Bern University
of Applied Sciences

Lab – 01

Ring Buffer Implementation

Andreas Habegger | Adrian Steiner
BTS4230 | Version 1.0.0 of 17.03.2025

Please be aware that the content is subject to change at any time. For the latest version, please check the website.

This Lab is based on the problem of Ex-22: UART Interrupt Transmit. It shows a possible solution how to organise data in a microcontroller without allocating memory during runtime. In this lab, you will implement your own ring buffer and use it to store multiple messages and transmit them using UART communication.

Objectives

- Advantages and disadvantages of a ring buffer
- Implementation of a C ring buffer module
- Implementation of an abstraction layer for C strings based on the ring buffer module
- HAL UART Transmit without DMA or interrupt
- Base timer with interrupt

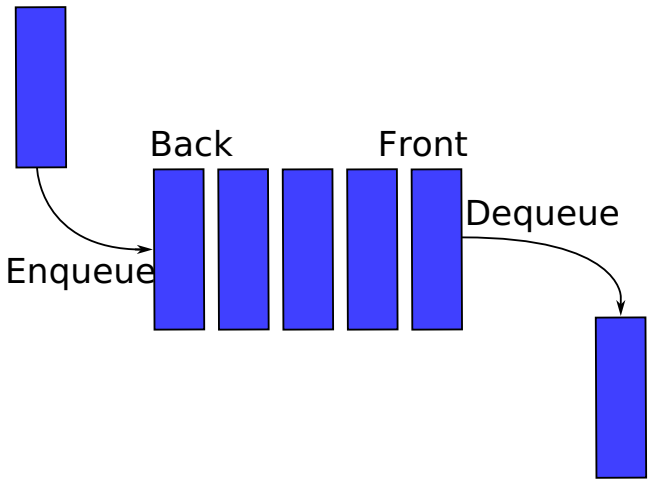
Outcomes

- Understanding of a ring buffer
- Using of void pointer, function pointer and callback functions
- Memory management in a MCU
- Interrupts and superloop behaviour.

Description

A circular buffer, circular queue, cyclic buffer or ring buffer is a data structure to store data in a single, fix sized memory buffer. It can be combined with a moving FIFO system (first in, first out) based on a fixed size array.

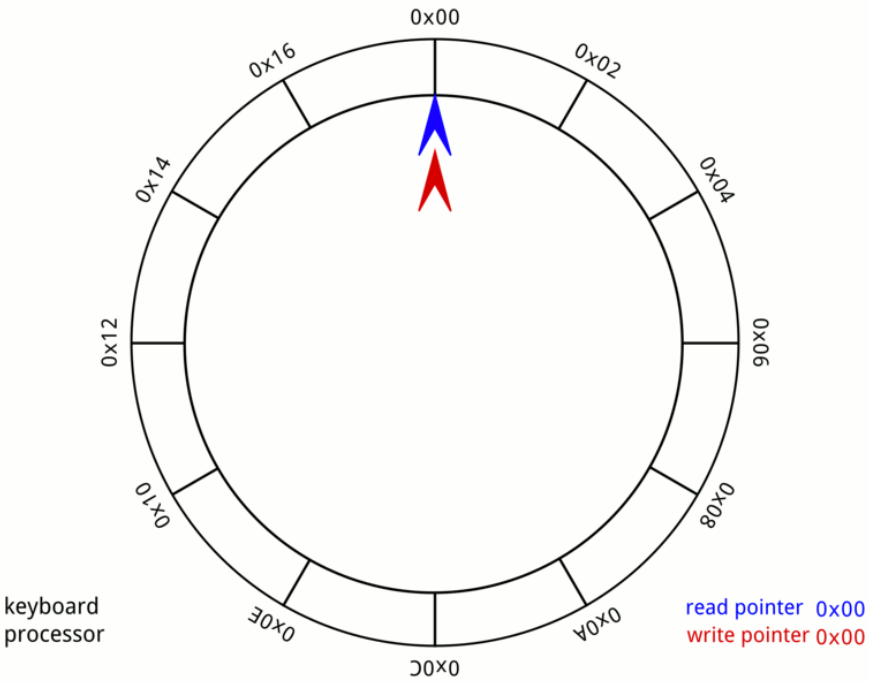
Representation of a FIFO queue



Representation of a data queue with a FIFO system. Source: Wikipedia/FIFO

When the front and back positions move through the array and jump back to the start at the end of the array, a ring is created which is given its name. See the animation below for a better understanding.

