# Ringbuffer

***Release 1.2.0***

**Tilen MAJERLE**

**Dec 04, 2019**

# CONTENTS

*Download library* · Github

CHAPTER

ONE

# FEATURES

- Written in ANSI C99, compatible with size_t for size data types
- Platform independent, no architecture specific code
- FIFO (First In First Out) buffer implementation
- No dynamic memory allocation, data is static array
- Uses optimized memory copy instead of loops to read/write data from/to memory
- Thread safe when used as pipe with single write and single read entries
- Interrupt safe when used as pipe with single write and single read entries
- Suitable for DMA transfers from and to memory with zero-copy overhead between buffer and application memory
- Supports data peek, skip for read and advance for write
- User friendly MIT license

CHAPTER

TWO

# REQUIREMENTS

- C compiler

CHAPTER

# THREE

# EXAMPLE CODE

Minimalistic example code to read and write data to buffer

```
/* Buffer variables */
ringbuff_t buff;                            /* Declare ring buffer structure */
uint8_t buff_data[8];                       /* Declare raw buffer data array */

/* Application variables */
uint8_t data[2];                            /* Application working data */
size_t len;

/* Application code ... */
ringbuff_init(&buff, buff_data, sizeof(buff_data)); /* Initialize buffer */

/* Write 4 bytes of data */
ringbuff_write(&buff, "0123", 4);

/* Try to read buffer */
/* len holds number of bytes read */
/* Read until len == 0, when buffer is empty */
while ((len = ringbuff_read(&buff, data, sizeof(data))) > 0) {
    printf("Successfully read %d bytes\r\n", (int)len);
}
```

CHAPTER

FOUR

# TABLE OF CONTENTS

## 4.1 Get started

### 4.1.1 Download library

Library is primarly hosted on Github.

- Download latest release from releases area on Github
- Clone *develop* branch for latest development

#### Download from releases

All releases are available on Github releases releases area.

#### Clone from Github

#### First-time clone

- Download and install `git` if not already
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Run `git clone --recurse-submodules https://github.com/MaJerle/ringbuff` command to clone repository including submodules or
- Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/ringbuff` to clone *development* branch
- Navigate to `examples` directory and run favourite example

#### Update cloned to latest version

- Open console and navigate to path in the system where your resources repository is. Use command `cd your_path`
- Run `git pull origin master --recurse-submodules` command to pull latest changes and to fetch latest changes from submodules
- Run `git submodule foreach git pull origin master` to update & merge all submodules

### 4.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page.

- Copy `ringbuff` folder to your project
- Add `ringbuff/src/include` folder to *include path* of your toolchain
- Add source files from `ringbuff/src/` folder to toolchain build
- Build the project

### 4.1.3 Minimal example code

Run below example to test and verify library

```
/* Buffer variables */
ringbuff_t buff;                            /* Declare ring buffer structure */
uint8_t buff_data[8];                       /* Declare raw buffer data array */

/* Application variables
uint8_t data[2];                            /* Application working data */

/* Application code ... */
ringbuff_init(&buff, buff_data, sizeof(buff_data)); /* Initialize buffer */

/* Write 4 bytes of data */
ringbuff_write(&buff, "0123", 4);

/* Print number of bytes in buffer */
printf("Bytes in buffer: %d\r\n", (int)ringbuff_get_full(&buff));

/* Will print "4" */
```

## 4.2 User manual

### 4.2.1 How it works

This section shows different buffer corner cases and provides basic understanding how data are managed internally.

Fig. 1: Different buffer corner cases

Let's start with reference of abbreviations in picture:

- `R` represents *Read* pointer. Read on read/write operations. Modified on read operation only
- `W` represents *Write* pointer. Read on read/write operations. Modified on write operation only
- `S` represents *Size* of buffer. Used on all operations, never modified (atomic value)
  - Valid number of `W` and `R` pointers are between `0` and `S - 1`
- Buffer size is `S = 8`, thus valid number range for `W` and `R` pointers is `0 - 7`.
  - `R` and `W` numbers overflow at `S`, thus valid range is always `0, 1, 2, 3, ..., S - 2, S - 1, 0, 1, 2, 3, ..., S - 2, S - 1, 0, ...`

