WELCOME (/FIRST-TIME-HERE)
ABOUT (/ABOUT)
BLOG (/BLOG)
ARCHIVE (/ARCHIVE)
RESOURCES
NEWSLETTER (/NEWSLETTER)
FOR BEGINNERS (/BEGINNERS)
GLOSSARY (/GLOSSARY)
OPEN-SOURCE LIBRARIES (/LIBRARIES)
TEMPLATES (/TEMPLATES)
SOFTWARE REFERENCES (/SOFTWARE-REFERENCES)
HARDWARE REFERENCES (/HARDWARE-REFERENCES)
DEVELOPMENT KITS (/DEV-KITS)
TECHNOLOGY RADAR (HTTPS://RADAR.THOUGHTWORKS.COM/?SHEETID=HTTPS%3A%2F%2FDOCS.GOOGLE.COM%2FSPREADSHEETS%2FD%2F1GZ82TDORTWFWRYDT3SJWKRFJF-43JMXI96-F3DT-RYE%2FEDIT%23GID%3D0)
YOUTUBE CHANNEL (HTTPS://WWW.YOUTUBE.COM/CHANNEL/UCCWU5DOSCQIUAVQTBISHSSQ)
SEARCH (/SEARCH-EA)
STORE (/STORE)
CONSULTING (HTTPS://EMBEDDEDARTISTRYCONSULTING.COM)

May 17, 2017 (/blog/2017/4/6/circular-buffers-in-cc)

Creating a Circular Buffer in C and C++ (/blog/2017/4/6/circular-buffers-in-cc)

Phillip Johnston (/blog?author=5771d1832994ca52eb5a9f35)

Updated: 2019-06-04

Due to the resource constrained nature of embedded systems, circular buffer data structures (https://en.wikipedia.org/wiki/Circular_buffer) can be found in most projects.

Circular buffers (also known as ring buffers) are fixed-size buffers that work as if the memory is contiguous & circular in nature. As memory is generated and consumed, data does not need to be reshuffled - rather, the head/tail pointers are adjusted. When data is added, the head pointer advances. When data is consumed, the tail pointer advances. If you reach the end of the buffer, the pointers simply wrap around to the beginning.

For a more detailed summary of circular buffer operation, please refer to the Wikipedia article (https://en.wikipedia.org/wiki/Circular_buffer). The rest of the article assumes you have an understanding of how circular buffers work.

Table of Contents:

Search

NEWSLETTER (/NEWSLETTER)

Want to stay up-to-date with the embedded systems world? Sign up for our newsletter (/newsletter).

SUPPORT THE WEBSITE (/SUPPORT-THE-WEBSITE)

We spend 80+ hours each month researching & writing.

Your support enables us to create quality content for the community:

DONATE NOW

FEATURED POSTS (/ARCHIVE)

PROGRAMMERS: LET'S STUDY SOURCE CODE

A GENERAL OVERVIEW OF WHAT HAPPENS BEFORE MAIN() (/BLOG/2019/4/8/A-

Why Use A Circular Buffer?

Circular buffers are often used as fixed-sized queues. The fixed size is beneficial for embedded systems, as developers often try to use static data storage methods rather than dynamic allocations.

Circular buffers are also useful structures for situations where data production and consumption happen at different rates: the most recent data is always available. If the consumer cannot keep up with production, the stale data will be overwritten with more recent data. By using a circular buffer, we can ensure that we are always consuming the most recent data.

For additional use cases, check out Ring Buffer Basics (http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer) on Embedded.com (http://embedded.com).

C Implementation

We will start with a C implementation, as this exposes us to some of the design challenges and tradeoffs when creating a circular buffer library.

USING ENCAPSULATION

Since we are creating a circular buffer library, we want to make sure users work with our library APIs instead of modifying the structure directly. We also want to keep the implementation contained within our library so we can change it as needed, without requiring end users to update their code. The user doesn't need to know any details about our structure, only that it exists.

In of our library header, we will forward declare the structure:

```
// Opaque circular buffer structure
typedef struct circular_buf_t circular_buf_t;
```

We don't want users to work with a `circular_but_t` pointer directly, as they might get the impression that they can dereference the value. We will create a handle type that they can use instead.

The simplest approach for our handle is to `typedef` the `cbuf_handle_t` as a pointer to the circular buffer. This will prevent us from needing to cast the pointer within our function implementation.

```
// Handle type, the way users interact with the API
typedef circular_buf_t* cbuf_handle_t;
```

An alternative approach would be to make the handle a `uintptr_t` or `void*` value. Inside of

WHAT CAN SOFTWARE ORGANIZATIONS LEARN FROM THE BOEING 737 MAX

HYPOTHESES ON SYSTEMS AND COMPLEXITY

EMBEDDED TEMPLATE LIBRARY

CATEGORIES (/ARCHIVE)

