LwRB

Tilen MAJERLE

Aug 22, 2023

CONTENTS

Welcome to the documentation for version v3.0.0.

LwRB is a generic *FIFO* (First In; First Out) buffer library optimized for embedded systems.

Download library Getting started Open Github Donate

CHAPTER

ONE

FEATURES

- Written in C (C11), 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
- Implements support for event notifications
- User friendly MIT license

CHAPTER

TWO

REQUIREMENTS

- C compiler
- Less than 1kB of non-volatile memory

CHAPTER

THREE

CONTRIBUTE

Fresh contributions are always welcome. Simple instructions to proceed:

1. Fork Github repository
2. Respect C style & coding rules used by the library
3. Create a pull request to `develop` branch with new features or bug fixes

Alternatively you may:

1. Report a bug
2. Ask for a feature request

CHAPTER

FOUR

EXAMPLE CODE

Minimalistic example code to read and write data to buffer

Listing 1: Example code

```c
/* Declare rb instance & raw data */
lwrb_t buff;
uint8_t buff_data[8];

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

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

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

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

CHAPTER

FIVE

LICENSE

```
MIT License

Copyright (c) 2023 Tilen MAJERLE

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER

SIX

TABLE OF CONTENTS

6.1 Getting started

Getting started may be the most challenging part of every new library. This guide is describing how to start with the library quickly and effectively

6.1.1 Download library

Library is primarly hosted on Github.

You can get it by:

- Downloading latest release from releases area on Github
- Cloning `main` branch for latest stable version
- Cloning `develop` branch for latest development

Download from releases

All releases are available on Github releases area.

Clone from Github

First-time clone

This is used when you do not have yet local copy on your machine.

- Make sure `git` is installed.
- Open console and navigate to path in the system to clone repository to. Use command `cd your_path`
- Clone repository with one of available options below
  - Run `git clone --recurse-submodules https://github.com/MaJerle/lwrb` command to clone entire repository, including submodules
  - Run `git clone --recurse-submodules --branch develop https://github.com/MaJerle/lwrb` to clone *development* branch, including submodules
  - Run `git clone --recurse-submodules --branch main https://github.com/MaJerle/lwrb` to clone *latest stable* branch, including submodules
- Navigate to `examples` directory and run favourite example

Update cloned to latest version

- Open console and navigate to path in the system where your repository is located. Use command `cd your_path`
- Run `git pull origin main` command to get latest changes on `main` branch
- Run `git pull origin develop` command to get latest changes on `develop` branch
- Run `git submodule update --init --remote` to update submodules to latest version

Note: This is preferred option to use when you want to evaluate library and run prepared examples. Repository consists of multiple submodules which can be automatically downloaded when cloning and pulling changes from root repository.

6.1.2 Add library to project

At this point it is assumed that you have successfully download library, either cloned it or from releases page. Next step is to add the library to the project, by means of source files to compiler inputs and header files in search path

- Copy `lwrb` folder to your project, it contains library files
- Add `lwrb/src/include` folder to *include path* of your toolchain. This is where *C/C++* compiler can find the files during compilation process. Usually using `-I` flag
- Add source files from `lwrb/src/` folder to toolchain build. These files are built by *C/C++* compiler. CMake configuration comes with the library, allows users to include library in the project as **subdirectory** and **library**.y
- Build the project

6.1.3 Minimal example code

To verify proper library setup, minimal example has been prepared. Run it in your main application file to verify its proper execution

Listing 1: Absolute minimum example

```
#include "lwrb/lwrb.h"

/* Declare rb instance & raw data */
lwrb_t buff;
uint8_t buff_data[8];

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

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

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

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

(continues on next page)

(continued from previous page)

```
18
19 /* Will print "4" */
```

6.2 User manual

6.2.1 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

6.2.2 Events

When using LwRB in the application, it may be useful to get notification on different events, such as info when something has been written or read to/from buffer.

Library has support for events that get called each time there has been a modification in the buffer data, that means on every read or write operation.

Some use cases:

- Notify application layer that LwRB operation has been executed and send debug message
- Unlock semaphore when sufficient amount of bytes have been written/read from/to buffer when application uses operating system
- Write notification to message queue at operating system level to wakeup another task

---

Note: Every operation that modified *read* or *write* internal pointers, is considered as read or write operation. An exception is *reset* event that sets both internal pointers to *0*

Listing 2: Example code for events