Ring buffer animation



Animiation of a ring buffer. Source: Wikipedia/Circular Buffer

The read pointer points to the next element that can be read to dequeue an element from the data structure. Whereas the write pointer points to the next free element to enqueue data. This can lead to two special cases:

- Read and write pointer point to the same address:

  ↪ This results in an empty buffer

- Write pointer is one position behind the read pointer:

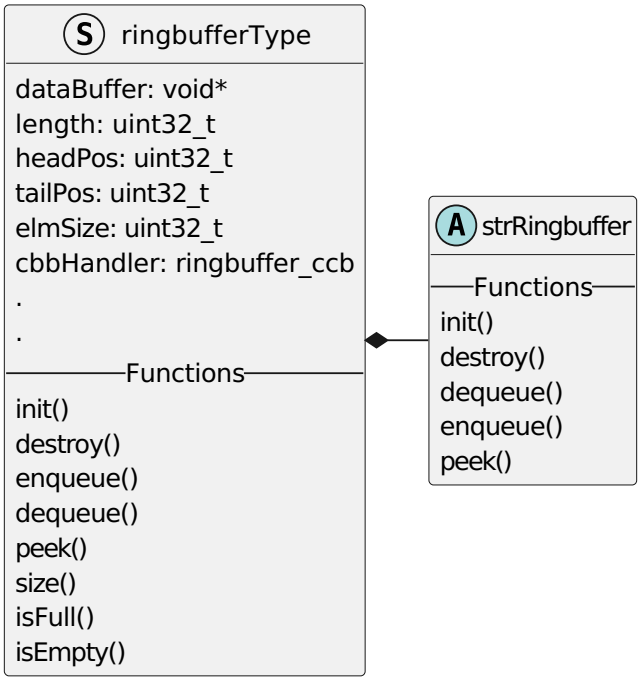
  ↪ This happens when the buffer is full and no more data can be stored.

The aim of this lab is to implement a module that provides a ring buffer with arbitrary data types. This module will then be inherited to handle C strings. Custom C strings are added to the module from the superloop and from an interrupt. Finally, the data stored in the ring buffer is sent to the computer via UART interface.

Tasks

The description explains how a ring buffer structure works and what is needed. The functionality of the ring buffer and the string abstraction can be summarised in the following diagram.

Base functionality of ring buffer module



> **Note**
>
> These are only the minimally used variables and functions. You can extend the structure members or functions of the modules.

For a better understanding, the struct members are briefly explained:

- `dataBuffer`: void pointer to the data array provided by the user. (Location where enqueued data will be stored.)
- `length`: The length of the array. (The buffers max. capacity.)
- `elmSize`: The size of one element of the data array. (Important since we use `void*` .)
- `tailPos`: The current tail position. (Where a new element can be enqueued/inserted.)
- `headPos`: The current head position. (First or the next dequeued element.)
- `cbbHandler`: Callback function. (Used to copy data to the buffer.)

The module has the following additional requirements, which must be met during implementation:

- The module should be able to handle any data. (Generic interface, use void pointers.)
- The size of the array (buffer) can be defined by the user.
- No memory allocation at run-time. (Use the array passed to the module, no heap operations within the module!)
- No errors or warnings.

Implementation

Start with a new STM32CubeMX project with UART support enabled.

Project Initialisation

1. Set up a STM32CubeMX project with the default configuration.
2. Enable UART2 with an usable baudrate without DMA or interrupts. (Interrupt capabilities can be used on a followup implementation.)

After initialising the project, implement the basic ring buffer module as described in the Tasks section and include all defined requirements.

Implementation of a base ring buffer module

3. Add the module (header and source files).
4. Define the interface of the module based on the UML element shown above.

Solution

The interface and doxygen description of module functions – this code goes into the module header file. The implementation – code of the module source file, is missing. You should be able to do this yourself, based on your skills, the function prototypes and the description.

