Implementing Circular Buffer in C

May 2014 (/archives/2014/05) Siddharth (/authors/siddharth-chandrasekaran) Programming (/category/programming) Print

Theory (/tag/theory) , Algorithm (/tag/algorithm)

Embedded software often involves state machines, circular buffers and queues. This article will give you an overview of the data structure and walks you through the steps involved in implementing circular buffers in low memory devices. If you are already familiar with the basics of such data structures, feel free to jump into the implementation section from the below table of contents.

Table of Contents

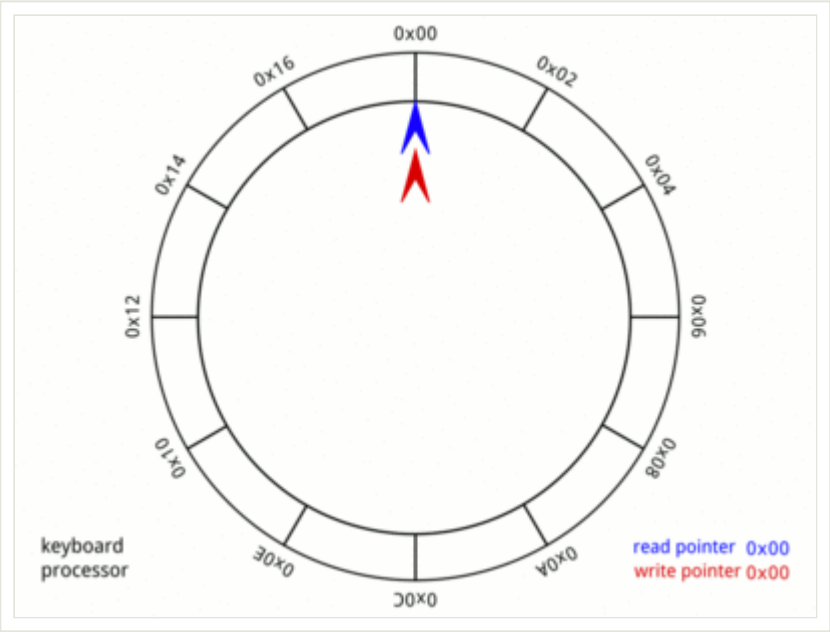
Theoretical Background

Choice of a good data structure or algorithm for a given problem comes after a deep understanding of the underlying theory. In this section we will go over some of the key aspects and problems of a circular buffer implementation. Hopefully, this will allow you to make informed decisions on the choice of data structure.

What is a circular buffer?

Circular buffer is a FIFO data structure that treats memory to be circular; that is, the read/write indices loop back to 0 after it reaches the buffer length. This is achieved by two pointers to the array, the "head" pointer and the "tail" pointer. As data is added (write) to the buffer, the head pointer is incremented and likewise, when the data is being removed (read) the tail pointer is incremented. The definition of head, tail, their movement direction and write and read location are all implementation dependent but the idea/goal remains the same.

So, for the sake of this discussion, we will consider, that a *write* is done at head and read at *tail*. Here is a nice GIF from Wikipedia (https://en.wikipedia.org/wiki/Circular_buffer):,



The picture says it all. The animation is very fast and may take some iterations before you notice all the cases involved. Do spend the time with this image as it gives a visual representation of the memory and pointers that will be used in later parts of this post.

Need for circular buffers

Circular buffers are excessively used to solve the single produce-consumer problem. That is, one thread of execution is responsible for data production and another for consumption. In the very low to medium embedded devices, the producer is often an Interrupt Service Routine (ISR) - perhaps data produced from sensors - and consumer is the main event loop.

But why does it have to be circular? why not a regular array? one might wonder. It is a very common question that pops out when you hear about circular buffers for the first time. Though the course of this article, the need for these data structures will be made very clear.