```
/**
 * \brief           Buffer event function
 */
void
my_buff_evt_fn(lwrb_t* buff, lwrb_evt_type_t type, size_t len) {
    switch (type) {
        case LWRB_EVT_RESET:
            printf("[EVT] Buffer reset event!\r\n");
            break;
        case LWRB_EVT_READ:
            printf("[EVT] Buffer read event: %d byte(s)!\r\n", (int)len);
            break;
        case LWRB_EVT_WRITE:
            printf("[EVT] Buffer write event: %d byte(s)!\r\n", (int)len);
            break;
        default: break;
    }
}

/* Later in the code... */
lwrb_t buff;
uint8_t buff_data[8];

/* Init buffer and set event function */
lwrb_init(&buff, buff_data, sizeof(buff_data));
lwrb_set_evt_fn(&buff, my_buff_evt_fn);
```

6.2.3 DMA for embedded systems

One of the key features of LwRB 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 LwRB, 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 LwRB memory to (usually) some hardware IP
- DMA transfers data from hardware IP to memory

Zero-copy data from LwRB memory

This describes how to pass LwRB output memory address as pointer to DMA (or any other processing function). After data is successfully processed, application can skip processed data and mark buffer as free for new data being written to it.

Fig. 3: Data transfer from memory to hardware IP

- 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:

Listing 3: Skip buffer data after usage

```
#include "lwrb/lwrb.h"

/* Declare rb instance & raw data */
lwrb_t buff;
uint8_t buff_data[8];

size_t len;
uint8_t* data;

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

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

/* IMAGE PART A */

```

(continues on next page)

(continued from previous page)

```
/* 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 = lwrb_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at read address */
    /* Function returns &buff_data[5] */
    data = lwrb_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 */
    lwrb_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 = lwrb_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at read address */
    /* Function returns &buff_data[0] */
    data = lwrb_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 */
    lwrb_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 lwrb, it might be necessary to repeat procedure multiple times

Listing 4: Skip buffer data for non-contiguous block

```
/* 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 = lwrb_get_linear_block_read_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at read address */
    data = lwrb_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 */
    lwrb_skip(&buff, len);
}
```

Zero-copy data to LwRB memory

Similar to reading data from buffer with zero-copy overhead, it is possible to write to lwrb 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 into buffer. After successful processing, 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:

Listing 5: Advance buffer pointer for manually written bytes

```
/* Declare rb instance & raw data */
lwrb_t buff;
uint8_t buff_data[8];

size_t len;
uint8_t* data;

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

```

(continues on next page)

(continued from previous page)

```c
/* 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 = lwrb_get_linear_block_write_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at write address */
    /* Function returns &buff_data[4] */
    data = lwrb_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 */
    lwrb_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 = lwrb_get_linear_block_write_length(&buff)) > 0) {
    /* Get pointer to first element in linear block at write address */
    /* Function returns &buff_data[0] */
    data = lwrb_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 */
    lwrb_advance(&buff, len);

```

(continues on next page)

(continued from previous page)

```
    /* 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 LwRB read pointer and length of contiguous 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

Listing 6: DMA usage with buffer

```
/* Declare rb instance & raw data */
lwrb_t buff;
uint8_t buff_data[8];

/* Working data length */
volatile size_t len;

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

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

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

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

    while (1);
}

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

(continues on next page)

