Introduction

The second assignment will focus on different ways to implement FIFOs. You will also develop a proper GitHub project for this assignment, complete with test harness, documentation, README, and Makefile. The assignment is due on February 7 at 11:59 pm Mountain time, and should be submitted on Canvas. Your submission will consist of two parts:

- Part 1 is a single private GitHub repo URL which will be accessed by SAs to review your code and documentation. For this assignment, this repo will have multiple .c and .h files, test cases, and a Makefile. Details are below.

- Part 2 will be a PDF containing all C code and README documentation. As before, the PDF is used specifically for plagiarism checks: Your code should be yours alone, developed by you. You should provide a URL for any significant code taken from a third-party source, and you should not take code artifacts from other students from this or prior semesters.

You may consult with other students, the SAs, and the instructor in reviewing concepts and crafting solutions to problems (and may wish to credit them in documentation).

Again for this assignment, you may NOT use standard C library functions, including the printf family, in your production code. However, you may use such functions in any of the test code. (You may also use malloc() and free() in your production code.)

Background

A FIFO object—often called a *queue* instead—is intended to connect some code which *produces* data with some code which *consumes* data. This design pattern is relatively widespread; it is commonly used when producer and consumer threads run asynchronously with respect to each other.

One common use of a FIFO might be some video capture hardware. In this example, there is a producer thread that captures a frame of uncompressed video from a hardware buffer in response to either a timer or an interrupt. The producer thread *enqueues* a pointer to the memory containing the video frame onto the FIFO. Asynchronously, meanwhile, a consumer thread is *dequeuing* video frames from the FIFO in order to encode those frames using h.264 (and eventually transmit them across a network). Because uncompressed video is a firehose of data[1], we want to minimize copying it, so our two threads will pass pointers to the video across the FIFO. Presumably the two threads will manage this memory—malloc'ing and freeing it, perhaps—outside the context of the FIFO.

A different possible use of a FIFO might be for bytes being read from a UART. In this example, the producer is likely to be the interrupt service routine (ISR) for the UART hardware. When one or more

[1] The data rate of fully uncompressed "1080p60" video is 1920 (horizontal resolution) ×1080 (vertical resolution) × 24 (bits per pixel) × 60 (frames per second) = ~3 Gigabits per second. In practice this video is often subsampled at the hardware level, but even with the subsampling it can be quite a bit of data.

bytes arrives at the UART and the interrupt is triggered, the ISR will enqueue the bytes onto a FIFO. Meanwhile, a consumer thread will periodically wake up and dequeue bytes that have arrived.

Although both of these examples use a FIFO object to separate a producer and consumer, there are significant differences:

Video FIFO	UART FIFO
Passes pointers to the data across the FIFO, not the data itself.	Copies the data across the FIFO.
Inherent atomicity: Both the producer and the consumer only deal with a *single frame* of video at a time.	The data is inherently a stream, and the producer and consumer will likely want to work with different size chunks of it. Typically, an ISR runs so quickly (relative to the speed of the UART) that the ISR will usually enqueue only one byte of data at a time, whereas the consumer might run much less frequently but deal with tens of bytes of data at a time.
Given the size and processing complexity of the video data, doesn't really care about the cost of using malloc() – this application is not going to be constrained by a few bytes of memory.	Absolutely cannot call malloc from within a FIFO enqueue function, since the enqueue function itself is likely to be called from the ISR, which is running with interrupts disabled. Moreover, on a small embedded system, we probably want to avoid malloc and free altogether.
The system might have multiple active instances of the FIFO object, and FIFO instances may be created and destroyed many times while the system is running.	The system only needs a single global instance of the FIFO object, which can therefore be statically initialized.

Because the requirements of the two applications are different, it makes sense that there might be differing FIFO implementations, and indeed that is the case. For this assignment, you will implement two FIFO objects:

a) **`llfifo`**: A linked-list implementation of a FIFO, suitable for the video example

b) **`cbfifo`**: A circular-buffer implementation of a FIFO, suitable for the UART example

Details for each of the implementations follow.

Assignment Detail

In normal use, considerable care must be taken to ensure that a FIFO implementation is thread-safe, since in the intended use, the producer and consumer threads could be calling the FIFO API simultaneously—or, conversely, perhaps one or both of these threads is an ISR that could run at any point. However, for this assignment we will ignore thread safety issues.

Throughout this assignment I am using the following terms:

- "Capacity" refers to the total size of the FIFO – how many elements could it hold, given its current memory usage?
- "Length" refers to the number of elements currently enqueued on the FIFO

With these definitions, it follows that 0 <= length <= capacity.

`llfifo`: Linked-list FIFO

The `llfifo` implementation will be capable of growing dynamically through an efficient use of malloc(). The caller will suggest an initial capacity when the FIFO is created, but if the caller subsequently attempts to enqueue more elements onto the FIFO than the current capacity would support, `llfifo` will silently increase its capacity up to the limits of available memory. Note that the capacity of an `llfifo` instance will not shrink until the FIFO is destroyed.