Synchronization and race conditions

I'm sure you have come across a race condition due to lack of synchronization in your programming career. Most people seem to think they don't apply to the low memory embedded world (which is in verge of extinction) where there is only one thread of execution. On the contrary, they exist there too (think of ISRs here), in fact, more predominantly there.

The nice thing about circular buffers is its elegance. The down side is its not easy to implement it without race conditions. Another good thing is that they can be implemented without the need for locks for a single producer and single consumer environment. It may not sound like much but, don't be too quick to judge, single producer-consumer needs account for a large portion overall circular buffer applications.

This makes it an ideal data structure for bare-metal embedded programs. The bottom line is that it *has* to be implemented correctly to be free of a race.

The full vs empty problem

The big pain point in circular buffers is that there is no *clean way* to differentiate the buffer full vs empty cases. This is because in both the cases, head is equal to tail. That is, initially head and tail pointed point to the same location; when data is filled into the buffer, head is incremented and eventually it wraps around (more on this later) and catches tail when you fill the N^{th} element into the N element buffer. At this point, head will point to the same location as tail but now the buffer is actually full, not empty.

There are a lot of ways/workarounds to deal with this but most of them introduce a lot of complexity and hinders readability. This article presents a method that gives importance to elegance in design.

In this method, we deliberately use only N-1 elements in the N element buffer. The last element is used (this of this more like a flag) to differentiate between empty and full cases. By this logic,

- if head is equal to tail -> the buffer is empty
- if (head + 1) is equal to tail -> the buffer is full

In essence, every push checks the `is_buffer_full` condition and every pop, checks the `is_buffer_empty`.

Overwrite or discard when full?

This is the last question that pops up regarding circular buffers. Whether new data has to be discarded or should it overwrite existing data when the buffer is full. The answer is, there is no clear advantage of one over the other, and most of the time its implementation/usage specific. If the most recent data makes more sense to your application, then go with the overwrite approach. On the other hand, if data has to processed on a first-come first-serve mode discard.

The following implementation will discard new data when the buffer is full.

Implementation in C

Now that we have dealt with the theory, let's proceed with the implementation by defining the data types and subsequently the core, push and pop methods.

In push and pop routines, we will compute the 'next' offset points to the location that the current write-to/read-from will happen. As discussed earlier, if the next location points to the location pointed by the tail then we know that the buffer is full, and we don't write data into the buffer (return an error). Similarly, when the head is equal to tail we know that buffer is empty and nothing can be read from it.

Data structure definition

```
typedef struct {
    uint8_t * const buffer;
    int head;
    int tail;
    const int maxlen;
} circ_bbuf_t;
```

There goes our primary structure to handle the buffer and its pointers. Notice that buffer is `uint8_t * const buffer`. `const uint8_t *` is a pointer to a byte array of constant elements, that is the value being pointed to can't be changed but the pointer itself can. On the other hand `uint8_t * const` is a constant pointer to an array of bytes in which the value being pointed to can changed but not the pointer.

This is done so that you will be able to make changes to the buffer but you will not be able to accidentally orphan the pointer. This is a very good safety measure that I strongly suggest you keep as-is.

Push data into the circular buffer

In majority of the use case scenarios, you will be calling this from within an ISR. Hence, a push should be as small and the whole routine should be enclosed within critical sections to make it synchronized in multi threaded environments.

Data has to be loaded before the head pointer is incremented to ensure that only valid data is read by the consumer thread (one which calls pop. See below).

Also, if you notice, we hold one byte as reserved space in the buffer. On first glance it may appear as an *off by one* but if you thing about it, you will realize its an engineering trade off. If I were to use that one extra byte, detection of full and empty cases becomes slightly complex and writing code that will handle all corner cases is time consuming and hard to debug if it comes to it.