(continued from previous page)

```
    }

    /* Get maximal length of buffer to read data as linear memory */
    len = lwrb_get_linear_block_read_length(&buff);
    if (len > 0) {
        /* Get pointer to read memory */
        uint8_t* data = lwrb_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 > 0) {
        /* Now skip the data (move read pointer) as they were successfully transferred over DMA */
        lwrb_skip(&buff, len);

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

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

Tip: Check STM32 UART DMA TX RX Github repository for use cases.

6.2.4 Thread safety

Ring buffers are effectively used in embedded systems with or without operating systems. Common problem most of implementations have is linked to multi-thread environment (when using OS) or reading/writing from/to interrupts. This is linked to common question *What happens if I write to buffer while another thread is reading from it?*

One of the main requirements (beside being lightweight) of *LwRB* was to allow *read-while-write* or *write-while-read* operations. This is achieved only when there is single write entry point and single read exit point.

Fig. 4: Write and read operation with single entry and exit points

Often called and used as *pipe* to write (for example) raw data to the buffer allowing another task to process the data from another thread.

Note: No race-condition is introduced when application uses LwRB with single write entry and single read exit point. LwRB uses *C11* standard `stdatomic.h` library to ensure read and write operations are race-free for any platform supporting C11 and its respected atomic library.

Thread (or interrupt) safety, with one entry and one exit points, is achieved by storing actual buffer read and write pointer variables to the local ones before performing any calculation. Therefore multiple *conditional* checks are guaranteed to be performed on the same local variables, even if actual buffer pointers get modified.

- Read pointer could get changed by interrupt or another thread when application tries to write to buffer
- Write pointer could get changed by interrupt or another thread when application ties to read from buffer

Note: Even single entry and single exit points may introduce race condition, especially on smaller system, such as 8-bit or 16-bit system, or in general, where arbitrary type (normaly *size_t*) is *sizeof(type) < architecture_size*. This is solved by C11 atomic library, that ensures atomic reads and writes to key structure members

Thread safety gets completely broken when application does one of the following:

- Uses multiple write entry points to the single LwRB instance
- Uses multiple read exit points to the single LwRB instance
- Uses multiple read/write exit/entry points to the same LwRB instance

Fig. 5: Write operation to same LwRB instance from 2 threads. Write protection is necessary to ensure thread safety.

Fig. 6: Write operation to same LwRB instance from main loop and interrupt context. Write protection is necessary to ensure thread safety.

Fig. 7: Read operation from same LwRB instance from 2 threads. Read protection is necessary to ensure thread safety.

Above use cases are examples when thread safety gets broken. Application must ensure exclusive access only to the part in *dashed-red* rectangle.

Listing 7: Thread safety example

```
/* Declare variables */
lwrb_t rb;

/* 2 mutexes, one for write operations,
    one for read operations */
mutex_t m_w, m_r;

/* 4 threads below, 2 for write, 2 for read */
void
thread_write_1(void* arg) {
    /* Use write mutex */
    while (1) {
        mutex_get(&m_w);
        lwrb_write(&rb, ...);
```

(continues on next page)

Fig. 8: Read and write operations are executed from multiple threads. Both, read and write, operations require exclusive access.

(continued from previous page)

```
        mutex_give(&m_w);
    }
}

void
thread_write_2(void* arg) {
    /* Use write mutex */
    while (1) {
        mutex_get(&m_w);
        lwrb_write(&rb, ...);
        mutex_give(&m_w);
    }
}

void
thread_read_1(void* arg) {
    /* Use read mutex */
    while (1) {
        mutex_get(&m_r);
        lwrb_read(&rb, ...);
        mutex_give(&m_r);
    }
}

void
thread_read_2(void* arg) {
    /* Use read mutex */
    while (1) {
        mutex_get(&m_r);
        lwrb_read(&rb, ...);
        mutex_give(&m_r);
    }
}
```

Read and write operations can be used simultaneously hence it is perfectly valid if access is granted to *read* operation while *write* operation from one thread takes place.

Note: 2 different mutexes are used for read and write due to the implementation, allowing application to use buffer in *read-while-write* and *write-while-read* mode. Mutexes are used to prevent *write-while-write* and *read-while-read* operations respectively

Tip: For *multi-entry-point-single-exit-point* use case, *read* mutex is not necessary. For *single-entry-point-multi-exit-point* use case, *write* mutex is not necessary.

Tip: Functions considered as *read* operation are `read`, `skip`, `peek` and `linear read`. Functions considered as *write*

operation are `write`, `advance` and `linear write`.

6.3 Tips & tricks

6.3.1 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*.

Listing 8: Application buffer size assignment