Bringup (13) (/blog/category/Bringup)
Build Automation (6) (/blog/category/Build+Automation)
Debugging (2) (/blog/category/Debugging)
Development Kits (7) (/blog/category/Development+Kits)
Embeddable Libraries (20) (/blog/category/Embeddable+Libraries)
Embedded C++ Series (14) (/blog/category/Embedded+C%2B%2B+Se
Embedded Clang (3) (/blog/category/Embedded+Clang)
Framework (3) (/blog/category/Framework)
Interviews (1) (/blog/category/Interviews)
Manufacturing (4) (/blog/category/Manufacturing)
Migrating from C to C++ (29) (/blog/category/Migrating+from+C+to+C%2
Monday Morning Reading (24) (/blog/category/Monday+Morning+Readin
Product Development (12) (/blog/category/Product+Development)
Quotes (1) (/blog/category/Quotes)
Templates (12) (/blog/category/Templates)

our interface, we would handle the translation to the appropriate pointer type. We keep the circular buffer type hidden from users, and the only way to interact with the data is through the handle.

We're going to stick with the simple handle implementation to keep our example code simple and straightforward.

API DESIGN

First, we should think about how users will interact with a circular buffer:

- They need to initialize the circular buffer container with a buffer and size
- They need to destroy a circular buffer container
- They need to reset the circular buffer container
- They need to be able to add data to the buffer
- They need to be able to get the next value from the buffer
- They need to know whether the buffer is full or empty
- They need to know the current number of elements in the buffer
- They need to know the max capacity of the buffer

Using this list, we can put together an API for our library. Users will interact with the circular buffer library using our opaque handle type, which is created during initialization.

I have chosen `uint8_t` as the underlying data type in this implementation. You can use any particular type that you like - just be careful to handle the underlying buffer and number of bytes appropriately.

```
/// Pass in a storage buffer and size
/// Returns a circular buffer handle
cbuf_handle_t circular_buf_init(uint8_t* buffer, size_t size);

/// Free a circular buffer structure.
/// Does not free data buffer; owner is responsible for that
void circular_buf_free(cbuf_handle_t cbuf);

/// Reset the circular buffer to empty, head == tail
void circular_buf_reset(cbuf_handle_t cbuf);

/// Put version 1 continues to add data if the buffer is full
/// Old data is overwritten
void circular_buf_put(cbuf_handle_t cbuf, uint8_t data);

/// Put Version 2 rejects new data if the buffer is full
/// Returns 0 on success, -1 if buffer is full
int circular_buf_put2(cbuf_handle_t cbuf, uint8_t data);

/// Retrieve a value from the buffer
/// Returns 0 on success, -1 if the buffer is empty
int circular_buf_get(cbuf_handle_t cbuf, uint8_t * data);

/// Returns true if the buffer is empty
bool circular_buf_empty(cbuf_handle_t cbuf);

/// Returns true if the buffer is full
```

```
bool circular_buf_full(cbuf_handle_t cbuf);

/// Returns the maximum capacity of the buffer
size_t circular_buf_capacity(cbuf_handle_t cbuf);

/// Returns the current number of elements in the buffer
size_t circular_buf_size(cbuf_handle_t cbuf);
```

DETERMINING IF A BUFFER IS FULL

Before we proceed, we should take a moment to discuss the method we will use to determine whether or buffer is full or empty.

Both the "full" and "empty" cases of the circular buffer look the same: `head` and `tail` pointer are equal. There are two approaches to differentiating between full and empty:

1. Waste a slot in the buffer:
    - Full state is `tail + 1 == head`
    - Empty state is `head == tail`
2. Use a `bool` flag and additional logic to differentiate states::
    - Full state is `full`
    - Empty state is `(head == tail) && !full`

Rather than waste a potentially valuable data slot, the implementation below uses the `bool` flag. Using the flag requires additional logic in the `get` and `put` routines to update the flag. We are comfortable with that tradeoff.

CIRCULAR BUFFER CONTAINER TYPE

Now that we have a grasp on the operations we'll need to support, we can design our circular buffer container.

We use the container structure for managing the state of the buffer. To preserve encapsulation, the container structure is defined inside of our library `.c` file, rather than in the header.

We will need to keep track of:

- The underlying data buffer
- The maximum size of the buffer
- The current "head" position (incremented when elements are added)
- The current "tail" (incremented when elements are removed)
- A flag indicating whether the buffer is full or not

```
// The hidden definition of our circular buffer structure
struct circular_buf_t {
    uint8_t * buffer;
    size_t head;
    size_t tail;
    size_t max; //of the buffer
    bool full;
};
```

Now that our container is designed, we are ready to implement the library functions.

IMPLEMENTATION

One important detail to note is that each of our APIs requires an initialized buffer handle. Rather than litter our code with conditional statements, we will utilize assertions to enforce our API requirements in the "Design by Contract (https://en.wikipedia.org/wiki/Design_by_contract)" style.

If the interfaces are improperly used, the program will fail immediately rather than requiring the user to check and handle the error code.

For example:

```
circular_buf_reset(NULL);
```

Produces:

```
=== C Circular Buffer Check ===
Assertion failed: (cbuf), function circular_buf_reset, file
../../circular_buffer.c, line 35.
Abort trap: 6
```

Another important note is that the implementation shown below is not thread-safe. No locks have been added to the underlying circular buffer library.

