Ring buffer basics

Ken Wada - August 07, 2013

The ring buffer's first-in first-out data structure is useful tool for transmitting data between asynchronous processes. Here's how to bit bang one in C without C++'s Standard Template Library.

The ring buffer is a circular software queue. This queue has a first-in-first-out (FIFO) data characteristic. These buffers are quite common and are found in many embedded systems. Usually, most developers write these constructs from scratch on an as-needed basis.

The C++ language has the Standard Template Library (STL), which has a very easy-to-use set of class templates. This library enables the developer to create the queue and other lists relatively easily. For the purposes of this article, however, I am assuming that we do not have access to the C++ language.

The ring buffer usually has two indices to the elements within the buffer. The distance between the indices can range from zero (0) to the total number of elements within the buffer. The use of the dual indices means the queue length can shrink to zero, (empty), to the total number of elements, (full). **Figure 1** shows the ring structure of the ring buffer, (FIFO) queue.

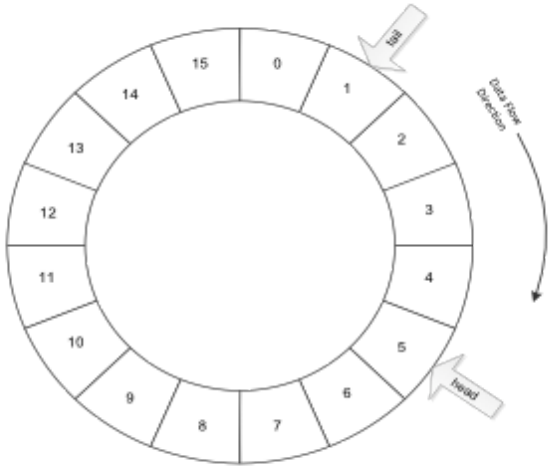

Figure 1: Structure of a ring buffer.

The data gets PUT at the head index, and the data is read from the tail index. In essence, the newest data "grows" from the head index. The oldest data gets retrieved from the tail index. **Figure 2** shows how the head and tail index varies in time using a linear array of elements for the buffer.

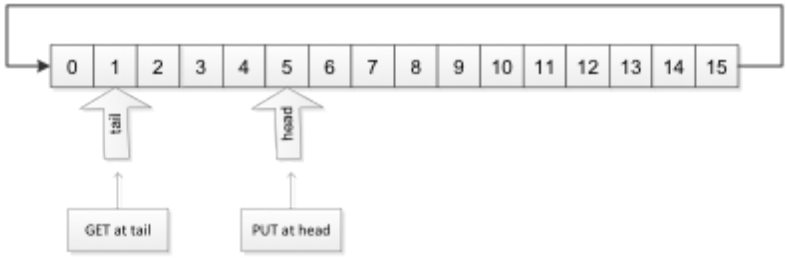

Figure 2: Linear buffer implementation of the ring buffer.

Use cases

Single process to single process

In general, the queue is used to serialize data from one process to another process. The serialization allows some elasticity in time between the processes. In many cases, the queue is used as a data buffer in some hardware interrupt service routine. This buffer will collect the data so that at some later time another process can fetch the data for further processing. This use case is the single process to process buffering case.

This use case is typically found as an interface between some very high priority hardware service buffering data to some lower priority service running in some background loop. This simple buffering use case is shown in **Figure 3**.

Figure 3: A single process to process buffer use case

In many cases, there will be a need for two queues for a single interrupt service. Using multiple queues is quite common for device drivers for serial devices such as RS-232, I2C or USB drivers.

Multiple processes to single process

A little less common is the requirement to serialize many data streams into one receiving streams. These use cases are quite common in multi-threaded operating systems. In this case, there are many client threads requesting some type of serialization from some server or broker thread. The requests or messages are serialized into a single queue which is received by a single process. Figure 4 shows this use case.

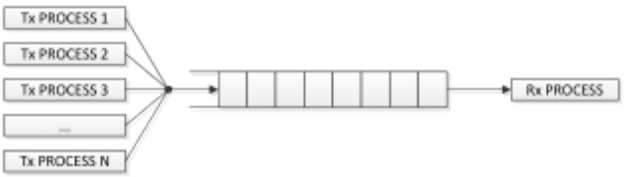

Figure 4: Multiple processes to process use case.

Single process to multiple processes

The least common use case is the single process to multiple processes case. The difficulty here is to determine where to steer the output in real time. Usually, this is done by tagging the data elements in such a way that a broker can steer the data in some meaningful way. **Figure 5** shows the single process to multiple processes use case. Since queues can be readily created, it is usually better to create multiple queues to solve this use case than it would be to use a single queue.