```
#include "lwrb/lwrb.h"

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

/* Create custom data structure */
/* Data is array of 2 32-bit words, 8-bytes */
uint32_t d[2];

/* Create buffer structures */
lwrb_t buff_1;
lwrb_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 */
lwrb_init(&buff_1, buff_data_1, sizeof(buff_data_1));
lwrb_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 */
```

(continues on next page)

(continued from previous page)

```
    len_1 = lwrb_write(&buff_1, d, sizeof(d));
    len_2 = lwrb_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 produces following output:

Listing 9: Application buffer size assignment 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 -->
```

6.4 API reference

List of all the modules:

6.4.1 LwRB

group **LWRB**

Lightweight ring buffer manager.

Typedefs

typedef atomic_ulong **lwrb_ulong_t**

typedef void (***lwrb_evt_fn**)(struct lwrb *buff, *lwrb_evt_type_t* evt, size_t bp)

Event callback function type.

Param buff **[in]** Buffer handle for event

Param evt **[in]** Event type

Param bp **[in]** Number of bytes written or read (when used), depends on event type

Enums

enum **lwrb_evt_type_t**

Event type for buffer operations.

Values:

enumerator **LWRB_EVT_READ**

Read event

enumerator **LWRB_EVT_WRITE**

Write event

enumerator **LWRB_EVT_RESET**

Reset event

Functions

uint8_t **lwrb_init**(*lwrb_t* *buff, void *buffdata, size_t size)

Initialize buffer handle to default values with size and buffer data array.

Parameters

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

Returns 1 on success, `0` otherwise

uint8_t **lwrb_is_ready**(*lwrb_t* *buff)

Check if buff is initialized and ready to use.

Parameters **buff** – **[in]** Buffer handle

Returns 1 if ready, `0` otherwise

void **lwrb_free**(*lwrb_t* *buff)

Free buffer memory.

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

Parameters **buff** – **[in]** Buffer handle

void **lwrb_reset**(*lwrb_t* *buff)

Resets buffer to default values. Buffer size is not modified.

Note: This function is not thread safe. When used, application must ensure there is no active read/write operation

Parameters **buff** – **[in]** Buffer handle

void **lwrb_set_evt_fn**(*lwrb_t* *buff, *lwrb_evt_fn* fn)

Set event function callback for different buffer operations.

Parameters

- **buff** – **[in]** Buffer handle
- **evt_fn** – **[in]** Callback function

size_t **lwrb_write**(*lwrb_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 `btw` number of bytes.

Parameters

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

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

size_t **lwrb_read**(*lwrb_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.

Parameters

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

Returns Number of bytes read and copied to data array

size_t **lwrb_peek**(const *lwrb_t* *buff, size_t skip_count, void *data, size_t btp)

Read from buffer without changing read pointer (peek only)

Parameters

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

Returns Number of bytes peeked and written to output array

size_t **lwrb_get_free**(const *lwrb_t* *buff)

Get available size in buffer for write operation.

Parameters **buff** – **[in]** Buffer handle

Returns Number of free bytes in memory

size_t **lwrb_get_full**(const *lwrb_t* *buff)

Get number of bytes currently available in buffer.

Parameters **buff** – **[in]** Buffer handle

Returns Number of bytes ready to be read

void ***lwrb_get_linear_block_read_address**(const *lwrb_t* *buff)

Get linear address for buffer for fast read.

Parameters **buff** – **[in]** Buffer handle

Returns Linear buffer start address

size_t **lwrb_get_linear_block_read_length**(const *lwrb_t* *buff)

Get length of linear block address before it overflows for read operation.

Parameters **buff** – **[in]** Buffer handle

Returns Linear buffer size in units of bytes for read operation

size_t **lwrb_skip**(*lwrb_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

Parameters

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

Returns Number of bytes skipped

void ***lwrb_get_linear_block_write_address**(const *lwrb_t* *buff)

Get linear address for buffer for fast read.

Parameters **buff** – **[in]** Buffer handle

Returns Linear buffer start address

size_t **lwrb_get_linear_block_write_length**(const *lwrb_t* *buff)

Get length of linear block address before it overflows for write operation.

Parameters **buff** – **[in]** Buffer handle

Returns Linear buffer size in units of bytes for write operation

size_t **lwrb_advance**(*lwrb_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

---

Parameters

- **buff** – **[in]** Buffer handle

- **len** – **[in]** Number of bytes to advance

Returns Number of bytes advanced for write operation

uint8_t **lwrb_find**(const *lwrb_t* *buff, const void *bts, size_t len, size_t start_offset, size_t *found_idx)

Searches for a *needle* in an array, starting from given offset.

Note: This function is not thread-safe.

Parameters

- **buff** – Ring buffer to search for needle in
- **bts** – Constant byte array sequence to search for in a buffer
- **len** – Length of the
  - bts array
- **start_offset** – Start offset in the buffer
- **found_idx** – Pointer to variable to write index in array where bts has been found Must not be set to `NULL`

Returns 1 if

- bts found, `0` otherwise

size_t **lwrb_overwrite**(*lwrb_t* *buff, const void *data, size_t btw)

Similar to *lwrb_write*, writes data to buffer, will overwrite existing values.

Note: Functionality is primary two parts, always writes some linear region, then writes the wrap region if there is more data to write. The r indicator is advanced if w overtakes it. This operation is a read op as well as a write op. For thread-safety mutexes may be desired, see documentation.

Parameters

- **buff** – **[in]** Buffer handle
- **data** – **[in]** Data to write to ring buffer
- **btw** – **[in]** Bytes To Write, length

Returns Number of bytes written to buffer, will always return btw

size_t **lwrb_move**(*lwrb_t* *dest, *lwrb_t* *src)

Move one ring buffer to another, up to the amount of data in the source, or amount of data free in the destination.

Note: This operation is a read op to the source, on success it will update the r index. As well as a write op to the destination, and may update the w index. For thread-safety mutexes may be desired, see documentation.

Parameters

- **dest** – **[in]** Buffer handle that the copied data will be written to

- **src** – **[in]** Buffer handle that the copied data will come from. Source buffer will be effectively read upon operation.

Returns Number of bytes written to destination buffer

struct **lwrb_t**

#include <lwrb.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

lwrb_ulong_t **r**

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

lwrb_ulong_t **w**

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

lwrb_evt_fn **evt_fn**

Pointer to event callback function

6.5 Changelog