  - Example `S = 4`: `0, 1, 2, 3, 0, 1, 2, 3, 0, 1, ...`
- Maximal number of bytes buffer can hold is always `S - 1`, thus example buffer can hold up to `7` bytes
- `R` and `W` pointers always point to the next read/write operation
- When `W == R`, buffer is considered empty.
- When `W == R - 1`, buffer is considered full.
  - `W == R - 1` is valid only if `W` and `R` overflow at buffer size `S`.
  - Always add `S` to calculated number and then use modulus `S` to get final value

---

**Note:** Example 1, add `2` numbers: `2 + 3 = (3 + 2 + S) % S = (3 + 2 + 4) % 4 = (5 + 4) % 4 = 1`

Example 2, subtract `2` numbers: `2 - 3 = (2 - 3 + S) % S = (2 - 3 + 4) % 4 = (-1 + 4) % 4 = 3`

---

Fig. 2: Different buffer corner cases

Different image cases:

- Case **A**: Buffer is empty as `W == R = 0 == 0`
- Case **B**: Buffer holds `W - R = 4 - 0 = 4` bytes as `W > R`
- Case **C**: Buffer is full as `W == R - 1` or `7 == 0 - 1` or `7 = (0 - 1 + S) % S = (0 - 1 + 8) % 8 = (-1 + 8) % 8 = 7`
  - `R` and `W` can hold `S` different values, from `0` to `S - 1`, that is modulus of `S`
  - Buffer holds `W - R = 7 - 0 = 7` bytes as `W > R`
- Case **D**: Buffer holds `S - (R - W) = 8 - (5 - 3) = 6` bytes as `R > W`
- Case **E**: Buffer is full as `W == R - 1` (`4 = 5 - 1`) and holds `S - (R - W) = 8 - (5 - 4) ) = 7` bytes

### 4.2.2 DMA on embedded systems

One of the key features of ringbuffer library is that it can be seamlessly integrated with DMA controllers on embedded systems.

---

**Note:** DMA stands for *Direct Memory Access* controller and is usually used to off-load CPU. More about DMA is available on Wikipedia.

---

DMA controllers normally use source and destination memory addresses to transfer data in-between. This features, together with ringbuffer, allows seamless integration and zero-copy of application data at interrupts after DMA transfer has been completed. Some manual work is necessary to be handled, but this is very minor in comparison of writing byte-by-byte to buffer at (for example) each received character.

Below are `2` common use cases:

- DMA transfers data from ringbuffer memory to (usually) some hardware IP
- DMA transfers data from hardware IP to memory

### Zero-copy data from memory

This describes how to pass ringbuffer output memory address as pointer to DMA (or any other processing function). After all the data are successfully processed, application can skip processed data and free ringbuff for new data being written to it.

- Case **A**: Initial state, buffer is full and holds `7` bytes
- Case **B**: State after skipping `R` pointer for `3` bytes. Buffer now holds `4` remaining bytes
- Case **C**: Buffer is empty, no more memory available for read operation

Code example:

```
/* Declare buffer variables */
ringbuff_t buff;
uint8_t buff_data[8];

size_t len;
uint8_t* data;

/* Initialize buffer, use buff_data as data array */
ringbuff_init(&buff, buff_data, sizeof(buff_data));

/* Use write, read operations, process data */
/* ... */

/* IMAGE PART A */

/* At this stage, we have buffer as on image above */
/* R = 5, W = 4, buffer is considered full */

/* Get length of linear memory at read pointer */
/* Function returns 3 as we can read 3 bytes from buffer in sequence */
/* When function returns 0, there is no memory available in the buffer for read␣
↪anymore */
if ((len = ringbuff_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at read address */
    /* Function returns &buff_data[5] */
    data = ringbuff_get_linear_block_read_address(&buff);

    /* Send data via DMA and wait to finish (for sake of example) */
    send_data(data, len);

    /* Now skip sent bytes from buffer = move read pointer */
    ringbuff_skip(&buff, len);

    /* Now R points to top of buffer, R = 0 */
    /* At this point, we are at image part B */
}

/* IMAGE PART B */

/* Get length of linear memory at read pointer */
/* Function returns 4 as we can read 4 bytes from buffer in sequence */
/* When function returns 0, there is no memory available in the buffer for read␣
↪anymore */
if ((len = ringbuff_get_linear_block_read_length(&buff)) > 0) {
```

(continues on next page)

