Instructions - Laboratory assignment 2 ht22

Johan Holmgren, 2018-09-14 - Created assignment Johan Holmgren, 2020-09-14 - Modification to work with VSCode, PlatformIO, and ESP32 board.

Overview and purpose of task

The purpose of this assignment is to learn about buffers, which is a useful data structure in embedded systems. In particular, you will implement a first-in-first-out queue using a circular buffer.

You will conduct this assignment either indivudally or in pairs of two students; however please note that the examination is individual.

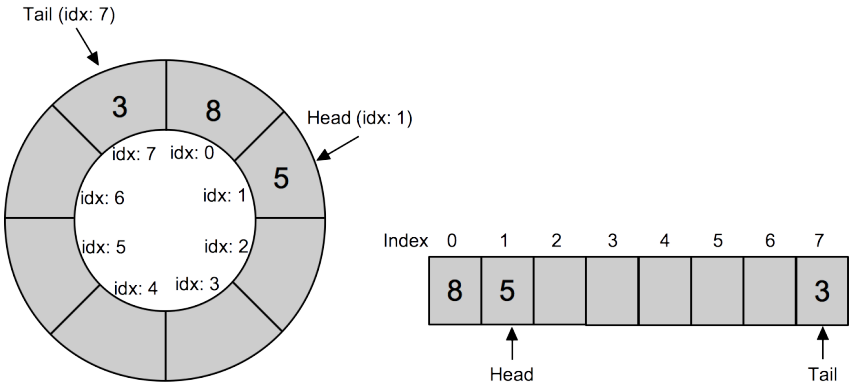
Introduction to circular buffers

A circular buffer is a data type, which can be described as a fixed size buffer whose start and end are connected to form a circle. A circular buffer is typically implemented using a normal array, where the elements that are currently represented in the buffer are stored in sequence somewhere in the array. An important feature of a circular buffer is that the sequence of elements can start and end anywhere in the array. A circular buffer is typically implemented using two pointers (head and tail), defining which positions in the array are currently used to store the data.

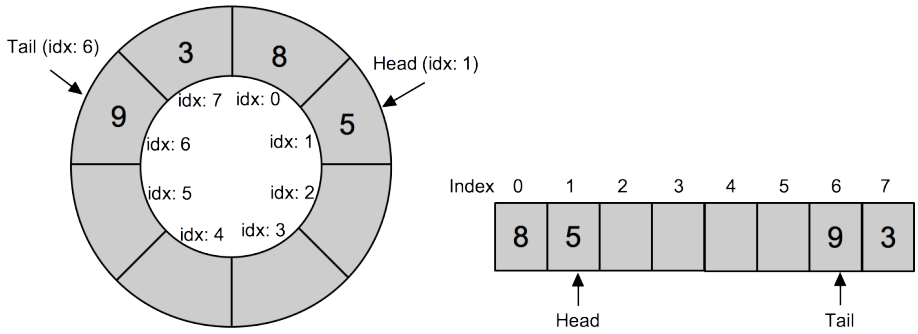
A typical use of a circular buffer in embedded system is to represent a queue, for example, in order to store input from a sensor that is handled in a *first-come-first-served* manner by the system. When adding and removing elements, the queue will obviously change size, and more importantly, it moves around in a circular way. It should be emphasized that elements are always added to the tail and removed at the head.

Using the queue terminology, *tail* points to the most recently added element, i.e., the last element in a first-come-first-served queue, and *head* points to the oldest element in the queue, i.e., the first element in a first-come-first-served queue.