The FIFO should implement the following API:

```
llfifo_t *llfifo_create(int capacity);

int llfifo_enqueue(llfifo_t *fifo, void *element);

void *llfifo_dequeue(llfifo_t *fifo);

int llfifo_length(llfifo_t *fifo);

int llfifo_capacity(llfifo_t *fifo);

void llfifo_destroy(llfifo_t *fifo);
```

Details for each of these functions may be found in the file llfifo.h.

Note that it is possible for an `llfifo` buffer element to be created, but to be currently unused. The API is designed this way because we want to use malloc efficiently—we don't want to call malloc() on every enqueue operation and call free() on every dequeue operation. Instead, your FIFO will need to have some means for keeping track of these already-created-but-currently-unused elements.

As an example, let's say the caller creates a new `llfifo` by calling **`llfifo_create(5)`**. Immediately after this function returns, the `llfifo` will have created 5 FIFO elements internally. However, at this point, the FIFO is empty, so those 5 elements are in this already-created-but-currently-unused state. The following code snippet extends the example:

```
llfifo_t my_fifo = llfifo_create(5);
assert (llfifo_length(my_fifo) == 0);
assert (llfifo_capacity(my_fifo) == 5);
llfifo_enqueue(my_fifo, "Sleepy");   // does not result in a malloc (len = 1, cap = 5)
llfifo_enqueue(my_fifo, "Grumpy");   // does not result in a malloc (len = 2, cap = 5)
llfifo_enqueue(my_fifo, "Sneezy");   // does not result in a malloc (len = 3, cap = 5)
llfifo_enqueue(my_fifo, "Happy");    // does not result in a malloc (len = 4, cap = 5)
llfifo_enqueue(my_fifo, "Bashful");  // does not result in a malloc (len = 5, cap = 5)
llfifo_dequeue(my_fifo);             // removes "Sleepy"; len = 4, cap = 5
llfifo_enqueue(my_fifo, "Dopey");    // does not result in a malloc (len = 5, cap = 5)
llfifo_enqueue(my_fifo, "Doc");      // DOES result in a malloc (len = 6, cap = 6)
```

`cbfifo`: Circular Buffer FIFO

The `cbfifo` implementation will be statically allocated, with a size of 128 bytes. (You should make this constant a #define in your code, however—our marketing department is notoriously fickle.) The FIFO should implement the following API:

```
size_t cbfifo_enqueue(void *buf, size_t nbyte);

size_t cbfifo_dequeue(void *buf, size_t nbyte);

size_t cbfifo_length();

size_t cbfifo_capacity();
```

Details for each of these functions may be found in the file cbfifo.h.

General Assignment Notes

For each of the FIFO implementations, you may wish to create an additional function for debugging, called for instance “cbfifo_dump_state()”. This function might use printf() to output all the internal state of the FIFO. This function is not a requirement of the assignment, but it will probably save you time in debugging.

Each FIFO should have a corresponding test function, for instance test_llfifo(). This function should thoroughly exercise the corresponding FIFO. It can either return 1 if all tests succeeded, and 0 otherwise; or it can call assert() on a test failure (in which case, the function returning indicates that all tests succeeded). As before, try hard to ensure that the tests cover all the possible cases for the FIFO.

Your main() routine should call all the test functions.

All functions should check for invalid inputs and return appropriate errors.

Your submission should include at least the following files:

```
llfifo.h*, llfifo.c
cbfifo.h*, cbfifo.c
test_llfifo.c
test_cbfifo.c
main.c
```

```
    Makefile
    README.md
```

(Files marked with a * are provided for you.)

Development Environment

For this project, you may use any version of the GCC compiler. If you're not directly using a Linux system, I recommend using VirtualBox to create a virtual machine running a recent version of Ubuntu, which will have the full GCC environment installed. As an alternative, you may use GCC tools for a Mac (which is free; Google "Command Line Tools for Xcode") or a Windows PC (using Cygwin or MinGW). See the SAs for any assistance you need in establishing a development environment.

Your code should follow the ESE C Style Guide (posted on Canvas) as closely as possible.

When compiling with GCC, use the -Wall and -Werror compiler flags at a minimum. Your code should have no compilation errors or warnings.

Grading

Points will be awarded as follows:

- 20% for overall elegance: Is your code professional, well documented, easy to follow, efficient, and elegant? Does it follow the ESE style guide? Is code needlessly duplicated?
- 30% for the correctness of the llfifo implementation
- 10% for the test coverage of the llfifo implementation
- 30% for the correctness of the cbfifo implementation
- 10% for the test coverage of the cbfifo implementation

Remember, if you do a good job on your test cases, you will almost certainly get the code entirely correct, as well. Good luck!