# Circular Buffer Module

*Instructions on how to use the cbuff_module and its performance characteristics on selected microcontroller platforms*

## Foreward

The Circular Buffer Module is designed to allow the user to simply create circular buffer objects of self designated size for applications where data buffering is needed. The buffer is designed to hold the data type 'unsigned char' which is defined as a new data type 'CBUFF'. This allows for optimal memory usage on most systems and flexibility for the user. The design is generic i.e. it is not targeted at any specific processor architecture or family of microcontrollers, allowing it to be used anywhere. The efficiency of the code and its performance is, therefore, very much dependant on the C compiler used to compile the code. This document serves to explain how to use the module, by providing examples, and the module's performance characteristics on several standard microcontroller architectures.

## Version Number

This document pertains to version v1.0 of the cbuff_module software.

## Change Notificaitons

This is the first full release of the CBUFF Module. There are no change notifications at this time.

## Updates, Support And Feature Requests

The CBUFF Module is supported through Project Kenai (http://kenai.com) under the project name 'cbuff-module'.

A fully tested stable release is always available under the downloads link on that site (http://kenai.com/projects/cbuff-module/downloads).

Requests for bug-fixes or features should be made on the project page under in the „Bugs Registration and Feature Requests“ forum (http://kenai.com/projects/cbuff-module/forums/bugs-and-features).

Limited support is available via the developer at codinghead@gmail.com.

To join the project, please contact me at codinghead@gmail.com.

## Key Software Capabilities

- Supports up to 16 independent circular buffers
- Buffers store 'unsigned char' data type data (typically 8-bit data)
- Buffer size up to 'unsigned int' supported (typically 65536 bytes)
- Fully re-entrant and pre-emptive implementation
- Supports put/get of single bytes or multiple bytes
- Supports clearing of individual buffers
- Remaining space calculation supported for individual buffers
- Fill level calculation supported for individual buffers
- Support for peeking head or tail (returns head/tail value without removing data)
- Support for returning last byte obtained through 'get'
- Support for removing last byte written through 'put'
- All memory allocation handled by the module's user
- All buffers protected from underflow/overflow
- Very small RAM footprint
- Fully portable ANSI C implementation, suitable for embedded microcontroller systems

## File Organisation

The files for this module are organised as described here:

| | |
|---|---|
| circBuffer | – Main directory |
| cbuff_module | – Source code for the circular buffer module |
| cbuffExamples | – Usage examples for various processor architectures |
| cbuffTest | – Buffer module test bench code used during development |
| Doc-O-Matic | – Code documentation generation files for use with Doc-O-Matic Express |
| DOM | – Documentation generated by Doc-O-Matic Express |
| Documentation | – Location of this and other documentation generated by hand |
| License | – Licensing information and other relevant documents |

## Circular Buffer Concept and Important Terminology

A circular buffer is a buffer which uses a First In-First Out (FIFO) approach for the temporary storage of data. The generator of the data, i.e. the data source, is typically termed the **producer**, while the receiver of the data is termed the **consumer**[1].

A circular buffer is usually uni-directional requiring that two circular buffers be used in conjuction with a bi-directional hardware interface such as RS-232. Circular buffers are typically used in applications where:

- An interface generates 'bursty' data, i.e. data comes in multi-byte packets with either regular or irregular spacing between packets, and the receiver of the data can only process the data with a slower, fixed processing rate. Note, however, that the average throughput of the producer cannot exceed the average throughput of the consumer, although the peak throughput may, on occaision, be higher.
- An interface generates data with a protocol with no fixed length. The data cannot be fully evaluated until a complete data frame is received (i.e. GPS data)
- Situations where the data has low priority for the application and processing time will only be made available on an as-needs basis when a defined amount of data has been received

Some microcontrollers and DSPs (such as the dsPIC30/33 from Microchip Technology Inc.) support some form of circular addressing mode in the form of Modulo Addressing, either in the underlying processors architecture or through a hardware module such as DMA. This means that a space in data memory can be defined for use as a circular buffer once a start and end address is supplied. When data is written into the buffer, and once the last address in this block of memory has been used, the hardware ensures that, upon the next address incrementation, the address wraps back to the first address of the defined block.