INITIALIZE AND RESET

Let's start at the beginning: initializing a circular buffer. Our API has clients provide the underlying buffer and buffer size, and we return a circular buffer handle to them.

We are required to create the circular buffer container on the library side. I have used `malloc` for simplicity. Systems which cannot use dynamic memory simply need to modify the `init` function to use a different method, such as allocation from a static pool of circular buffer containers.

Another approach would be to break encapsulation, allowing users to statically declare circular buffer container structures. In this case, `circular_buf_init` needs to be updated to take a struct pointer, or `init` can create a container structure on the stack and return it. However, since encapsulation is broken, users will be able to modify the structure without using the library routines.

```
// User provides struct
void circular_buf_init(circular_buf_t* cbuf, uint8_t* buffer,
    size_t size);

// Return a struct
circular_buf_t circular_buf_init(uint8_t* buffer, size_t size)
```

Once we've created our container, we need populate the values and call `reset` on it. Before we return from `init`, we ensure that the buffer container has been created in an empty state.

```
cbuf_handle_t circular_buf_init(uint8_t* buffer, size_t size)
{
    assert(buffer && size);
    cbuf_handle_t cbuf = malloc(sizeof(circular_buf_t));
    assert(cbuf);

    cbuf->buffer = buffer;
    cbuf->max = size;
    circular_buf_reset(cbuf);

    assert(circular_buf_empty(cbuf));

    return cbuf;
}
```

The purpose of the reset function is to put the buffer into an "empty" state, which requires updating `head`, `tail`, and `full`:

```
void circular_buf_reset(cbuf_handle_t cbuf)
{
    assert(cbuf);

    cbuf->head = 0;
    cbuf->tail = 0;
    cbuf->full = false;
}
```

Since we have a method to create a circular buffer container, we need an equivalent method for destroying the container. In this case, we call `free` on our container. We do not attempt to free the underlying buffer, since we do not own it.

```
void circular_buf_free(cbuf_handle_t cbuf)
{
    assert(cbuf);
    free(cbuf);
}
```

STATE CHECKS

Next, we'll implement the functions related to the state of the buffer container.

The full function is the easiest to implement, since we have a flag representing the state:

```
bool circular_buf_full(cbuf_handle_t cbuf)
{
    assert(cbuf);

    return cbuf->full;
}
```

Since we have the `full` flag to differentiate between full or empty state, we combine the flag with a check that `head == tail`:

```
bool circular_buf_empty(cbuf_handle_t cbuf)
{
    assert(cbuf);

    return (!cbuf->full && (cbuf->head == cbuf->tail));
}
```

The capacity of our buffer was supplied during initialization, so we just return that value to the user:

```
size_t circular_buf_capacity(cbuf_handle_t cbuf)
{
    assert(cbuf);

    return cbuf->max;
}
```

Calculating the number of elements in the buffer was a trickier problem than I expected. Many proposed size calculations use modulo, but I ran into strange corner cases when testing that out. I opted for a simplified calculation using conditional statements.

If the buffer is full, we know that our capacity is at the maximum. If `head` is greater-than-or-equal-to the `tail`, we simply subtract the two values to get our size. If `tail` is greater than `head`, we need to offset the difference with `max` to get the correct size.

```
size_t circular_buf_size(cbuf_handle_t cbuf)
{
    assert(cbuf);

    size_t size = cbuf->max;

    if(!cbuf->full)
    {
        if(cbuf->head >= cbuf->tail)
        {
            size = (cbuf->head - cbuf->tail);
        }
        else
        {
            size = (cbuf->max + cbuf->head - cbuf->tail);
        }
    }

    return size;
}
```

ADDING AND REMOVING DATA

With the bookkeeping functions out of the way, it's time to dig into the meat: adding and removing data from the queue.

Adding and removing data from a circular buffer requires manipulation of the `head` and `tail` pointers. When adding data to the buffer, we insert the new value at the current `head` location, then we advance `head`. When we remove data from the buffer, we retrieve the value of the current `tail` pointer and then advance `tail`.

Adding data to the buffer requires a bit more thought, however. If the buffer is `full`, we need to advance our `tail` pointer as well as `head`. We also need to check whether inserting a value triggers the `full` condition.

We are going to implement two versions of the `put` function, so let's extract our pointer advancement logic into a helper function. If our buffer is already full, we advance `tail`. We always advance `head` by one. After the pointer has been advanced, we populate the `full` flag by checking whether `head == tail`.

