

Documentation of Library KK-Buffer

License and Copyright

Published under MIT License

with Copyright (c) 2025 Kay Kasper

See LICENSE

Introduction

A library for a general buffer implementation, that is based on a template class and supports various value types. A second template type enables 2 maximum capacities for optimized performance.

The GeneralBuffer class is a feature rich implementation of a general usable buffer. For queues and stacks are the subclasses FIFO and LIFO with reduced functionality available.

The general buffer is implemented as a ring buffer, that supports the putting of values by one thread like an interrupt routine and the getting and handling of values by another thread like the loop without real conflicts.

Advantages

- general use for temporary storage
- value manipulation in order or out of order
- access to all values at any time by direct addressing
- template based for various storage types
- template based for 2 different optimized maximum sizes
- best performance up to 250 values capacity (type IND = uint8_t)
- up to 16000 values capacity passible (type IND = uint16_t or int)
- very high performance for most used functions
- small footprint (RAM usage, minimum 8 bytes with IND = uint8_t) for each buffer instance
- works with externaly defined or internally allocated storage
- supports queues (FIFO) and stacks (LIFO)
- supports up to 2 parallel threads (one putting and one getting values)
- inserting, deleting and overwriting of values at any position in the buffer

Installation

The library can be installed via Arduino Library Manager or by downloading the archive from directory „library“ and unpacking the archive in IDEs libraries directory.

Getting started

Only a few lines of code are necessary to understand the logic behind:

```
#include <GeneralBuffer.h>
#include <GeneralBufferExtensions.h>

#define CAPACITY 27
GeneralBuffer<byte> testBuffer(CAPACITY);

void setup() {
    Serial.begin(9600);

    // fill the buffer with characters that shall be printed
    for(byte i=0 ; i<CAPACITY-1 ; i++){
        testBuffer.putLastValue('A' + i, true);
    }
    testBuffer.putLastValue('\n', true);
}

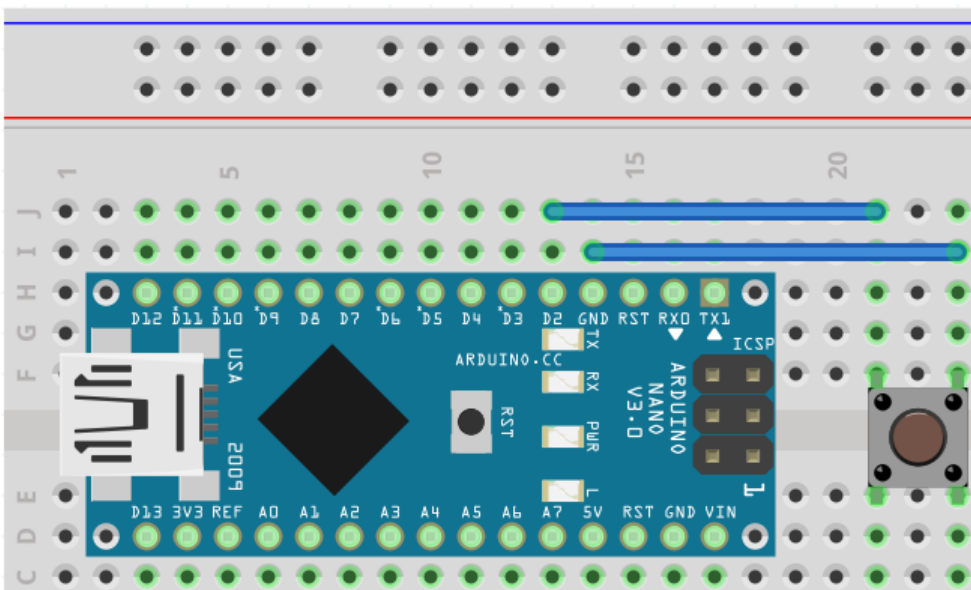
void loop() {
    // do what you want in the loop
    // and write one (the next) character at a loop() call

    if(testBuffer.hasValue()){
        Serial.print((char)testBuffer.getFirstValue());
    }
}
```

See the following examples and more details below.

Examples

Based on a typical easy electrical design (like shown in the following picture with Arduino Nano and a push button)



several examples are included to show the usage and possibilities.