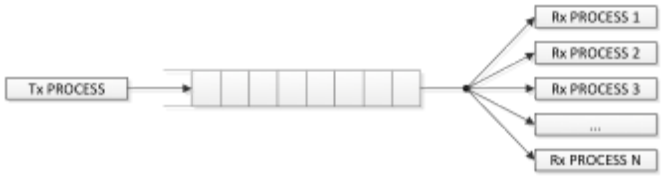

Figure 5: Single process to multiple processes use case.

Figure 6 shows how to reorganize the single process to multiple process use case using a set of cascaded queues. In this case, we have inserted an Rx / Tx Broker Dispatcher service, which will parse the incoming requests to each of the individual process queues.

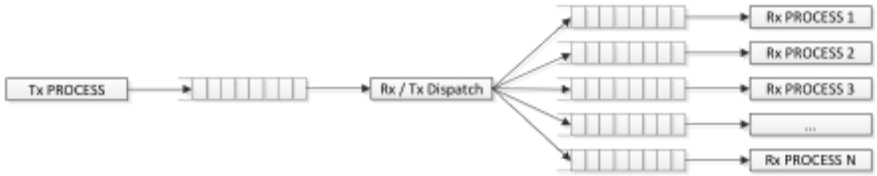

Figure 6: Single process to multiple process use case using a dispatcher and multiple queues.

Overflow

Managing overflow

One must be able to handle the case where the queue is full and there is still incoming data. This case is known as the overflow condition. There are two methods which handle this case. They are to drop the latest data or to overwrite the oldest data. Either style may be implemented. In our case, I will use the drop latest incoming data method.

Design features

In this very specific software queue implementation, I shall use the KISS principle to implement a very simple ring buffer queue. The basic purpose here is to create a queue which can handle a stream of bytes into a fixed buffer for the single process to single process use case. This type of ring buffer is very handy to have for a simple buffered serial device. The design features for this simple queue is as follows:

1. The buffer will contain a fixed number of bytes. This number of bytes

will be set by a macro definition in the header file.

2. The overflow condition will be managed via the drop latest information process. This means in the event of an overflow, the latest incoming data will be dropped.

Given these features leads us to our first listing. Again, the 1st listing (**Listing 1**) is the main ring buffer header file. This file defines all the interfaces and attributes required for our very simple ring buffer implementation.

Listing 1. The interfaces and attributes for a simple ring buffer object.

```
#ifndef __RINGBUFS_H
   #define __RINGBUFS_H
   #define RBUF_SIZE    256

   typedef struct  ringBufS
   {
     unsigned char buf[RBUF_SIZE];
     int head;
     int tail;
     int count;
   } ringBufS;

   #ifdef  __cplusplus
     extern  "C" {
   #endif
     void  ringBufS_init  (ringBufS *_this);
     int   ringBufS_empty (ringBufS *_this);
     int   ringBufS_full  (ringBufS *_this);
     int   ringBufS_get   (ringBufS *_this);
     void  ringBufS_put   (ringBufS *_this, const unsigned char c);
     void  ringBufS_flush (ringBufS *_this, const int clearBuffer);
   #ifdef  __cplusplus
     }
   #endif
#endif
```

The simple ring buffer record

Inspection of **Listing 1** shows very little is required to implement the simple ring buffer. Also, the buffer size is fixed via the RBUF_SIZE macro, which in this case happens to be 256 bytes. The following is a list of what is contained within the ringBufS record.

buf[]

This is the managed buffer. The size of this buffer is set by the RBUF_SIZE macro. In our case, we are managing a 256 byte buffer of unsigned characters.

`head`

This is the head index. The incoming bytes get written to the managed buffer using this index. The arithmetic for this index is in modulo-256. This is because the RBUF_SIZE macro is defined as 256. Of course, if we defined a different value for the buffer size then the modulus arithmetic will change accordingly.

`tail`

This index is used to retrieve the oldest data in the queue. This index also follows the same modulus arithmetic as in the HEAD index case.

`count`

This field is used to keep track of the total number of elements currently in the queue. The maximum value of this field is set by the RBUF_SIZE macro.

The simple ring buffer methods

There are six(6) methods for the simple unsigned character ring buffer. A brief description of these methods is shown in **Table 1**.