Note the use of the modulo operator (%) below. Modulo will cause the `head` and `tail` values to reset to 0 when the maximum size is reached. This ensures that `head` and `tail` are always valid indices of the underlying data buffer.

```
static void advance_pointer(cbuf_handle_t cbuf)
{
    assert(cbuf);

    if(cbuf->full)
       {
        cbuf->tail = (cbuf->tail + 1) % cbuf->max;
    }

    cbuf->head = (cbuf->head + 1) % cbuf->max;
    cbuf->full = (cbuf->head == cbuf->tail);
}
```

We can make a similar helper function which is called when removing a value from the buffer. When we remove a value, the `full` flag is set to `false`, and the tail pointer is advanced.

```
static void retreat_pointer(cbuf_handle_t cbuf)
{
    assert(cbuf);

    cbuf->full = false;
    cbuf->tail = (cbuf->tail + 1) % cbuf->max;
}
```

We'll create two versions of the `put` function. The first version inserts a value into the buffer and advances the pointer. If the buffer is full, the oldest value will be overwritten. This is the standard use case for a circular buffer

```
void circular_buf_put(cbuf_handle_t cbuf, uint8_t data)
{
    assert(cbuf && cbuf->buffer);

    cbuf->buffer[cbuf->head] = data;

    advance_pointer(cbuf);
}
```

The second version of the `put` function returns an error if the buffer is full. This is provided for demonstration purposes, but we do not use this variant in our systems.

```
int circular_buf_put2(cbuf_handle_t cbuf, uint8_t data)
{
    int r = -1;

    assert(cbuf && cbuf->buffer);

    if(!circular_buf_full(cbuf))
    {
        cbuf->buffer[cbuf->head] = data;
        advance_pointer(cbuf);
        r = 0;
    }

    return r;
}
```

To remove data from the buffer, we access the value at the `tail` and then update the `tail` pointer. If the buffer is empty we do not return a value or modify the pointer. Instead, we return an error to the user.

```
int circular_buf_get(cbuf_handle_t cbuf, uint8_t * data)
{
    assert(cbuf && data && cbuf->buffer);

    int r = -1;

    if(!circular_buf_empty(cbuf))
    {
        *data = cbuf->buffer[cbuf->tail];
        retreat_pointer(cbuf);

        r = 0;
    }

    return r;
}
```

That completes the implementation of our circular buffer library.

USAGE

When using the library, the client is responsible for creating the underlying data buffer to `circular_buf_init`, and a `cbuf_handle_t` is returned:

```
uint8_t * buffer  = malloc(EXAMPLE_BUFFER_SIZE * sizeof(uint8_t));
cbuf_handle_t cbuf = circular_buf_init(buffer,
    EXAMPLE_BUFFER_SIZE);
```

This handle is used to interact with all remaining library functions:

```
bool full = circular_buf_full(cbuf);
bool empty = circular_buf_empty(cbuf);
printf("Current buffer size: %zu\n", circular_buf_size(cbuf);
```

Don't forget to free both the underlying data buffer and the container when you are done:

```
free(buffer);
circular_buf_free(cbuf);
```

A test program (https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer_test.c) which uses the circular buffer library (https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer/circular_buffer.c) can be found in the embedded-resources repository (https://github.com/embeddedartistry/embedded-resources/).

C++

C++ lends itself to a cleaner circular buffer implementation than C.

CLASS DEFINITION

We'll start off by defining our C++ class. We want our C++ implementation to support any type of data, so we are going to make it a templated class.

Our APIs are going to be similar to the C implementation. Our class will provide interfaces for:

- Resetting the buffer to empty
- Adding data
- Removing data
- Checking full/empty state
- Checking the current number of elements in the buffer
- Checking the total capacity of the buffer

We will also utilize C++ smart pointers (https://embeddedartistry.com/blog/2016/9/19/rfr0r76r0ovd0gk574kfsldxfbklgs) to ensure sure we don't leave any data around once our buffer is destroyed. This means we can manage the buffer for the user.

Another benefit of C++ is the triviality of making this class thread-safe: we can rely on the `std::mutex` type (assuming this is defined for your platform).

Here's our class definition:

```
template <class T>
class circular_buffer {
public:
    explicit circular_buffer(size_t size) :
        buf_(std::unique_ptr<T[]>(new T[size])),
        max_size_(size)
    { // empty }

    void put(T item);
    T get();
    void reset();
    bool empty() const;
    bool full() const;
    size_t capacity() const;
    size_t size() const;

private:
    std::mutex mutex_;
    std::unique_ptr<T[]> buf_;
    size_t head_ = 0;
    size_t tail_ = 0;
    const size_t max_size_;
    bool full_ = 0;
};
```

C++ IMPLEMENTATION

Our C++ circular buffer mimics much of the logic from the C implementation, but results in a much cleaner and more reusable design. Also, the C++ buffer utilizes `std::mutex` to provide a thread-safe implementation.

INITIALIZATION

When constructing our class, we allocate the data for our underlying buffer and set the buffer size. This removes the overhead required with the C implementation.

