serial parameters should be defined in a single place (probably at the top of USART.c or similar) using well-commented #defines.

You will need to define two functions in order to tie your USART2 code in with the standard system I/O functions such as printf(), getchar(), and so forth:

```
int __io_putchar(int ch);
```

Writes the specified bytes to either stdout. Returns -1 on error or 1 on success.

```
int __io_getchar(void);
```

Reads one character from the serial connection and returns it. Returns -1 if no data is available to be read.

In previous assignments, you have created DEBUG and RUN build targets with slightly different behavior. Because SerialIO is intended only for internal use during development, for this assignment you only need the DEBUG build target.

Submission Details

Your primary submission is due on **Wednesday, November 13, at 9:00 am**. At that time, you should submit the following to Canvas:

- Your code must be tagged with "ready-for-grading" before the due date.
- Two screenshots: The first showing the terminal parameters you use to connect, and the second showing your interactive terminal session, similar to the excerpt above. You should run the following commands:
  - Echo Good Day!
  - echo good day!

- lEd oN // Four spaces between d and o
- Led Off
- HEXDUMP 0 64 (note, decimal second argument)
- HEXdump a0 0x20 (hex second argument)
- hexdump a0 yyy
- hexDUMP yyy 55

Grading

Points will be awarded as follows:

Points	Item
5	Did you submit both screenshots to Canvas as requested?
25	Overall elegance: Is your code professional, well documented, easy to follow, efficient, and elegant?(15) Is each module (USART, LED, hexdump, cbfifo, command processor) located in its own file, with an appropriate .h file to expose the interface?(15)
30	Does the USART code allow text to be output to the terminal window reliably, using 19200 baud rate, 8 data bits, odd parity, and 1 stop bit?(10) Are these values easily changed by altering well-documented #defines in your code?(5) Is this code interrupt-based and tied into printf correctly?(15)
30	Does your command processor work as specified? (5) Are prompts printed correctly (with no missing or extra CRs or LFs)?(5) Are characters echoed back to the user correctly?(5) Can the commands be entered in either case(2), with spaces(2) and is the backspace handled correctly(3)? Does the “echo” and “led” command work correctly with error cases? (10)
10	Is it trivially easy to add a new command to your command processor?(5) Can the command processor accept new commands while executing a current command as described above.(5)

Good luck and happy coding!