Circular Buffer 1.1

This is the official documentation for the Circular Buffer V1.1 library.
Document version: 1.1 as of 18th May 2022

The full version of this library can be found here:
https://github.com/MHumm/CircularBuffer

Disclaimer: while we try to keep this document updated and correct, we cannot guarantee that the content is 100% error free and/or 100% complete. If you find any issues with it please tell us so we can improve it.

Contents

1 What is Circular Buffer?

Circular buffer is a generics-based library implementing a circular buffer concept, which is a sort of a queue but with a fixed maximum size. This means the buffer size is fixed when it is created. Entries can be overwritten and when deleting items from the buffer the time required is constant as in contrast to a normal array no elements need to be moved around in memory. As the library is written generics-based the data type used for the items stored in the buffer depends on the declaration of the buffer object instance and can either be a simple type like integer, single or Boolean or it can be something managed like a string or object and records can be used as type as well.

The current implementation has one limitation though (feel free to create a pull request changing this – but be sure to make it configurable!): if the buffer is full and further data is requested to be added an exception is raised. Other implementations might just overwrite the oldest element in such a case. This implementation here requires, at least in this version, to remove or delete an item from the buffer first before a new one can be added.

A basic set of Dunit based unit tests is being provided as well to ensure that modifications of Circular Buffer do not break anything. While not covering 100% of all possible test cases it helped us quite a lot during development as they uncovered many failures which we could fix before releasing it.

1.1 Text conventions used in this documentation

Text formatted in *`Courier New and italics`* references method or parameter names, properties, variables and class or unit names of Circular Buffer itself. Text formatted in *italics* but not in Courier New references Delphi RTL types or unit names.

1.2 Revision history of this document and the library

1.0	2021/11/26	Initial revision, published with Circular Buffer V1.0
1.1	2022/05/18	Fixed crashes when calling *FreeObjectsIfOwned* when the buffer got completely filled

2 How to install it?

If not installing via *GetIt Package Manager* (found in the IDE's Tools menu)
Installation is quite easy:

1. Either do a Git clone on the repository or download the ZIP file (see green *Code* button on Github) and unzip it into an empty directory.

2. Add the *Source* folder of Circular Buffer to the search paths of all platforms used. For this open Tools/Options/Language/Delphi/Library, select the platform and add it to that then select the next platform used etc. Close the dialog via *Save*.

3. Open the Ringbuffer project group, select the demo project and run it.

3 How to uninstall it?

If it was installed via GetIt uninstallation needs to be done via GetIt as well.
Otherwise: delete the folder it was installed to and remove all references to the *Source* folder from all platform search paths.

4 Methods

The following methods are provided:

Method	Purpose
`Create`	Constructor. Creates the instance with space for the number of elements given as parameter `Size`.
`Add(Item: T)`	Adds one single item of the type of the ring buffer. Example: `var` `Buf: TRingBuffer<Integer>;` `Begin` `Buf := TRingbuffer<Integer.Create(5);` `Add(23);` This creates an instance with a capacity of 5 elements and adds one integer element with the value 23. Trying to add an element to a completely filled buffer results in an *`EBufferFullException`* exception.
`Add(Items: TRingbufferArray)`	Adds several items of the type of the ring buffer. Such an array can be declared like this (example for an integer ring buffer): *`IntegerArray: TRingbuffer<Integer>.TRingbufferArray;`* Trying to add more elements than space left in the buffer results in an *`EBufferFullException`* and none of the elements will have been added.
`Remove:T`	Returns the first element stored in the ring buffer and removes it

	from the buffer. Trying to remove an element from a completely empty buffer results in an *EBufferEmptyException* exception.
Remove:TRingbufferArray	Returns the first *Count* elements stored in the ring buffer and removes them from the buffer. Trying to remove elements from a completely empty buffer results in an *EBufferEmptyException* exception. Trying to remove more elements than stored in the buffer results in an *EArgumentOutOfRangeException* exception.
Delete	Deletes *Count* elements from the buffer without returning their values. Trying to delete more elements than stored in the buffer results in an *EArgumentOutOfRangeException* exception.
Clear	Deletes all items from the buffer. The buffer is empty afterwards.
Peek:T	Returns the item at the index specified via *Index* without removing it from the buffer. Trying to return an element from a not yet used index position results in an *EArgumentOutOfRangeException* exception. Example: buffer contains 2 elements; index position of the requested element is 5.
Peek: TRingbufferArray	Returns *Count* items at the index specified via *Index* without removing them from the buffer. I either more elements than contained in the buffer are requested or *Index* + *Count* exceeds the upper limit of the buffer, an EArgumentOutOfRangeException exception is raised.

5 Properties

The following properties are provided:

Property	**Purpose**
Size	Capacity of the buffer: number of the elements which can be stored in the buffer.
Count	Number of elements currently stored in the buffer.
Notify	With this an event handler can be assigned which is called every time the buffer content is modified in any way. The event handler must be a method and is called when either *Add, Remove, Delete* or *Clear* is called. The handler can only distinguish between Add and Remove.
OwnsObjects	Boolean property only useful when the items are of a class type. When set to true *Delete* or *Clear* will free the object instances of the items contained in the buffer.

6 Unit Test TestInsight integration

Circular Buffer's DUnit unit test project has integrated support for Stefan Glienke's TestInsight IDE plugin. Using this plugin, the unit tests can be run in background without interrupting one's workflow. Depending on settings they are run in fixed intervals or every time the project is saved. Alternatively, they still can be run manually.

How to set this up?

1. Download the TestInsight installer. It is linked in this wiki page: https://bitbucket.org/sglienke/testinsight/wiki/Home

2. Close the IDE and install it.

3. Start the IDE and open the Circular Buffer project group.

4. Call View/TestInsight Explorer. A window will pop up. This window can be docked somewhere in the IDE. The Object Inspector might be a good place, because it is not used when working in the DUnit test project or use the right half of the messages panel.

5. Activate the DUnit test project.

6. Right click on the DUnit test project and select TestInsight from the context menu. This enables use of TestInsight for running the tests instead of the DUnit GUI.

7. In TestInsight panel either click on the disk button to run the tests every time the project is saved or on the timer button to run the tests in a fixed interval.

8. If the disk button has been selected save the unit test project. TestInsight will create a list of all available tests and run them. By default, tests are listed by status, but it might be more helpful to select the *list by fixture* option, as this resembles the same grouping of tests as shown in DUnit GUI, which might be more familiar and might be easier for finding a particular test.

7 Demos

In order to make your life easier, Ringbuffer ships with a demo application:

- *RingbufferDemo*
  Simplistic console based demo showing basic use of the ring buffer for integers, records and objects.