Unlike the C implementation, the C++ constructor does not call `reset`. Because we specify initial values for our member variables, our circular buffer starts out in the correct state.

```
explicit circular_buffer(size_t size) :
    buf_(std::unique_ptr<T[]>(new T[size])),
    max_size_(size)
{
    //empty constructor
}
```

Our reset behavior puts the buffer back to an empty state (`head == tail && !full_`).

```
void reset()
{
    std::lock_guard<std::mutex> lock(mutex_);
    head_ = tail_;
    full_ = false;
}
```

STATE TRACKING

The logic of the `empty` and `full` cases is the same as the C example:

```
bool empty() const
{
    //if head and tail are equal, we are empty
    return (!full_ && (head_ == tail_));
}
```

```
bool full() const
{
    //If tail is ahead the head by 1, we are full
    return full_;
}
```

In the C++ circular buffer implementation, `size` and `capacity` report the number of elements in the queue rather than the size in bytes. This allows us to be agnostic to the underlying details of the type.

```
size_t capacity() const
{
    return max_size_;
}

size_t size() const
{
    size_t size = max_size_;

    if(!full_)
    {
        if(head_ >= tail_)
        {
            size = head_ - tail_;
        }
        else
        {
            size = max_size_ + head_ - tail_;
        }
    }

    return size;
}
```

ADDING DATA

The logic for `put` matches the C implementation. This implementation uses the "overwrite the oldest value" behavioral pattern.

```
void put(T item)
{
    std::lock_guard<std::mutex> lock(mutex_);

    buf_[head_] = item;

    if(full_)
    {
        tail_ = (tail_ + 1) % max_size_;
    }

    head_ = (head_ + 1) % max_size_;
    full_ = head_ == tail_;
}
```

RETRIEVING DATA

The logic behind `get` matches the C implementation. Unlike the C implementation, an empty value is returned if the buffer is empty.

```
T get()
{
    std::lock_guard<std::mutex> lock(mutex_);

    if(empty())
    {
        return T();
    }

    //Read data and advance the tail (we now have a free space)
    auto val = buf_[tail_];
    full_ = false;
    tail_ = (tail_ + 1) % max_size_;

    return val;
}
```

USAGE

The C++ circular buffer is much simpler to use than the C implementation.

To instantiate a circular buffer, we just declare an object and specify the templated type for our buffer. Here's an example using a buffer of 10 `uint32_t` entries:

```
circular_buffer<uint32_t> circle(10);
```

Adding data is easy:

```
uint32_t x = 100;
circle.put(x);
```

And getting data is equally easy:

```
x = circle.get()
```

Remember that since this is a templated class, you can create a circular buffer of any type that you need.

Putting it All Together

Example implementations can be found in the `embedded-resources` Github repository (https://github.com/embeddedartistry/embedded-resources/).

- C circular buffer example (https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer_test.c)

    - C circular buffer library (https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer/circular_buffer.c)
- C++ circular buffer example (https://github.com/embeddedartistry/embedded-resources/blob/master/examples/cpp/circular_buffer.cpp)

If you are looking to extend this library, a useful exercise is to add additional APIs to enable users to add/remove multiple elements with a single operation. You can also make the C implementation thread-safe.

THREAD SAFETY WITH THE LOOKAHEAD METHOD

One approach for thread-safety without a mutex is the "lookahead" method. This method supports a single producer thread and single consumer thread; multiple producers or consumers will require a lock.

Instead of using the boolean flag to differentiate between the full and empty cases, we will always leave one cell empty. By using a single empty cell to detect the "full" case, we can support a single producer and single consumer without a lock (as long as `put` and `get` don't modify the same variables).

You may be concerned about wasting a slot, but this tradeoff is often much cheaper than the cost of using an OS lock primitive.

Further Reading

- C++ Smart Pointers (https://embeddedartistry.com/blog/2016/9/19/rfr0r76r0ovd0gk574kfsldxfbklgs)
- Ditch Your C-Stye Pointers for Smart Pointers (https://embeddedartistry.com/blog/2017/7/12/ditch-your-c-style-pointers-for-smart-pointers)

For more information on circular buffers:

- Embedded.com: Ring buffer basics (http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer)
- Wikipedia: Circular buffer (https://en.wikipedia.org/wiki/Circular_buffer)
- Boost Circular Buffer (https://theboostcpplibraries.com/boost.circularbuffer)
- C++: Performance of a Circular Buffer vs Vector, Deque, and List (https://www.codeproject.com/Articles/1185449/Performance-of-a-Circular-Buffer-vs-Vector-Deque-a)

There is a proposal for adding a circular buffer type to the C++ standard library:

- P0059: A Proposal to Add Ring Span to the Standard Library (http://open-std.org/JTC1/SC22/WG21/docs/papers/2017/p0059r3.pdf)
    - Ring Span (https://github.com/Quuxplusone/ring_view)
    - Ring Span Lite (https://github.com/martinmoene/ring-span-lite)

Change Log

2019-06-04

Fixed a typo (thanks Chris Svec!) and changed some wording related to the opaque type.

2018-12-19

Added note about avoiding concurrency problems with a single producer and single consumer using an empty slot.