```
# Changelog

## Develop

## v3.0.0

- Added macros for optional STDATOMIC. Global `-DLWRB_DISABLE_ATOMIC` macro will disable␣
↪C11 `<stdatomic.h>` functionality.
- Add `lwrb_move` and `lwrb_overwrite`
- Fix `lwrb_find` which failed to properly search for tokens at corner cases

## v3.0.0-RC1

- Split CMakeLists.txt files between library and executable
- Change license year to 2022
- Update code style with astyle
- Minimum required version is C11, with requirement of `stdatomic.h` library
- Add `.clang-format` draft

## v2.0.3
```

(continues on next page)

(continued from previous page)

```
- Add `library.json` for Platform.IO

## v2.0.2

- Add `volatile` keyword to all local variables to ensure thread safety in highest␣
↪optimization
- Add local variables for all read and write pointer accesses
- Remove generic `volatile` keyword from func parameter and replace to struct member

## v2.0.1

- Fix wrong check for valid RB instance
- Apply code style settings with Artistic style options
- Add thread safety docs

## v2.0.0

- Break compatibility with previous versions
- Rename function prefixes to `lwrb` instead of `ringbuff`
- Add astyle code syntax correction

## v1.3.1

- Fixed missing `RINGBUFF_VOLATILE` for event callback causes compiler warnings or errors

## v1.3.0

- Added support for events on read/write or reset operation
- Added optional volatile parameter for buffer structure
- Fix bug in skip and advance operation to return actual amount of bytes processed
- Remove `BUF_PREF` parameter and rename with fixed `ringbuff_` prefix for all functions

## v1.2.0

- Added first sphinx documentation

## v1.1.0

- Code optimizations, use pre-increment instead of post
- Another code-style fixes

## v1.0.0

- First stable release
```

INDEX

L