Interface	Description
ringBufS_init	Initialize the queue
ringBufS_empty	Determine whether or not the queue is empty
ringBufS_fulll	Determine whether or not the queue is full
ringBufS_get	Get a byte from the queue, (TAIL)
ringBufS_put	Put a byte to the queue , (HEAD)
ringBufS_flush	Flush the queue and optionally clear the buffer bytes to 0

Table 1 The set of ring buffer queue methods

Table 1: The set of ring buffer queue methods.

`ringBufS_init`

Listing 2 shows the implementation of the simple ring buffer initialization process. In this case, we simply clear all of the object memory. This not only flushes the queue. It also clears the buffer too.

Listing 2: initializing the simple ring buffer.

```
#include  <string.h>
#include  "ringBufS.h"

void ringBufS_init (ringBufS *_this)
{
    /*****
      The following clears:
        -> buf
        -> head
        -> tail
        -> count
      and sets head = tail
    ***/
    memset (_this, 0, sizeof (*_this));
}
```

`ringBufS_empty`

Listing 3 shows the implementation for checking the empty status of the queue. In this case we just check to see whether or not the count field is zero.

Listing 3. Testing the queue empty condition.

```
 #include  "ringBufS.h"

int ringBufS_empty (ringBufS *_this)
{
    return (0==_this->count);
}
```

Check the queue full condition

`ringBufS_full`

Of course, we also need to check the queue full condition. **Listing 4** shows that this is simply a check of the count field against the buffer size.

Listing 4. Testing the queue full condition.

```
 #include  "ringBufS.h"

int ringBufS full (ringBufS * this)
```

```
{
    return (_this->count>=RBUF_SIZE);
}
```

ringBufS_get

In **Listing 5** we get a byte from the queue. Notice that the return value in this method is an integer. We use the value of -1 to signal that an attempt was made to retrieve a value from an empty queue.

We introduce the `modulo_inc()` method here. This method encapsulates the modulus arithmetic required for updating the buffer indices.

Listing 5. Getting a byte from the queue.

```
 #include  "modulo.h"
#include  "ringBufS.h"

int ringBufS_get (ringBufS *_this)
{
    int c;
    if (_this->count>0)
    {
      c           = _this->buf[_this->tail];
      _this->tail = modulo_inc (_this->tail, RBUF_SIZE);
      --_this->count;
    }
    else
    {
      c = -1;
    }
    return (c);
}
```

ringBufS_put

Listing 6 shows how we put a byte to the queue. This listing shows the incoming byte is dropped if the queue is full.

Listing 6. Placing the data into the queue.

```
#include  "modulo.h"
#include  "ringBufS.h"

void ringBufS_put (ringBufS *_this, const unsigned char c)
{
    if (_this->count < RBUF_SIZE)
    {
      _this->buf[_this->head] = c;
      _this->head = modulo_inc (_this->head, RBUF_SIZE);
```

```
      ++_this->count;
    }
}
```

ringBufS_flush

Listing 7 shows how to flush the queue. In this case we set the count and the indices to zero. We can optionally clear the buffer to zero (0) if required. In some cases this may be useful for diagnostics purposes.

Listing 7. Flushing the queue.

```
#include  <string.h>
#include  "ringBufS.h"

void ringBufS_flush (ringBufS *_this, const int clearBuffer)
{
  _this->count  = 0;
  _this->head   = 0;
  _this->tail   = 0;
  if (clearBuffer)
  {
    memset (_this->buf, 0, sizeof (_this->buf));
  }
}
```

First the simple ring buffer

I present a very simple fixed size ring buffer first-in-first-out queue. Also, I mentioned some of the use cases for the ring buffer. This simple ring buffer can be used in a wide variety of simple serial device interfaces. In fact, I have used this simple construct many times in several projects.

The next step is to go over this simple case and to analyze the core design pattern within this simple ring buffer implementation. We can extend this simple pattern to give us a construct that has a bit more capability and can be used in more complex systems. We can extend this simple design pattern to accommodate the following:

1. Queuing up different data types
2. Varying the user buffer and user buffer size
3. Reuse the core design pattern in a type-safe way

Extending this simple ring buffer design to a more reusable and extensible design will be a topic for a future article.

To access the code in this article, download the rinBufS.zip file in Embedded.com's source code library.

Ken Wada *is president and owner of Aurium Technologies, an independent product design and consulting firm in California's Silicon Valley. Ken has over 25 years of experience architecting and designing high-tech products and systems, including the FASTRAK vehicle-sensing system for toll roads and bridges. His expertise includes industrial automation, biotechnology, and high-speed optical networks. Ken holds four patents. You may reach him at krwada@cs.com.*

Related posts:

- DebugTip: Circular History Buffer
- Abstract types using C
- Demystifying constructors
- More about C++ classes