2018-08-04

The article was restructured and rewritten.Thanks to everyone who provided feedback along the way. The examples have been updated to:

- Remove defensive programming
- Use assertions
- Create a standalone library using an opaque structure
- Expand the APIs, including a calculation for the current circular buffer size
- Update the library so it didn't waste a slot

Related Posts

IMPLEMENTING MALLOC WITH FREERTOS (/BLOG/2018/1/15/IMPLEMENT MALLOC-WITH-FREERTOS)
FreeRTOS (/blog?tag=FreeRTOS), C (/blog?tag=C), Libc (/blog?tag=Libc), Featured (/blog?tag=Featured), Memory (/blog?tag=Memory) · Bringup (/blog?category=Bringup)

IMPLEMENTING LIBC: STDLIB, PT. 2 (/BLOG/2017/10/18/IMPLEMEN LIBC-STDLIB-PT-2)
C (/blog?tag=C), libc (/blog?tag=libc) · Bringup (/blog?category=Bringup)

LIBC: STDLIB, PT. 1 (/BLOG/2017/4/10/LIBC-STDLIB)
C (/blog?tag=C), libc (/blog?tag=libc) · Bringup (/blog?category=Bringup)

Tagged: Library (/blog/tag/Library), C (/blog/tag/C), C++ (/blog/tag/C%2B%2B), Featured (/blog/tag/Featured)

8 Likes   Share

COMMENTS (39)   Oldest First   Subscribe via e-mail

Preview   POST COMMENT…

Murali 2 years ago · 0 Likes

Thanks for nice explanation. One thing to correct
1. In circular_buf_get (C)
if(cbuf && data && !circular_buf_empty(*cbuf))
should be
if(cbuf && data && !circular_buf_empty(cbuf))

Phillip Johnston (http://www.about.me/phillip.johnston)
(http://www.about.me/phillip.johnston) 2 years ago · 0 Likes

cbuf in that context is a pointer to a circular buffer, and the circular_buf_empty API does not take a pointer, so we need to dereference.

Reflecting on this with the benefit of time, I do think that a better API choice for embedded systems would be to have the circular_buf_empty() function take a pointer to a structure, which would allow you to skip the dereference.

Phillip Johnston
(http://www.about.me/phillip.johnston) (http://www.about.me/phillip.johnston) 2 years ago · 0 Likes

That would also apply to circular_buf_full()

Rahman 2 years ago · 0 Likes

I have one doubt .. How to retrieve the data in main function…?
can you explain with example?

Phillip Johnston (http://www.about.me/phillip.johnston)
(http://www.about.me/phillip.johnston) 2 years ago · 0 Likes

There is a functional example in the embedded-resources repository using a circular buffer built for uint8_t values:
https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer.c

In short, values are added to the queue with the `circular_buf_put()` API.

```
circular_buf_put(&amp;cbuf, i);
```

Values can be retrieved by the `circular_buffer_get()` API:

```
uint8_t data; circular_buf_get(&amp;cbuf, &amp;data);
```

Values will be retrieved based on the next value in line in the queue, until the current values in the queue are depleted.

Ryan Hill A year ago · 0 Likes

Hi, thank you very much for this tutorial. I've been trying for so long to implement a

Hi, thank you very much for this tutorial, I've been trying for so long to implement a circular buffer in C and this helped a lot! However, as I'm new to C programming I'm struggling with the next bit, which I feel should be the easiest part. I'm trying to fill the buffer with sensor readings and take an average of them, creating a moving average filter where the average updates with each new data point inputted into the buffer. Is this possible with this model or am I just missing something? I currently have the buffer filling with 9 of the same sensor readings, it doesn't change, and then it resets the buffer and refills, this isn't right for a moving average filter. Has anyone got any recommendations on where i'm going wrong?

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) A year ago · 0 Likes

Hard to say without example source code. Have you referenced the example circular buffer code in the embedded-resources project to see API usage?

https://github.com/embeddedartistry/embedded-resources/blob/master/examples/c/circular_buffer.c

Ryan Hill A year ago · 0 Likes

I've implemented the code on your GitHub and it performs the correct function. Where your code inputs a new value with each increment of 'i', I want to add a new sensor reading. Then when the buffer is full, i want to add a new value to the end of it and take an average value of the buffer each time a value is added.

Gustavo Velasco-Hernandez A year ago · 0 Likes

Great post! A concise ring buffer explanation for both C and C++. In the put() function/method you assume overwrite as default instead of discarding data. Is there any reason for that behavior or should the reader implement the behavior that fits the requirements?
Thanks.

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) A year ago · 1 Like

Hey Gustavo,

That's a great question, and something I should have touched on in the original article. Ultimately, the desired behavior should be selected for the application use case.

The reason I opted for overwriting data if the buffer is full is that I've primarily

The reason I opted for overwriting data if the buffer is full is that I've primarily worked with teams who prioritize new data over old data. If the buffer is full, they would rather lose the oldest (now stale) item rather than the most recent.

In either case, I think that a full circular buffer likely indicates some other problem on the system, as your consumer is not able to keep up with the data being produced. Perhaps there is a thread prioritization issue, the buffer isn't large enough, an interrupt storm is blocking the primary threads from running, etc.

welsh_t A year ago · 0 Likes

Any ideas for how to go about building circular buffers to store 12 bit samples from an adc in c? Can I just replace uint8_t with uint16_t and waste 2 bits per sample?

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) A year ago · 0 Likes

Yes, that is what I would do.