(continued from previous page)

```
    /* Get pointer to first element in linear block at read address */
    /* Function returns &buff_data[0] */
    data = ringbuff_get_linear_block_read_address(&buff);

    /* Send data via DMA and wait to finish (for sake of example) */
    send_data(data, len);

    /* Now skip sent bytes from buffer = move read pointer */
    /* Read pointer is moved for len bytes */
    ringbuff_skip(&buff, len);

    /* Now R points to 4, that is R == W and buffer is now empty */
    /* At this point, we are at image part C */
}

/* IMAGE PART C */

/* Buffer is considered empty as R == W */
```

Part **A** on image clearly shows that not all data bytes are linked in single contiguous block of memory. To send all bytes from ringbuff, it might be necessary to repeat procedure multiple times

```
/* Initialization part skipped */

/* Get length of linear memory at read pointer */
/* When function returns 0, there is no memory available in the buffer for read␣
↪anymore */
while ((len = ringbuff_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at read address */
    data = ringbuff_get_linear_block_read_address(&buff);

    /* If max length needs to be considered */
    /* simply decrease it and use smaller len on skip function */
    if (len > max_len) {
        len = max_len;
    }

    /* Send data via DMA and wait to finish (for sake of example) */
    send_data(data, len);

    /* Now skip sent bytes from buffer = move read pointer */
    ringbuff_skip(&buff, len);
}
```

### Zero-copy data to memory

Similar to reading data from buffer with zero-copy overhead, it is possible to write to ringbuff with zero-copy overhead too. Only difference is that application now needs pointer to write memory address and length of maximal number of bytes to directly copy in buffer. After processing is successful, buffer advance operation is necessary to manually increase write pointer and to increase number of bytes in buffer.

- Case **A**: Initial state, buffer is empty as `R == W`
  - Based on `W` pointer position, application could write `4` bytes to contiguous block of memory

- Case **B**: State after advancing *W* pointer for *4* bytes. Buffer now holds *4* bytes and has 3 remaining available
- Case **C**: Buffer is full, no more free memory available for write operation

Code example:

```
/* Declare buffer variables */
ringbuff_t buff;
uint8_t buff_data[8];

size_t len;
uint8_t* data;

/* Initialize buffer, use buff_data as data array */
ringbuff_init(&buff, buff_data, sizeof(buff_data));

/* Use write, read operations, process data */
/* ... */

/* IMAGE PART A */

/* At this stage, we have buffer as on image above */
/* R = 4, W = 4, buffer is considered empty */

/* Get length of linear memory at write pointer */
/* Function returns 4 as we can write 4 bytes to buffer in sequence */
/* When function returns 0, there is no memory available in the buffer for write␣
↪anymore */
if ((len = ringbuff_get_linear_block_write_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at write address */
    /* Function returns &buff_data[4] */
    data = ringbuff_get_linear_block_write_address(&buff);

    /* Receive data via DMA and wait to finish (for sake of example) */
    /* Any other hardware may directly write to data array */
    /* Data array has len bytes length */
    /* Or use memcpy(data, my_array, len); */
    receive_data(data, len);

    /* Now advance buffer for written bytes to buffer = move write pointer */
    /* Write pointer is moved for len bytes */
    ringbuff_advance(&buff, len);

    /* Now W points to top of buffer, W = 0 */
    /* At this point, we are at image part B */
}

/* IMAGE PART B */

/* Get length of linear memory at write pointer */
/* Function returns 3 as we can write 3 bytes to buffer in sequence */
/* When function returns 0, there is no memory available in the buffer for write␣
↪anymore */
if ((len = ringbuff_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at write address */
    /* Function returns &buff_data[0] */
    data = ringbuff_get_linear_block_read_address(&buff);

    /* Receive data via DMA and wait to finish (for sake of example) */
```

(continues on next page)

(continued from previous page)

```
    /* Any other hardware may directly write to data array */
    /* Data array has len bytes length */
    /* Or use memcpy(data, my_array, len); */
    receive_data(data, len);

    /* Now advance buffer for written bytes to buffer = move write pointer */
    /* Write pointer is moved for len bytes */
    ringbuff_advance(&buff, len);

    /* Now W points to 3, R points to 4, that is R == W + 1 and buffer is now full */
    /* At this point, we are at image part C */
}

/* IMAGE PART C */

/* Buffer is considered full as R == W + 1 */
```

### Example for DMA transfer from memory

This is an example showing pseudo code for implementing data transfer using DMA with zero-copy overhead. For read operation purposes, application gets direct access to ringbuffer read pointer and length of contigouos memory.

