Lab: Queue

Part 1

Write the unit-tests first, and then the implementation for the following header file:

```
#pragma once
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
/* In this part, the queue memory is statically defined
 * and fixed at compile time for 100 uint8s
 */
typedef struct {
  uint8_t queue_memory[100];
  // TODO: Add more members as needed
} queue_s;
// This should initialize all members of queue_s
void queue__init(queue_s *queue);
/// @returns false if the queue is full
bool queue__push(queue_s *queue, uint8_t push_value);
/// @returns false if the queue was empty
bool queue__pop(queue_s *queue, uint8_t *pop_value);
size_t queue__get_item_count(const queue_s *queue);
```

Students often time create non optimal and incorrect implementation of a queue. Remember that a queue means FIFO data structure, which means oldest item pushed should be the first one out of the pop operation. Here are some unit tests that you are required to add to your test. This test will ensure that your implementation is correct.

```
void test_comprehensive(void) {
  const size_t max_queue_size = 100; // Change if needed
```

```
  for (size_t item = 0; item < max_queue_size; item++) {
    const uint8_t item_pushed = (uint8_t) item;
    TEST_ASSERT_TRUE(queue__push(&queue, item_pushed));
    TEST_ASSERT_EQUAL(item + 1, queue__get_item_count(&queue));
  }

  // Should not be able to push anymore
  TEST_ASSERT_FALSE(queue__push(&queue, 123));
  TEST_ASSERT_EQUAL(max_queue_size, queue__get_item_count(&queue));

  // Pull and verify the FIFO order
  for (size_t item = 0; item < max_queue_size; item++) {
    uint8_t popped_value = 0;
    TEST_ASSERT_TRUE(queue__pop(&queue, &popped_value));
    TEST_ASSERT_EQUAL((uint8_t)item, popped_value);
  }
  // Test wrap-around case
  const uint8_t pushed_value = 123;
  TEST_ASSERT_TRUE(queue__push(&queue, pushed_value));
  uint8_t popped_value = 0;
  TEST_ASSERT_TRUE(queue__pop(&queue, &popped_value));
  TEST_ASSERT_EQUAL(pushed_value, popped_value);

  TEST_ASSERT_EQUAL(0, queue__get_item_count(&queue));
  TEST_ASSERT_FALSE(queue__pop(&queue, &popped_value));
}
```

Part 2

Write the unit-tests first, and then the implementation for the following header file. This is a slight variation of Part 1 and it provides you with the static memory based programming pattern popular in Embedded Systems where we deliberately avoid allocating memory on the heap.

```
#pragma once
#include <stdbool.h>
```

```
#include <stddef.h>
#include <stdint.h>
/* In this part, the queue memory is statically defined
 * by the user and provided to you upon queue__init()
 */
typedef struct {
  uint8_t *static_memory_for_queue;
  size_t static_memory_size_in_bytes;
  // TODO: Add more members as needed
} queue_s;
/* Initialize the queue with user provided static memory
 * @param static_memory_for_queue This memory pointer should not go out of scope
 *
 * @code
 *   static uint8_t memory[128];
 *   queue_s queue;
 *   queue__init(&queue, memory, sizeof(memory));
 * @endcode
 */
void queue__init(queue_s *queue, void *static_memory_for_queue, size_t static_memory_size_in_bytes);
/// @returns false if the queue is full
bool queue__push(queue_s *queue, uint8_t push_value);
/// @returns false if the queue was empty
/// Write the popped value to the user provided pointer pop_value_ptr
bool queue__pop(queue_s *queue, uint8_t *pop_value_ptr);
size_t queue__get_item_count(const queue_s *queue);
```

Requirements

- Test thoroughly
  - Do not hack internals of a module.
  - This means that only operate using the APIs, and do not modify the data structure
  - As an example, to test `pop()`, push elements using the API rather than hacking `struct.write_index++`
- Create a thorough test like this one at the end of your basic tests:
  - Push to the capacity of the queue
  - Then pop all elements
  - Finally push value of 0x1A and pop value of 0x1A
- Do not "shift" any elements in your `pop()` operation
  - Keep track of read and write indexes separately

    - It would be horrible pop operation that has to shift thousands of elements over by 1
- Pop test should explicitly test to make sure the popped value is what was pushed
    - This means that the `pop()` API depends on the `push()` API to work

---

Advanced API Design

We can also experiment with an "iterator" based API design pattern in C which involves a function pointer and callbacks. This is an optional section that does not need to be addressed in your lab.

```
// lab_queue.h:
typedef void (*queue_callback_f)(uint8_t item);
// API to iterate through each item in the queue
// Note that this would not pop any items
void queue__iterate_items(queue_s *queue, queue_callback_f callback);
```

Implementation for the iterate API would be something like the following:

```
void queue__iterate_items(queue_s *queue, queue_callback_f callback) {
  if (NULL != queue) {
    size_t index = queue->pop_index;

    for (size_t count = 0; count < queue__get_item_count(queue); count++) {
      callback(queue->queue_memory[index]);
      ++index;
    }
  }}
```

The unit-testing is where things get a little more interesting. Naive way of unit-testing would be:

```
 static int callback_count;
static void callback(uint8_t item) {
  ++callback_count;
  if (1 == callback_count)
    TEST_ASSERT_EQUAL(12, item);
  if (2 == callback_count)
    TEST_ASSERT_EQUAL(34, item);
  if (3 == callback_count)
```

```
    TEST_ASSERT_EQUAL(56, item);
  printf("Item: %d\n", item);
}
void test_queue__iterate_items(void) {
  queue__push(&queue, 12);
  queue__push(&queue, 34);
  queue__push(&queue, 56);
  queue__iterate_items(&queue, &callback);}
```

More advanced method of unit-testing would be:

```
void test_queue__iterate_items_with_stub_v2(void) {
  queue__push(&queue, 12);
  queue__push(&queue, 34);
  queue__push(&queue, 56);
  queue_callback_stub_Expect(12);
  queue_callback_stub_Expect(34);
  queue_callback_stub_Expect(56);
  queue__iterate_items(&queue, queue_callback_stub);
}
```

In order to get the `queue_callback_stub_Expect()` framework, you need to create this file and then mock it at your unit-test file. Note that this file is a header only file, and we merely need it to do `#include "Mockqueue_callback.h"` that is provided below.

```
#pragma once
void queue_callback_stub(uint8_t lab243);
```

Revision #5
Created 2 years ago by Preet Kang
Updated 2 years ago by Preet Kang