Jesus A year ago · 0 Likes

Hello Phillip, thank you for the great tutorial. When using a circular buffer to have a more efficient FIFO, would the following still make sense?

```
if(cbuf->head == cbuf->tail)
{
cbuf->tail = (cbuf->tail + 1) % cbuf->size;
}
```

It seems, I would be skipping retrieving one sample of data whenever the rate of retrieving data is slower than of inserting data. What work around would you recommend? I was thinking of having a bufferFull flag as well

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) A year ago · 0 Likes

A buffer full flag is the easiest way to prevent wasting a sample. In retrospect, I should have just done that from the start. I'll update this article in the coming weeks to reflect that.

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 11 months ago · 0 Likes

[illegible]

Hi Jesus, the article has been rewritten with support for the full flag.

Steve A year ago · 0 Likes

Hi Phillip
NIce clear explanation - thanks
I wonder if there is a problem though in your C++ example on github.
On line 35 you define:
T get(void) const
however, line 46:
tail_ = (tail_ + 1) % size_
then attempts to modify a member variable (tail_)

My compiler complains that:
... error: cannot assign to non-static data member within const member function 'get'
which makes sense.

Is this a mistake in the github code? - the const keyword does not appear in the function declaration in your code in the blog.

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) A year ago · 0 Likes

Hi Steve,

Thanks for writing. That is an error in the code - I got const happy :). Interestingly my compiler doesn't complain, which worries me. What compiler version are you using so I can dig into it?

Diego (http://meigahack.com) 11 months ago · 0 Likes

[illegible]

Hello,

Firstly, thanks for the nice explanation. I am creating my own circular library using yours as reference. I will share it with you.

However, I think you should not call "circular_buf_empty" function using "cbuf" as parameter when you did not know if it exists.

In my opinion, "if(cbuf && data && !circular_buf_empty(*cbuf))" should be like:

if(cbuf && data)
{
if (!circular_buf_empty(*cbuf))
{
/* Instructions */

}

Thanks!
Diego

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 11 months ago · 0 Likes

Hi Diego,

You are right about the problem with the code. I'm working on a revision for this article, as it's currently out of date with the example code.

The example code uses pointers, and it checks that cbuf is not null before using the values.

One thing to note is that in this statement:

```
if(cbuf &amp;&amp; data &amp;&amp; !circular_buf_empty(*cbuf))
```

Empty will not actually execute unless cbuf is valid (due to the && operator). Not that it makes the API any better :)

You should see an updated article next week.

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 11 months ago · 0 Likes

As promised, the post has been completely updated!

Marcin 8 months ago · 0 Likes

Hi Phillip,

thank you for taking your time and creating a nice tutorial. I wanted to ask you, why not to use std::deque as an underlying container for circular buffer?

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 8 months ago · 0 Likes

The ring buffer is useful when you have a pre-allocated buffer of a fixed size that you want to use for dynamically adding/removing objects. I used a std::unique_ptr in the example, but a C-style array or std::array could also be used (and are more common in my implementations).

Other containers, such as std::vector or std::deque, utilize dynamic memory

Other containers, such as std::vector or std::deque, utilize dynamic memory allocation whenever a new element is added (unless you reserve the buffer size up front). If you used one of these containers, you would not need to use the ring buffer structure at all.

ali (http://null) 6 months ago · 0 Likes

What does it mean

tail_ = (tail_ + 1) % size;

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 6 months ago · 0 Likes

1. Increment the tail (tail + 1)
2. Perform the modulo operation to cause tail_ to wrap back around to 0 if we reach size

Modulo is: Divide by the size, and take the remainder.

Muhammet 5 months ago · 0 Likes

Thanks for the post, one question:
in size(), why do you return max_size_ if the buffer is full (i.e. if full_ == true) ?

regards

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 4 months ago · 0 Likes

This is a holdover from a different implementation. I think it is no longer needed.

dirk lüsebrink (https://github.com/crux) 5 months ago · 0 Likes

I know you mentioned the modulo option in your size implementation. I think you can (I can) solve all corner cases like this:

circular_buf_size(cbuf_handle_t cbuf) {
if(full) { return max-capacity }

return (head + max-capacity - tail) % max-capacity
}

works for me and has no conditionals. But that is more a question of personal style I works for me and has no conditionals. But that is more a question of personal style I guess.

Nonetheless, thanx and great write-up, helped me to rinse out the index calculation errors I had in my javascript implementation of this,

}

Phillip Johnston (http://www.about.me/phillip.johnston)

4 months ago · 0 Likes
(http://www.about.me/phillip.johnston)

I remember trying that at one point and running into a problem. Not sure what was going on there, but I agree your version is superior. I'll work it in during the next update.

Dmitry 4 months ago · 0 Likes

Hi,

Thank you.
Just want to point out that this comment “//If tail is ahead the head by 1, we are full” is not correct. Did I misunderstand it? If tail is ahead the head that means that the buffer was full before the ongoing pop operation, and it will become empty when tail will == head.