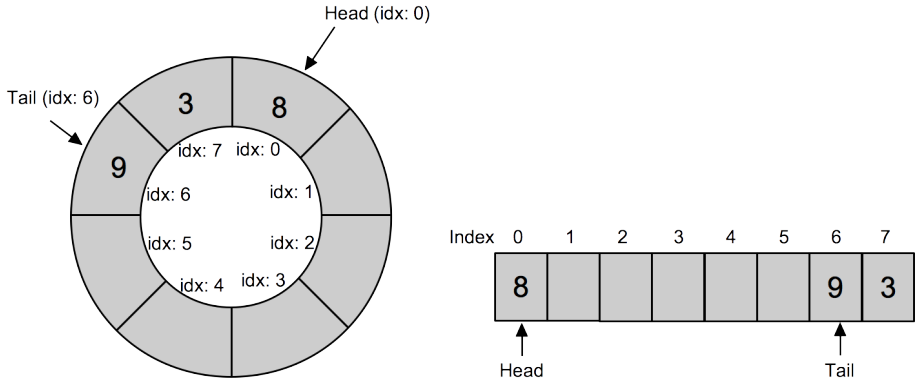
The following image illustrates a circular buffer queue with the values 5 (the first element), 8, and 3 (the last element).



The following image shows what happens when we add the element 9 to the buffer/queue.



Finally, the following image shows what happens when we remove the element at the head.



Please note what happens with the head and tail pointers when we add and remove elements. By studying the images you can also see how the sequence of elements moved one step to the left when adding and removing one element.

Description of task

Your task in this laboratory assignment is to implement a number of functions that allow you to work with a circular buffer containing `int` (integer) values.

To help you get started, you are provided a visual studio code project using PlatformIO with the following settings:

- platform: espressif32
- board: nodemcu-32s
- framework: espidf
- monitor_speed: 115200

You find the "startup code" in a zip file on Canvas. To get started, you import the existing project into visual studio code. The project contains the following source files:

- `circular_buffer.h` - Header file specifying all of the functions you should implement.
- `circular_buffer.c` - The file where you implement the functions specified in `circular_buffer.h`
- `main.c` - The file where you write the code to test your circular buffer.

Pleae note that the code should run on your ESP32 board, and you should illustrate that your circular buffer works correctly by applying all of the implemented functions, and test your functions according to the test requirements specified below.

Implement a circular buffer

As mentioned above, you are provided a c header file `circular_buffer.h`, which contains a circular buffer data structure and a number of function declarations.

The circular buffer data structure is defined in the following way:

```
struct circularBuffer{
  int * data;
  int head;
  int tail;
  int maxLength;
};
```

Given the description above of a circular buffer queue, you should implement the following functions, which are declared in`circular_buffer.h`.

- `void initCircularBuffer()`
- `int contains()`
- `int addElement()`
- `int removeValue()`
- `int removeHead()`
- `void printBuffer()`

Specifications of each of these functions are given below.

An important problem you need to solve is the "empty or full" problem. You implement your circular buffer using head and tail, which will point to the same place in the array both when the buffer is full and when the buffer is empty. As part of the assignment you should identify a solution for this problem. Please note that this means that you might need to make some changes to the `struct circularBuffer` data structure.

Another problem is to decide what to do when the circular buffer is full. In particular, when you need to add an element to a buffer that is already full you need to decide whether you should overwrite one element in the buffer or if you should discard the element that you are about to add. If you decide to overwrite an existing element, you also need to decide if you should overwrite the element at the tail or at the head. During examination, you should be able to motivate your approach of how to manage the problem of having a full buffer.

For this task, you can assume that there will be always sufficient amount of memory available for your buffer. However, in an embedded system, you need to check that your buffer does not use more memory than you have, and that you do not use memory addresses that are used for other purposes.

Please note that you are recommended to test, using pen and paper, what will happen to your circular buffer when you run the operations you will implement in your functions. This might help you understand how to write your code.

Function: void initCircularBuffer() Function declaration:

```
void initCircularBuffer(struct circularBuffer* bufferPtr, int* data, int maxLen);
```

This function is used to initalize the buffer using, e.g. a statically allocated array of integers.

This function is already implemented in the code startup code.

Function: int contains() Function declaration:

```
int contains(struct circularBuffer* bufferPtr, int value);
```

This function should check if the buffer pointed to by `bufferPtr` contains at least one element with the value specified by the `value` argument.

The function should return:

- `value` if an element with the input value was found in the buffer
- `INT_MIN` (defined in `limits.h`) if no element with the argument value was not found.

Function: int addElement() Function declaration:

```
int addElement(struct circularBuffer* bufferPtr, int value);
```

This function should add the value specified by the `value` argument at the tail of the buffer.

The function should return:

- `value` if the value was successfully added to the buffer.
- `INT_MIN` (defined in `limits.h`) if the value was not added.

Fuction: int removeValue() Function declaration:

```
int removeValue(struct circularBuffer* bufferPtr, int value);
```

This function should remove all elements in the buffer whose value matches the `value` argument.

The function should return:

- `value` if at least one element was removed from the queue.
- `INT_MIN` (defined in `limits.h`) if no element was removed.

Function: int removeHead() Function declaration:

```
int removeHead(struct circularBuffer* bufferPtr);
```

Remove the oldest element in the buffer, that is, the value at the head of the queue.

The function should return:

- The value of the head element if it was successfully removed.
- `INT_MIN` (defined in `limits.h`) if no element was removed (i.e., the queue was empty when the function was called.

Function: void printBuffer() Function declaration:

```
void printBuffer(struct circularBuffer* bufferPtr);
```

Print the content of the buffer in meaningful, easy-to-understand way. The function should print, at least, the following information in the terminal:

- All values currently stored in the buffer, starting from head.
- The position (i.e., array index) of the head pointer.
- The position (i.e., array index) of the tail pointer.

Test your circular buffer

As part of this assignment you should carefully test your circular buffer. In particular, you should write black box and white box tests according to the specification below. For each test, you should present your output in the visual studio code terminal (for example, using `printf`) in such a way that the output can be understood by the examiner of your assignment.

Black box tests You should write a function for each of the following test cases (for example in `main.c`):

1. Add one element using `addElement` and remove one element using `removeHead()`. Check that added and removed element is the same.
2. Add one element, add another element (with different value), remove one element, remove one element. Check that added and removed elements are in same order.
3. Add `BUFFER_SIZE` number of differeny elements in sequence, and remove `BUFFER_SIZE` elements. Check that added and removed elements are in same order.
4. Add `BUFFER_SIZE+1`elements. Check that function failed or that last element overwrites the first added element, depending on your choice of functionality.
5. Repeat test 1 `BUFFER_SIZE+1` times. Check that added and removed elements are in same order.
6. Run `contains()` on empty buffer. Check that it returns value `INT_MIN`.
7. Add one element. Check that buffer contains an element with the added value.
8. Add two elements. Check that the buffer contains the value of the second element you added.
9. Add `BUFFER_SIZE` elements with different values. Check that the buffer contains the value of the last element you added.

White box tests You should implement white box tests for the functions in `circular_buffer.h`.

For each of the white box test cases you should check that the buffer contains the expected values in the right places, and that values of the head and tail are correct.

initCircularBuffer() Check that the following is true after running `initCircularBuffer`:

- buffer.maxLength equals BUFFER_SIZE
- buffer.head equals 0
- buffer.tail equals 0

contains() The `contains()` function is covered in the black box tests specified above.

addElement()

- Add an element to an empty buffer, when the element should be added at the beginning of the buffer array.

- Add an element to an empty buffer, when the element should be added at some other place than the beginning of the buffer array.
- Add an element to a buffer already containing one element. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.
- Add an element to a buffer already containing `BUFFER_SIZE-1` elements. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.

removeValue()

- Remove elements with the argument value if buffer contains 1 element. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.

- Remove element with specified value if buffer is full.
  - Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.
  - Different placements of the elements (one or more) to remove might need to be considered, and you should figure out which.
- Try remove elements with specified value if buffer is empty.

removeHead()

- Remove element when buffer contains only 1 element. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.
- Remove element when buffer contains two elements. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.
- Remove element when buffer contains three elements. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.
- Try remove element when circular buffer is empty. Different placements of the tail and header might need to be considered, and you need to figure out which combinations are relevant.

Submission and examination

When you are done with the assignment, you should upload the `src` folder of your visual studio code project (in a zip file) using the submission page for assignment 2 on Canvas.

You should also demonstrate and discuss your solution with the examiner of assignment 2, which you do during of the scheduled presentation times for laboratory assignment 2 (examination times will be published on Canvas). During

the discussion, you should be able to show that your code works correctly, explain the code, and answer some basic questions about your solution.