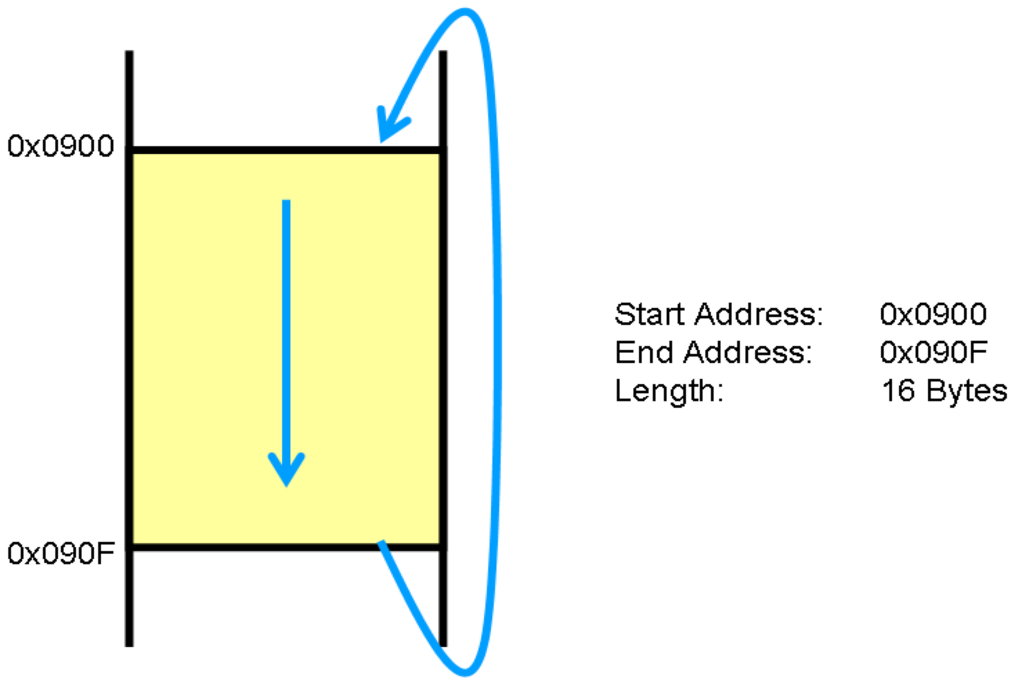

*Figure 1: Implementation of a circular buffer using available modulo addressing support of target hardware*

1 *See also http://en.wikipedia.org/wiki/Producer-consumer_problem*

With this generic circular buffer implementation there is no assumption about the target processer where the module is to be used. Of course on a standard microcontroller or DSP there is only a linear address space available. Thus a software approach is used to firstly define a block of memory as a circular buffer, and secondly to implement the address wrap-around described above. The diagram below helps to describe how the circular buffer is implemented in the linear memory of the processor, and how it then can be conceptulised in its circular format.