Phillip Johnston (http://www.about.me/phillip.johnston)

3 months ago · 0 Likes
(http://www.about.me/phillip.johnston)

Hi Dmitry,

I will rework the code and article to remove the comment. That comes from an old implementation which kept an empty value in the queue (which made it thread-safe, but people requested the current implementation style).

George Sloata 3 months ago · 0 Likes

Great post !
I needed a circular buffer example, as I am new to cpp. But I have no idea how to change this to use (Arduino) String. Do you have any advice ? Thank you !

Phillip Johnston (http://www.about.me/phillip.johnston)

3 months ago · 0 Likes
(http://www.about.me/phillip.johnston)

There are a few options:

1) Use the circular buffer to manage type `char`. You could add an API to return the pointer to the underlying buffer, which would allow you to treat the full buffer contents as a C-string.
2) Declare an Arduino String, call `reserve` for the maximum size of the buffer, and then manage the buffer using the ring-span-lite library:
https://github.com/martinmoene/ring-span-lite

Manuel Furio Ferrero 2 months ago · 0 Likes

Hi, thanks for the tutorial.
I can't use the malloc so I need a different approach.
I don't understant why I need to create a buffer on my application and then pass it to the circular_buf_init where it allocate another buffer.
Can't I simply allocate a static buffer somewhere in my application or in the circular_buffer.c file and use that?
In your example you use the malloc for the application buffer and then you pass it to the init where you use again a malloc to create a new buffer:
uint8_t * buffer = malloc(EXAMPLE_BUFFER_SIZE * sizeof(uint8_t));
cbuf_handle_t cbuf = circular_buf_init(buffer, EXAMPLE_BUFFER_SIZE);

Can you please clarify this?
Thanks in advance.

Phillip Johnston (http://www.about.me/phillip.johnston)

(http://www.about.me/phillip.johnston) 2 months ago · 0 Likes

Hi Manuel,

malloc was used at the time for ease of implementation; the main goal was to show how a circular buffer works and how to hide information in C (i.e., hiding the internal implementation of the circular buffer.

The problem is that the circular buffer structure definition is not available in the C header file. That means you can only declare a pointer to it in your code. There are two solutions:

1. Don't use information hiding, and move the struct definition to the header. The init function no longer needs to return a handle; you use the address of your module's locally declared struct as the handle.

2. Use information hiding and create a pool of structs inside the circular buffer library implementation. You could control this as a compiler definition, such as `CIRCULAR_BUFFER_CONTAINER_POOL_SIZE`. You need a way of tracking structs that are currently allocated (such as `bool inUse` flag in the struct), and a lock if using in multi-threaded code. Instead of `malloc`ing the container struct, you would assign the next available struct from the pool to the user. Instead of `free`, you would return this struct to

the pool.

Does that help?

Manuel Furio Ferrero 2 months ago · 0 Likes

Never mind: once I read again the code I got it: the first malloc is for the actual buffer, the second one is for the circualr buffer struct. Both of them are needed. I ended up using a global variable for the struc and rearranging the init to use that and return its address.

Phillip Johnston (http://www.about.me/phillip.johnston) 2 months ago · 0 Likes

(http://www.about.me/phillip.johnston)

That will work. See my response to your previous comment in case you need multiple circular buffers. I'll draft another article on making this static-memory friendly.

Costas 2 months ago · 0 Likes

Thank for the very clear demonstration and kudos.
Is there a way to read the values from the buffer without popping them out?

Phillip Johnston (http://www.about.me/phillip.johnston) 2 months ago · 0 Likes

(http://www.about.me/phillip.johnston)

Not in this design. But you could easily add a front() or peek() API which returns the next valid value without popping it.

gyuest666 A month ago · 0 Likes

How would I go about redsigning this buffer to hold a custom struct instead of uint8_t?

You state:

"I have chosen uint8_t as the underlying data type in this implementation. You can use any particular type that you like - just be careful to handle the underlying buffer and number of bytes appropriately."

I have created custom struct, buffer for that struct and changed all the functions to either accept or return the new buffer type and new struct type.

This seems to work up to a point. Sometimes I get hard fault, at the function circular_buf_put() (Same result for circular_buf_put2()), I can not figure out for life of me what causes it.

Last snapshot from the buffer handle shows me the buffer->max value to be 65000 or some other ridiculous value, it obviously was not meant to be modified after the initialisation. That value seems to imply it got set to minus one maybe?

I thought since the struct now is composed of multiple types, pointer would need to be incremented additional number of times, but that doesn't seem like the solution either.

This is running on a microcontroller, but with only a single interrupt which populates the buffer with structs and a single thread that depopulates the buffer.

Could you expand on your "Thread Safety with the Lookahead Method" ?

Would implementation of that involve checking whether the buffer is full - 1 before deciding whether new data can be inserted or rejected?

Thank you for doing this write up, it has helped me tremendously.

Newer Post
Serial Debugging On the Particle Electron (/blog/2017/5/6/debugging-on-the-particle-electron)

Older Post
Getting Started: Developing for Particle Electron on Your Host (/blog/2017/5/4/getting-started-with-particle-electron)