# Circular Buffer uint32 - Implementation Notes

Version: 1.0.1

Date: March 31, 2016

# Table of contents:

## About

This software is a light implementation of a circular buffer for unsigned integers. This implementation could be used for instance for systems with low memory.

This document describes the files which are included in this folder. It also describes how an application can use the circular buffer for unsigned integers. Note that this document is not a software design document.

.

## Files

The following files are included in the package:

| File | Description |
|---|---|
| `main.cpp` | The file implements the main C++ function. It implements several test cases. |
| `circular_buffer_uint32.h` | This is file where the circular buffer data structure is defined. The application which uses the implementation of the circular buffer must include this file. |
| `circular_buffer_uint32.cpp` | This is the circular buffer implementation. |
| `config.h` | This is the configuration file. Its content is explained here. |
| `I_circular_buffer_uint32.h` | This is an interface file. |
| `Makefile` | The file which can be used with the `make` utility. |

**Table 1:** Files included in the delivery

## How to compile the deliverables

There is a `Makefile` in the package. One can compile the files running this command line:

```
make all
```

The output executable file after running the command is: `cb_exec`.

To clean the solution run this command:

```
make clean
```

You can also compile the files without using `make` with this command line:

```
g++ -Wall -std=c++11 circular_buffer_uint32.cpp main.cpp -o cb_exec
```

## Using the circular buffer in your application

The source code of the implementation is in the `circular_buffer_uint32.cpp` file. You have to add this file to your project. You also have to add `circular_buffer_uint32.h`, `I_circular_buffer_uint32.h` and `config.h` to your directory which contains the header files. Finally you have to include `circular_buffer_uint32.h` in your source code when defining a circular buffer variable.

Example:

```
// MyApp.cpp

#include “circular_buffer_uint32.h”

char memory[20];
```

```
int main( ) {

circular_buffer_uint32 my_buffer(memory, 20);

my_buffer.push(100);
my_buffer.push(100);
my_buffer.pop( );

return 0;

}
```

## Initialization

There are two ways to initialize the circular buffer.

One way is to pass the address of a memory block and its size in bytes to the constructor. In this scenario the circular buffer contains one physical memory block.

Example:

```
char memory[20];

circular_buffer_uint32 my_buffer(memory, 20);
```

The other way is to pass several blocks of memory to the constructor. The multiple physical memory blocks will be concatenated virtually and will form the working memory of the circular buffer.

Example:

```
char memory1[20];
char memory2[40];
char memory3[80];


mem_block blocks[] = {{memory1, 20},
                      {memory2, 40},
                              {memory3, 80}};

circular_buffer_uint32 my_buffer(blocks, 3);
```

## Configuration

The `config.h` file contains the configuration settings of the circular buffer. **Table 2** explains the settings.

| **Setting** | **Description** |
|---|---|
| `#define NDEBUG` | When commented out the asserts are disabled |
| `const int MAX_BUFFERS = 20;` | The maximum number of physical memory blocks which can be used for the circular buffer. |
| `const bool OVERWRITE = true;` | If overwrite feature is on then the oldest element will be overwritten after a push operation when the buffer is already full. If overwrite feature is off then pushing elements to the buffer will be ignored when the buffer is already full. |

**Table 2:** Configuration settings

## Tips and tricks

To keep the implementation light we have not used error handling. When the initialization of the circular buffer object fails, then the circular buffer is full and empty in the same time. In this case the maximum size of the buffer is zero.

## Known issues and risks

- The implementation does not check whether the memory blocks which make up the circular buffer overlap. The functionality of the buffer is unpredictable in case of overlapping memory blocks

- The counter which keeps the number of elements from the circular buffer is defined as a `size_t` type. If the circular buffer is configured to store more elements than a `size_t` variable can hold then the implementation is unpredictable.

- The implementation requires a C++11 compatible compiler or newer.

- The implementation has been tested on Ubuntu 16.04 64-bit version.

- An implementation based on templates could be easily developed based on the current implementation.