So, in conclusion, for small and fixed size data units, just reserve one byte while you can still keep your sanity.

```
int circ_bbuf_push(circ_bbuf_t *c, uint8_t data)
{
    int next;

    next = c->head + 1;  // next is where head will point to after this write.
    if (next >= c->maxlen)
        next = 0;

    if (next == c->tail)  // if the head + 1 == tail, circular buffer is full
        return -1;

    c->buffer[c->head] = data;  // Load data and then move
    c->head = next;             // head to next data offset.
    return 0;  // return success to indicate successful push.
}
```

Pop data from the circular buffer

Pop routine is called by the application process to pull data off the buffer. This also has to be enclosed in critical sections if more than one threads are reading off this buffer (although that's not how it is usually done)

Here, the tail *can* be moved to the next offset before the data has been read since each data unit is one byte and we reserve one byte in the buffer when we are fully loaded. But in more advanced circular buffer implementations, data units does not *need* to be of the same size. In such cases, we try to save even the last byte adding more check and bounds.

In such implementations, if tail is moved before read, the data to be read can potentially be overwritten by a newly pushed data. So its a general best practice to the read data and then move the tail pointer.

```
int circ_bbuf_pop(circ_bbuf_t *c, uint8_t *data)
{
    int next;

    if (c->head == c->tail)  // if the head == tail, we don't have any data
        return -1;

    next = c->tail + 1;  // next is where tail will point to after this read.
    if(next >= c->maxlen)
        next = 0;

    *data = c->buffer[c->tail];  // Read data and then move
    c->tail = next;              // tail to next offset.
    return 0;  // return success to indicate successful push.
}
```

Typical use case

I think its pretty obvious that you have to define a buffer of a certain length and then create an instance of `circ_bbuf_t` and assign the pointer to buffer and its `maxlen`. It also goes without saying that the buffer has to be global or it has to be in stack so long as you need to use it.

To make maintenance a little easier, you could use this macro but compromises code readability for new users.

```
#define CIRC_BBUF_DEF(x,y)                \
    uint8_t x##_data_space[y];            \
    circ_bbuf_t x = {                     \
        .buffer = x##_data_space,         \
        .head = 0,                        \
        .tail = 0,                        \
        .maxlen = y                       \
    }
```

So for example if you need a circular buffer of length 32 bytes, you would do something like this in your application,

```
CIRC_BBUF_DEF(my_circ_buf, 32);

int your_application()
{
    uint8_t out_data=0, in_data = 0x55;

    if (circ_bbuf_push(&my_circ_buf, in_data)) {
        printf("Out of space in CB\n");
        return -1;
    }

    if (circ_bbuf_pop(&my_circ_buf, &out_data)) {
        printf("CB is empty\n");
        return -1;
    }

    // here in_data = in_data = 0x55;
    printf("Push: 0x%x\n", in_data);
    printf("Pop:  0x%x\n", out_data);
    return 0;
}
```

You can find a complete implementation of the above at EmbedJournal/c-utils (https://github.com/EmbedJournal/c-utils) in files circular-byte-buffer.c (https://github.com/EmbedJournal/c-utils/blob/master/circular-byte-buffer.c) and circular-byte-buffer.h (https://github.com/EmbedJournal/c-utils/blob/master/circular-byte-buffer.h).

I hope this post was of some help in understanding circular buffers. We will see more such data structures and an advanced extensions to this circular buffer that allows push/pop of non just bytes but any data type (even user defined data types) with type checking and other niceness in a future post.

```
Edit History:
04 Aug 2018 - Reworded to clarify need for unused byte in buffer
05 Aug 2018 - Refactor code; Added reference to EmbedJournal/c-utils.git
01 Feb 2019 - Modify post heading; some rewording/restructuring
```

Siddharth Chandrasekaran

(/authors/siddharth-chandrasekaran)

Siddharth is the founder and editor of embedjournal.com. He is a Firmware Engineer, techie, and a movie-buff. His interests include, Programming, Embedded Systems, Linux, Robotics, CV, Carpentry and a lot more. You get to know him on the following social channels.

Read more about Siddharth (/authors/siddharth-chandrasekaran)