Data Types

```
/**
 * @brief Ring buffer copy callback function to copy
 *  the element from the user into the ringbuffer.
 *
 * @param 1 destination
 * @param 2 source
 * @return 0 if successful, !=0 otherwise
 */
```

```
typedef int (*ringbuffer_ccb)(void *, const void *);

/**
 * @brief ringbuffer The ring buffer handler structure
 *
 */
typedef struct
{
  void *dataBuffer;          ///< buffer array reference
  uint32_t length;           ///< array length of buffer
  uint32_t headPos;          ///< current head position (first element)
  uint32_t tailPos;          ///< current tail position, (where a new element
can inserted)
  uint32_t
elmSize;          ///< Size of an element in the buffer (use sizeof operator
of an element)
  ringbuffer_ccb
cbbHandler; ///< Interface for callback function to copy data from and to the
buffer
} ringbufferType;

/**
 * @brief Used to encode different errors.
 * This StatusTypes are used to send back meaningful
 * information to the caller of module functions.
 *
 */
typedef enum {
  RINGBUFFER_OK = 0,         ///< No error
  RINGBUFFER_OVERFLOW,       ///< Buffer is full and no new data can be
inserted
  RINGBUFFER_EMPTY,          ///< current ring buffer is empty
  RINGBUFFER_ARGUMENT_ERROR, ///< Argument is not valid
  RINGBUFFER_ERROR           ///< Error in function or user copy callback
function
} ringbuffer_StatusType;
```

De-/Init Functions

```
/**
 * @brief Initialize a ring buffer structure
 *
 * @param rbHandler Reference to a ring buffer structure
 * @param buffer Reference to the buffer array
 * @param length The length of the buffer
 * @param elmSize Size of an element in the buffer
 * @param ccbHandler A function pointer to a element copy function
 * @return Status code -- see ringbuffer_StatusType
 */
ringbuffer_StatusType ringbuffer_init(
    ringbufferType *rbHandler,
    void *buffer,
    uint32_t length,
    uint32_t elmSize,
    ringbuffer_ccb ccbHandler);
```

```c
/**
 * @brief Destroy ring buffer structure
 *
 * @param rbHandler Reference to an initialized ring buffer structure.
 * @return Address of the buffer.
 * @note: Use the returned address to free the memory
 *        if it has been allocated on heap.
 */
void *ringbuffer_destroy(ringbufferType *rbHandler);
```

Action Functions

```c
/**
 * @brief Enqueue an element at the tail of the ring buffer.
 *
 * @param rbHandler Reference to the ring buffer structure
 * @param data The data to insert into the ring buffer
 * @return Status code -- see ringbuffer_StatusType
 */
ringbuffer_StatusType ringbuffer_enqueue(ringbufferType *rbHandler, void
*data);

/**
 * @brief Dequeue an element from the head of the ring buffer
 *
 * @param rbHandler Reference to the ringbuffer structure
 * @param data The data which will be dequeued from the ringbuffer
 * @return Status code -- see ringbuffer_StatusType
 */
ringbuffer_StatusType ringbuffer_dequeue(ringbufferType *rbHandler, void
**data);

/**
 * @brief Return the reference to the data
 *        at the head of the ring buffer without dequeuing it.
 *
 * @param rbHandler Reference to the ring buffer structure
 * @return Data stored in the element at the head of the ring buffer,
 *         or NULL if the queue is empty.
 */
void *ringbuffer_peek(const ringbufferType *rbHandler);

/**
 * @brief Number of elements stored in the ring buffer
 *
 * @param rbHandler Reference to the ring buffer structure
 * @return uint32_t Number of elements in the ring buffer
 */
uint32_t ringbuffer_size(const ringbufferType *rbHandler);
```

Support Functions

```c
/**
 * @brief Check whether buffer is full
 *
```

```
 * @param rbHandler Reference to the ring buffer structure
 * @return True if buffer is full, False otherwise
 */
bool ringbuffer_isFull(const ringbufferType *rbHandler);

/**
 * @brief Check whether buffer is empty
 *
 * @param rbHandler Reference to the ring buffer structure
 * @return True if buffer is empty, False otherwise
 */
bool ringbuffer_isEmpty(const ringbufferType *rbHandler);
```

5. Implement all previously defined functions of the module.

The new ring buffer module can be used to implement an inherited module for handle C strings. The purpose of this is to make it easier and less prone to errors to work with the base module.

C String Handler module

6. Implement a higher abstraction of the ring buffer that manages C strings.

Solution

The interface and doxygen description of module functions – this code goes into the module header file. The implementation – code of the module source file, is missing. You should be able to do this yourself, based on your skills, the function prototypes and the description.

C-Strings ring-buffer managed

```
/**
 * @brief More specific interface to handle C-Strings with a ring buffer.
 *         Used to initialize the underlining generic ring buffer.
 *
 * @param strRingBuf Reference to a ring buffer structure.
 * @param buffer Reference to a char array to manage the C-strings
 * @param length The length of the char array
 * @param strLength The max length of a C-string stored within the buffer
 * @return ringbuffer_status_t RINGBUFFER_OK if successful
 */
ringbuffer_StatusType strRingbuffer_init(
    ringbufferType *strRingBuf,
    char **buffer,
    uint32_t length,
    uint32_t strLength);

/**
 * @brief Destroy string ring buffer structure
```

```
 *
 * @param strRingBuf Reference to the ring buffer structure.
 * @return Address of the buffer.
 * @note: Use the returned address to free the memory
 *        if it has been allocated on heap.
 */
char **strRingbuffer_destroy(ringbufferType *strRingBuf);

/**
 * @brief Enqueue a C-String.
 *        The C-String is truncated if it is longer than the max
 *        possible strLength.
 *
 * @param strRingBuf Reference to the ring buffer structure
 * @param data A reference to a valid C-String inserted into the ring buffer.
 * @return Status code -- see ringbuffer_StatusType
 */
ringbuffer_StatusType strRingbuffer_enqueue(
    ringbufferType *strRingBuf,
    const char *str);

/**
 * @brief Dequeue a C-String from the head of the ring buffer
 *
 * @param strRingBuf Reference to the ringbuffer structure
 * @return Reference to the dequeued string or NULL for an empty buffer
 */
char *strRingbuffer_dequeue(ringbufferType *strRingBuf);

/**
 * @brief Return the reference to the C-String at the head
 *        of the ring buffer without dequeuing it.
 *
 * @param strRingBuf Reference to the ring buffer structure
 * @return Data stored in the element at the head of the ring buffer,
 *         or NULL if the ring buffer is empty.
 */
char *strRingbuffer_peek(const ringbufferType *strRingBuf);
```

Copy callback function

Note

To simplify your implementation use C-String functions from the C standard library such as `strncpy()`, `memcpy()`, `strlen()`, etc.

7. Test your implementation by following the instructions:

   - Enqueue the string ring buffer in the superloop.

Solution

This code goes in the superloop and is executed to enqueue new data.

```
strRingbuffer_enqueue(&myRingbuffer_Handler, "This exercise is fun!\n");
```

- Transfer of string ring buffer data from the superloop via UART.

Solution

This function is defined in your main file and called periodically by the super loop. There is no real need for a function, but for further development of your code base it is a good practice to encapsulate it in a function.

```
void transmitBufferedString(UART_HandleTypeDef *uartHandler, ringbufferType
*stringRingbuffer) {
  for (char *message = NULL; (message =
strRingbuffer_dequeue(stringRingbuffer));) {
    HAL_UART_Transmit(uartHandler, (uint8_t *)message, strlen(message),
HAL_MAX_DELAY);
  }
}
```

Question

a. Do you receive the complete message?

b. What do you get when the ring buffer is full?

At the end of this lab, examine the behaviour of filling the ring buffer by IRQs from a basic timer (TIM6, TIM7) that inserts some placeholder messages.

Using ring buffer with interrupt

8. Configure and activate a basic timer based IRQ that fires periodically (frequency $10\,\mathrm{Hz}$).
9. In the timer's ISR, fill the string ring buffer with some placeholder string.
10. Answers to the following questions:

Question

a. Are you receiving both messages?

b. Are both messages complete and uninterrupted?

c. Why do you think they are behaving this way?

d. How could this implementation be improved?