Simple

Does not need a push button to be connected but only Serial output.

```
/*
  Copyright (c) 2025 Kay Kasper
  under the MIT License (MIT)
*/

#include <GeneralBuffer.h>
#include <GeneralBufferExtensions.h>

/*
  Simple example that puts values into the buffer
  in the setup and writes the content in the loop.
*/

#define CAPACITY 27
GeneralBuffer<byte> testBuffer(CAPACITY);
bool printed = false;

// the setup function is called once for initialization
void setup() {
  Serial.begin(9600);

  Serial.print("\r\nFilling the buffer with ");
  Serial.print(CAPACITY);
  Serial.println(" characters including '\n':");

  for(byte i = 0 ; i < CAPACITY - 1 ; i++){
    testBuffer.putLastValue('A' + i, true);
  }
  testBuffer.putLastValue('\n', true);
}

// the loop function runs over and over again forever
void loop() {
  if(!printed){
    Serial.println("Printing buffer content:");
    printed = true;
  }

  if(testBuffer.hasValue()){
    Serial.print((char)testBuffer.getFirstValue());
  }
}
```

SimpleStateChange

Simple example that puts values into the buffer, whenever a change of an digital input at INPUT_PIN was identified. The value is defined to be positive for changes from LOW to HIGH and negative for changes from HIGH to LOW. The buffer content is written in the loop one value per loop() call.

Prerequisites: Serial output and a push button that is connected on one side to GND and on the other side to INPUT_PIN (inverse or negative logic).

Interrupt

Example that puts values into the buffer, whenever a interrupt function is called and a digital input is read.

The value is defined to be positive for HIGH state (button pushed) and negative for LOW state (button released). Due to button bouncing with very fast pin state changes, not always is a positive number followed by a negative number and vice versa.

The values height tells the time difference (in micros) between previous and current call of the interrupt routine. The buffer content is written as output in the loop one value per loop() call.

Prerequisites: Serial output and a push button that is connected on one side to GND and on the other side to INPUT_PIN (inverse or negative logic).

Sort

Example that uses buffers to sort values with different algorithms.

Random values are created in an array and printed. Then they are sorted by 3 algorithms. For comparison the needed microseconds are also written as output. In the loop() function the processing is repeated continuously.

Prerequisites: Serial output

TestBuffer

Example that tests the functionality of the GeneralBuffer class with byte values by putting and getting values in predefined orders to check the results.

Results will be printed out via Serial. If checking doesn't have findings, no output will be printed. Only errors are printed. The test is based on a fixed capacity of 6 values.

At the end also some performance testing is included.

Prerequisites: Serial output

TestByteFIFO

Example that tests the functionality of the FIFO class with byte values by putting and getting values in predefined orders to check the results.

Results will be printed out via Serial. If checking doesn't have findings, no output will be printed. Only errors are printed. The test is based on a fixed capacity of 6 values.

At the end also some performance testing is included.

Prerequisites: Serial output

TestByteLIFO

Example that tests the functionality of the LIFO class with byte values by putting and getting values in predefined orders to check the results.

Results will be printed out via Serial. If checking doesn't have findings, no output will be printed. Only errors are printed. The test is based on a fixed capacity of 6 values.

At the end also some performance testing is included.

Prerequisites: Serial output

TestIntFIFO

Example that tests the functionality of the FIFO class with int values and int internal calculations by putting and getting values in predefined orders to check the results.

Results will be printed out via Serial. If checking doesn't have findings, no output will be printed. Only errors are printed. The test is based on a fixed capacity of 6 values.

At the end also some performance testing is included.

Prerequisites: Serial output

GeneralBuffer

General template based buffer for various types and universal use due to getting and putting values at any position. FIFO and LIFO subclasses available for easy usage. Can help to decouple "parallel" (e.g. interrupt and loop) running activities on a microcontroller.

The buffer is implemented as ring buffer. The storage capacity is limited to max 250 or max 16000 values, depending on the 2. type IND in the variable definition. If 2. type is an 8 Bit type (uint8_t/byte/char) the limit is 250. Other types like int or uint16_t limit it to 16000. Due to the best performance on microcontrollers for all internal calculations uint_t (default) is preferred, if capacity is sufficient. The priority in the implementation was the performance with additionally keeping the code lean and understandable.

The implementation is not thread-safe when real random access by several parallel threads is used. If this is needed, another solution must be chosen. Interrupts are not prohibited in the code. On the other hand it is possible, with some small restrictions, to handle "parallel" activities (of max 2 threads) in microcontrollers with this implementation. If only values are added (by using "putLastValue()") in e.g. interrupt routines and only values are removed (by using "hasValue()" and "getFirstValue()") within e.g. the loop, practically no problems will appear.

Whenever "parallel" activities are writing and reading the same buffer, it must be ensured, that over the time different speeds in the processing will not exceed the capacity. The speed in the processing and the defined capacity of the buffer must be aligned.