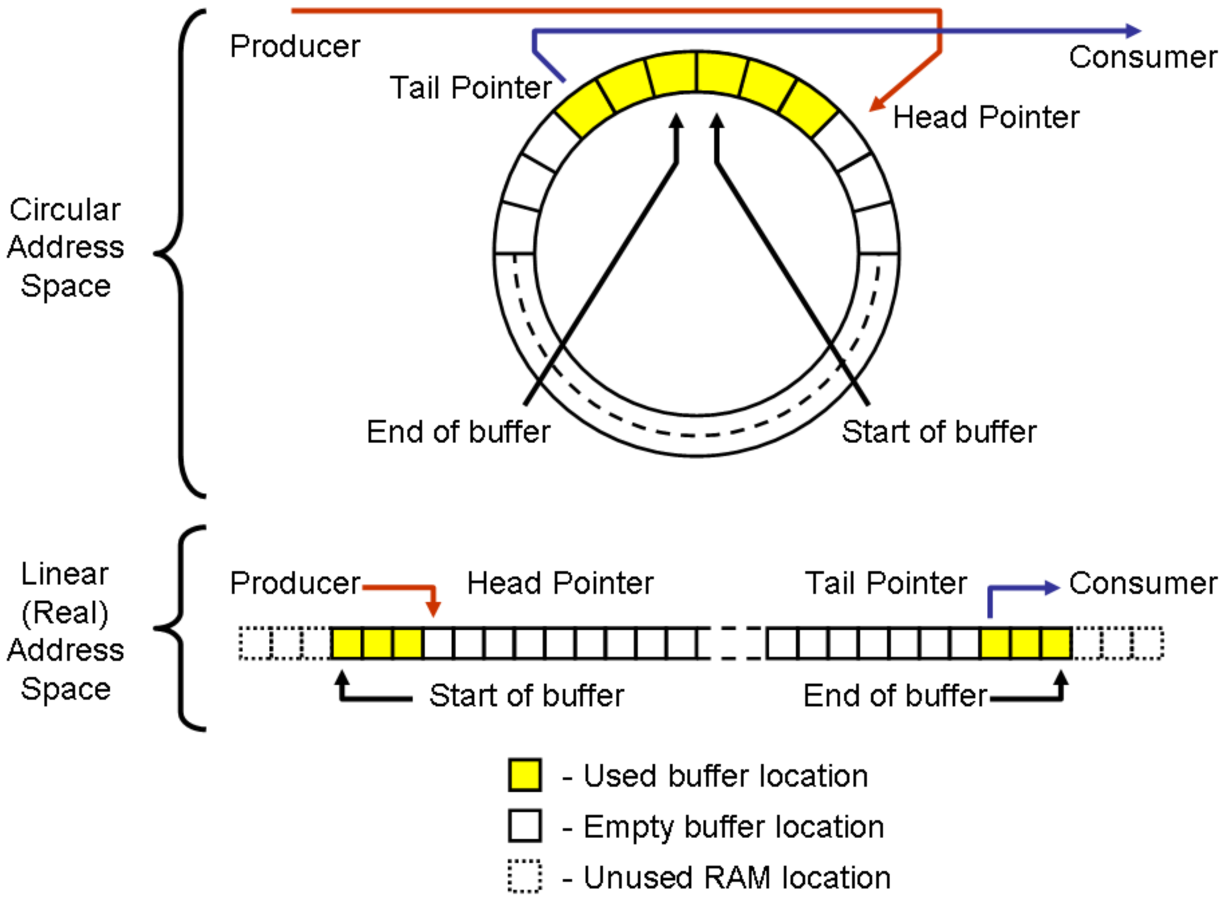

*Figure 2: Visualisation of a circular buffer, and its implementation in the linear address space in the memory of a typical processor*

The important terms here are:

| | |
|---|---|
| • Head | - the next free space in the buffer |
| • Head pointer | - a pointer to this next free space |
| • Tail | - the place where the oldest piece of data in the buffer is stored |
| • Tail pointer | - a pointer to the location where this data is stored |
| • Start of buffer | - address in processor's memory where the reserved space for the circular buffer starts |
| • End of buffer | - address in processor's memory where the reserved space for the circular buffer ends |
| • Producer | - the source generating the data |
| • Consumer | - the target which is evaluating and using the data |

## Files needed to use the CBUFF Module

In order to ensure that the the CBUFF Module is available in your project, you will need to add the CBUFF Module source files to the project where it is to be used. You can either copy the source files to the same directory as that where your application code it to be found, or you can simply reference the necessary files from your project workspace if you are using an IDE for project development (such as MPLAB).

Follow these steps to include the CBUFF Module in your IDE based project:

1. Add the file „cbuff.c“, found in the /cbuff_module directory, to your project workspace (in MPLAB for example)
2. Optional: add the „cbuff.h“ file to your project workspace.
3. Add the following code to the top of all source files using the CBUFF Module's capabilities:

   ```
   #include „cbuff.h“
   ```

## Using the CBUFF Module

In order to create one or more circular buffers using the CBUFF module, please follow the following steps, refering when necessary to the code documentation in the /DOM directory.

For further clarification, use the example code in the /cbuffExamples directory.

1. Declare one (or more) arrays of data type CBUFF and one (or more) variables of data type CBUFFOBJ.
2. Call `cbuffCreate()` one or more times to instantiate one or more CBUFFOBJ buffer objects. Save the return values in individual CBUFFNUM declared variables. The return value is unique to the CBUFFOBJ object you have created. Without it, you won't be able to use `cbuffOpen()` to open the buffer in the next step.
3. Call `cbuffOpen()` for each buffer object when you are ready to use the buffer, referencing the CBUFFNUM variable you were returned from `cbuffCreate()`. Save the return value, which is unique to the buffer object you have opened, in a HCBUFF declared variable. This value will be required for all further CBUFF Module calls.
   **WARNING! :-** In a multi-tasking/multi-threaded system, ensure that `cbuffOpen()` is called inside a **critical section**[2] if more than one thread/task could potentially attempt to open a buffer object with the same buffer number. If the buffer object has not yet been opened there is a risk of a **race condition**[3] occuring whereby two or more threads simultaneously requesting the same buffer object all acquire the chosen resource.
4. With the handle to the buffer object now available you can use the `cbuffGetXXXX()`, `cbuffPutXXXXX()`, `cbuffPeekXXXX()` and `cbuffUnputxxxx()` functions to manipulate the buffer.
5. (Optional) If you wish to release the buffer object for use by another thread or task, call `cbuffClose()` refering to the HCBUFF handle you received when opening the buffer. Be sure to save the CBUFFNUM value this function returns if you wish to remove this buffer object from the list of objects in the modules linked list of buffer objects and free the memory associated with this object.

2 *See also http://en.wikipedia.org/wiki/Critical_section*
3 *See also http://en.wikipedia.org/wiki/Race_condition*

6. (Optional) If you wish to free the memory associated with this buffer object, call `cbuffDestroy()`. If this call is successful, the memory associated with the CBUFFOBJ buffer object and the CBUFF array can be freed.
7. (Optional) If you are finished with the CBUFF Module and have destroyed all available CBUFFOBJ buffer objects, you may de-initialise the CBUFF Module using the `cbuffDeinit()` function.

A general code example is given here. For processor specific examples, please refer to the examples in the cbuffExamples/ directory.

```
#define BUFFER_ONE_SIZE  20
#define BUFFER_TWO_SIZE  40

int main(void)
{
     CBUFF     bufferOne[BUFFER_ONE_SIZE];
     CBUFF     bufferTwo[BUFFER_TWO_SIZE];
     CBUFFOBJ  bufferOneObj;
     CBUFFOBJ  bufferTwoObj;
     CBUFFNUM  bufferNumOne;
     CBUFFNUM  bufferNumTwo;
     HCBUFF    hBufferOne;
     HCBUFF    hBufferTwo;
     CBUFF     dataIn;

     cbuffInit();

     bufferNumOne = cbuffCreate(bufferOne, BUFFER_ONE_SIZE,
                              bufferOneObj);
     bufferNumTwo = cbuffCreate(bufferTwo, BUFFER_TWO_SIZE,
                              bufferTwoObj);

     if (bufferNumOne == 0 ||
         bufferNumTwo == 0)
     {
          /* Handle error; create failed */
     }

     hBufferOne = cbuffOpen(bufferNumOne);
     hBufferTwo = cbuffOpen(bufferNumTwo);

     if (hBufferOne == (void *) 0 ||
         hBufferTwo == (void *) )
     {
          /* Handle error; open failed */
     }

     if (cbuffPutByte(hBufferOne, 'A') != CBUFF_PUT_OK)
     {
          /* Handle error; failed to put data in buffer */
     }
     if (cbuffPutByte(hBufferOne, 'z') != CBUFF_PUT_OK)
     {
          /* Handle error; failed to put data in buffer */
     }

     if (cbuffGetByte(hBufferOne, &dataIn) != CBUFF_GET_OK)
     {
          /* Handle error; failed to get data from buffer */
     }
     if (cbuffGetByte(hBufferTwo, &dataIn) != CBUFF_GET_OK)
     {
          /* Handle error; failed to get data from buffer */
```

```
    }

    bufferOneNum = cbuffClose(hBufferOne);
    bufferTwoNum = cbuffClose(hBufferOne);

    cbuffDestroy(bufferOneNum);
    cbuffDestroy(bufferTwoNum);

    cbuffDeinit();

}
```

*Code Example 1: Generic usage example of the CBUFF Module*

## Further hints on CBUFF Module Usage

When using the CBUFF Module, the following information may be useful to know regarding how certain functions are implemented.

cbuffCreate() & cbuffOpen() - the cbuffCreate() function adds the new buffer object which it is passed to a linked list of buffer objects stored in the CBUFF module. Due to the way in which this function is programmed, each new buffer object is inserted into the start of the linked list for reasons of speed. The disadvantage of this approach is that, when you then call cbuffOpen() to get a handle to your buffer, the buffers you created first take longest to get a handle to. Therefore it is recommended that, in situations where you regularly wish to open and close buffers, the buffers which are open and closed most often are created last. This will ensure that the time to open them is as short as possible.

cbuffDestroy() - This function also suffers from a performance hit when removing buffers from the linked list if the buffers are removed in the order they are created. In order to keep execution time as short as possible, it is recommended to destroy buffers in the reverse order to that in which they were created where possible.

## Memory and Stack Usage for Selected MCUs and DSPs

The following lists some memory usage numbers for some selected MCUs and DSPs. These numbers could affected by the compiler toolsuite version used, so your mileage may vary!

Memory usage for the functions themselves is currently not available.

Stack usage for the functions is currently not available.

### *x86 with GCC Compiler*

Compiler Version: 3.4.5 (mingw-vista special r3)

Compiler Switches: -Wall

Static Memory Usage: 8 bytes (CBUFFOBJ * + CBUFFNUM)

Per Buffer Instance Memory Usage: 32 + buffer_size bytes

Per Buffer Creation Memory Usage: 4 bytes

### *PIC32 with Microchip C32 Compiler*

Compiler Version: v1.11(A)

MPLAB Version: v8.50

Static Memory Usage: 8 bytes (.sbss section of cbuff.o)

Per Buffer Instance Memory Usage: 28 + buffer_size bytes

Per Buffer Creation Memory Usage: 4 bytes

### *dsPIC33 with Microchip C30 Compiler*

Compiler Version: v3.23

MPLAB Version: v8.50

Compiler Switches: -g -Wall

Static Memory Usage: 4 bytes (.nbss section of cbuff.o)

Per Buffer Instance Memory Usage: 16 + buffer_size bytes

Per Buffer Creation Memory Usage: 2 bytes

### *PIC24 with Microchip C30 Compiler*

Compiler Version: v3.23

MPLAB Version: v8.50

Compiler Switches: -g -Wall

Static Memory Usage: 4 bytes (.nbss section of cbuff.o)

Per Buffer Instance Memory Usage: 16 + buffer_size bytes

Per Buffer Creation Memory Usage: 2 bytes

## *PIC18 with Microchip C18 Compiler*

Compiler Version: v3.35

MPLAB Version: v8.50

Compiler Switches: -Ou- -Ot- -Ob- -Op- -Or- -Od- -Opa-

Static Memory Usage: 4 bytes (.udata_cbuff.o)

Per Buffer Instance Memory Usage: 15 + buffer_size bytes

Per Buffer Creation Memory Usage: 2 bytes

## *PIC18 with Hi-Tect PICC-18 Compiler*

Not available.

## *PIC16F1 with Hi-Tech PICC Compiler*

Not available.

## Execution Time Measurements for Selected MCUs and DSPs

There are currently no measurements available for execution time of the CBUFF Module's functions.

## Known Issues

All known issues are documented in „bugzilla“ under Project Kenai.

The link to this information is here - http://kenai.com/bugzilla/buglist.cgi?product=cbuff-module&order=Importance&limit=25

There are no critical or major bugs in this release.

## Future Improvements

In the next release, the following improvements are planned:

- Issue #3520 – Add a 'clear buffer' function that also reset the buffer contents to 0