It is assumed that after DMA transfer completes, interrupt is generated (embedded system) and buffer is skipped in the interrupt.

**Note:** Buffer skip operation is used to mark sent data as processed and to free memory for new writes to buffer

```
/* Buffer */
ringbuff_t buff;
uint8_t buff_data[8];

/* Working data length */
size_t len;

/* Send data function */
void send_data(void);

int
main(void) {
    /* Initialize buffer */
    ringbuff_init(&buff, buff_data, sizeof(buff_data));

    /* Write 4 bytes of data */
    ringbuff_write(&buff, "0123", 4);

    /* Send data over DMA */
    send_data();

    while (1);
}

/* Send data over DMA */
void
```

(continues on next page)

(continued from previous page)

```
send_data(void) {
    /* If len > 0, DMA transfer is on-going */
    if (len) {
        return;
    }

    /* Get maximal length of buffer to read data as linear memory */
    len = ringbuff_get_linear_block_read_length(&buff);
    if (len) {
        /* Get pointer to read memory */
        uint8_t* data = ringbuff_get_linear_block_read_address(&buff);

        /* Start DMA transfer */
        start_dma_transfer(data, len);
    }

    /* Function does not wait for transfer to finish */
}

/* Interrupt handler */
/* Called on DMA transfer finish */
void
DMA_Interrupt_handler(void) {
    /* Transfer finished */
    if (len) {
        /* Now skip the data (move read pointer) as they were successfully␣
↪transferred over DMA */
        ringbuff_skip(&buff, len);

        /* Reset length = DMA is not active */
        len = 0;

        /* Try to send more */
        send_data();
    }
}
```

## 4.3 Tips & tricks

### 4.3.1 Application buffer size

Buffer size shall always be `1` byte bigger than anticipated data size.

When application uses buffer for some data block *N* times, it is advised to set buffer size to *1* byte more than *N* * *block_size* is. This is due to *R* and *W* pointers alignment.

**Note:** For more information, check *How it works*.

```
/* Number of data blocks to write */
#define N          3

/* Create custom data structure */
```

(continues on next page)

(continued from previous page)

```
/* Data is array of 2 32-bit words, 8-bytes */
uint32_t d[2];

/* Create buffer structures */
ringbuff_t buff_1;
ringbuff_t buff_2;

/* Create data for buffers. Use sizeof structure, multiplied by N (for N instances) */
/* Buffer with + 1 bytes bigger memory */
uint8_t buff_data_1[sizeof(d) * N + 1];
/* Buffer without + 1 at the end */
uint8_t buff_data_2[sizeof(d) * N];

/* Write result values */
size_t len_1;
size_t len_2;

/* Initialize buffers */
ringbuff_init(&buff_1, buff_data_1, sizeof(buff_data_1));
ringbuff_init(&buff_2, buff_data_2, sizeof(buff_data_2));

/* Write data to buffer */
for (size_t i = 0; i < N; ++i) {
    /* Prepare data */
    d.a = i;
    d.b = i * 2;

    /* Write data to both buffers, memory copy from d to buffer */
    len_1 = ringbuff_write(&buff_1, d, sizeof(d));
    len_2 = ringbuff_write(&buff_2, d, sizeof(d));

    /* Print results */
    printf("Write buffer 1: %d/%d bytes; buffer 2: %d/%d\r\n",
        (int)len_1, (int)sizeof(d),
        (int)len_2, (int)sizeof(d));
}
```

When the code is executed, it procudes following output:

```
Write: buffer 1: 8/8; buffer 2: 8/8
Write: buffer 1: 8/8; buffer 2: 8/8
Write: buffer 1: 8/8; buffer 2: 7/8 <-- See here -->
```

## 4.4 API reference

List of all the modules:

**Contents**

### 4.4.1 Ring buffer

**Contents**



*group* **RINGBUFF**
Generic ring buffer manager.

#### Defines

**BUF_PREF**(x)
Buffer function/typedef prefix string.

It is used to change function names in zero time to easily re-use same library between applications. Use `#define BUF_PREF(x) my_prefix_ ## x` to change all function names to (for example) `my_prefix_buff_init`

**Note** Modification of this macro must be done in header and source file aswell

#### Functions

uint8_t **ringbuff_init**(*ringbuff_t* *buff, void *buffdata, size_t *size*)
Initialize buffer handle to default values with size and buffer data array.

**Return** `1` on success, `0` otherwise

**Parameters**

- `[in] buff`: Buffer handle
- `[in] buffdata`: Pointer to memory to use as buffer data
- `[in] size`: Size of `buffdata` in units of bytes Maximum number of bytes buffer can hold is `size - 1`

void **ringbuff_free**(*ringbuff_t* *buff)
Free buffer memory.

**Note** Since implementation does not use dynamic allocation, it just sets buffer handle to `NULL`

**Parameters**

- `[in] buff`: Buffer handle

void **ringbuff_reset**(*ringbuff_t* *buff)
Resets buffer to default values. Buffer size is not modified.

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_write**(*ringbuff_t* *buff, **const** void *data, size_t *btw*)
Write data to buffer Copies data from `data` array to buffer and marks buffer as full for maximum `count` number of bytes.

**Return** Number of bytes written to buffer. When returned value is less than `btw`, there was no enough memory available to copy full data array

**Parameters**

- `[in] buff`: Buffer handle
- `[in] data`: Pointer to data to write into buffer
- `[in] btw`: Number of bytes to write

size_t **ringbuff_read**(*ringbuff_t *buff*, void *data*, size_t *btr*)
Read data from buffer Copies data from buffer to `data` array and marks buffer as free for maximum `btr` number of bytes.

**Return** Number of bytes read and copied to data array

**Parameters**

- `[in] buff`: Buffer handle
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btr`: Number of bytes to read

size_t **ringbuff_peek**(*ringbuff_t *buff*, size_t *skip_count*, void *data*, size_t *btp*)
Read from buffer without changing read pointer (peek only)

**Return** Number of bytes peeked and written to output array

**Parameters**

- `[in] buff`: Buffer handle
- `[in] skip_count`: Number of bytes to skip before reading data
- `[out] data`: Pointer to output memory to copy buffer data to
- `[in] btp`: Number of bytes to peek

size_t **ringbuff_get_free**(*ringbuff_t *buff*)
Get number of bytes in buffer available to write.

**Return** Number of free bytes in memory

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_get_full**(*ringbuff_t *buff*)
Get number of bytes in buffer available to read.

**Return** Number of bytes ready to be read

**Parameters**

- `[in] buff`: Buffer handle

void ***ringbuff_get_linear_block_read_address**(*ringbuff_t *buff*)
Get linear address for buffer for fast read.

**Return** Linear buffer start address

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_get_linear_block_read_length** (*ringbuff_t *buff*)
Get length of linear block address before it overflows for read operation.

**Return** Linear buffer size in units of bytes for read operation

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_skip** (*ringbuff_t *buff*, size_t *len*)
Skip (ignore; advance read pointer) buffer data Marks data as read in the buffer and increases free memory for up to `len` bytes.

**Note** Useful at the end of streaming transfer such as DMA

**Return** Number of bytes skipped

**Parameters**

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to skip and mark as read

void ***ringbuff_get_linear_block_write_address** (*ringbuff_t *buff*)
Get linear address for buffer for fast read.

**Return** Linear buffer start address

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_get_linear_block_write_length** (*ringbuff_t *buff*)
Get length of linear block address before it overflows for write operation.

**Return** Linear buffer size in units of bytes for write operation

**Parameters**

- `[in] buff`: Buffer handle

size_t **ringbuff_advance** (*ringbuff_t *buff*, size_t *len*)
Advance write pointer in the buffer. Similar to skip function but modifies write pointer instead of read.

**Note** Useful when hardware is writing to buffer and application needs to increase number of bytes written to buffer by hardware

**Return** Number of bytes advanced for write operation

**Parameters**

- `[in] buff`: Buffer handle
- `[in] len`: Number of bytes to advance

**struct ringbuff_t**
*#include* <*ringbuff.h*> Buffer structure.

### Public Members

uint8_t ***buff**

Pointer to buffer data. Buffer is considered initialized when `buff != NULL` and `size > 0`

size_t **size**

Size of buffer data. Size of actual buffer is `1` byte less than value holds

size_t **r**

Next read pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

size_t **w**

Next write pointer. Buffer is considered empty when `r == w` and full when `w == r - 1`

# INDEX