The storage for the handled values can be dynamically (based on necessary capacity) generated internally or can be handed over as (outside the class) defined field. In microcontrollers it is better to use outside predefined fields of the same type as the buffer itself (1. type VAL) is defined. The instantiation of a new buffer object with dynamically allocated memory is slower than the other version.

The values are stored as a sequence (without gaps) or queue which has a beginning and an end. Internally two pointers are defining in the ring the first value (the beginning) and the last value (the end). The first value is defined to be at position 1 and the last value is to be defined at position [usedCapacity]. See the next picture:



- used positions, U total number of used positions
- free positions, X+Y total number of free capacity
- total capacity = used capacity + free capacity = U + X + Y

The two template types are:

1. type VAL defines the type of the handled values (mandatory)
2. type IND defines the type of the internal pointers and calculations (optional, default uint8_t)

Parameters

Parameter names	Description
bufferLen	number of elements of the storage field, that can be used by the buffer; values from 2 to 251 (type IND 8 Bit) or 16001 (type IND > 8 Bit) allowed, must be 1 bigger than expected capacity
bufferAddr	pointer to the first value of the storage field, must be pointer to a storage field of type VAL
capacity	number of values, that shall be handled by the buffer
num	number of requested additional or available values in the buffer, should be less or equal to defined capacity
value	the value that shall be stored in the buffer
allowOverwrite	the information, if overwriting is allowed (true) or forbidden (false)
pos	the position of the effected value, range from 1 to usedCapacity or sometimes usedCapacity+1
refValue	reference value for searching / finding
startPos	starting position in the buffer, is first compared value, range 1 to usedCapacity
forward	search direction: true means all positions >= startPos, false means all positions <= startPos
condition	comparison condition as one of the constants GB_COMP_EQUAL, GB_COMP_GREATER_EQUAL, GB_COMP_GREATER, GB_COMP_LESS, GB_COMP_LESS_EQUAL

Constants

Constants for comparison	Description
GB_COMP_EQUAL	stored value == refValue
GB_COMP_GREATER_EQUAL	stored value >= refValue
GB_COMP_GREATER	stored value > refValue
GB_COMP_LESS	stored value < refValue

GB_COMP_LESS_EQUAL

stored value \leq refValue

FIFO

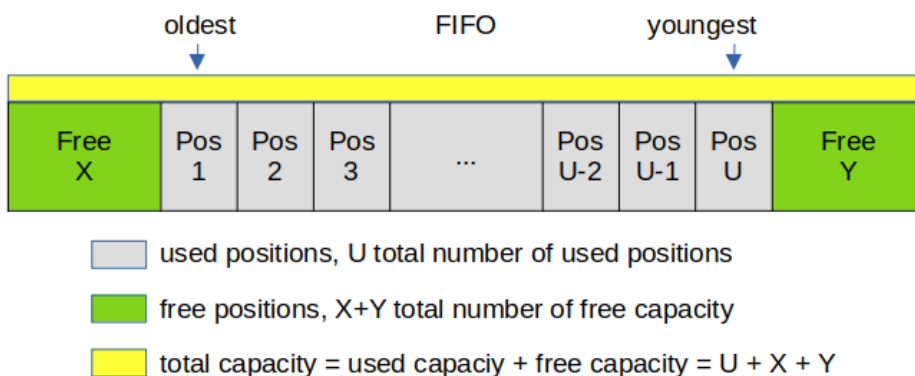
Specific subclass of GeneralBuffer for easier handling of first in and first out buffers or queues. The FIFO class is defined to work only with predefined external storage, which makes the memory handling more predictable and obvious.

Several functions of the base class GeneralBuffer are declared to be privat in the FIFO context, so that they are not allowed to be used here.

Positions are in the same order as in GeneralBuffer. The „getValue()“ function always returns the oldest value.

Parameters, constants and template types are the same as of class GeneralBuffer.

For the understanding of the storage organisation see the following picture:



LIFO

Specific subclass of GeneralBuffer for easier handling of last in and first out buffers or stacks. The LIFO class is defined to work only with predefined external storage, which makes the memory handling more predictable and obvious.

Several functions of the base class GeneralBuffer are declared to be privat in the LIFO context, so that they are not allowed to be used here.

In contrast to the positions of GeneralBuffer and FIFO, the positions are reversed: position 1 is the last added value and position [usedCapacity] is the oldest value in the stack. The „getValue()“ function always returns the youngest value.

Parameters, constants and template types are the same as of class GeneralBuffer.

For the understanding of the storage organisation see the following picture:

