CircBuf 1.0

Lennart Yseboodt

August 1, 2007

Contents

1 Introduction

This manual describes the operation of CircBuf, a compact implementation of circular buffers. This circular buffer acts as a FIFO (First In First Out). They are thread-safe and deadlock free.

FIFO's are commonly used to communicate between two separate control flows in a microprocessor. This can be between two processes running under a scheduler, or in a system without an operating system to communicate between interrupt routines and the main routine of the program.

Circbuf allows you to put data elements in a fifo and later retrieve them. You choose yourself how large a single data element is and how many of them there are. You are responsible for allocating the memory and passing the pointer to the initialization routine.

To use the fifo you make a request to read or a request to write. You get a pointer back that is zero if there is no data available or no space available respectively. If you get a valid pointer you can read or write from it. You must ensure not to write or read to much, there is no memory protection.

1.1 FIFO's

An example session with a FIFO is shown in Figure 1. A white block means that element is empty. Blue means it is filled with data. The light blue element is the element that was last written to the FIFO. The light blue element is also the one you will get when you make a read request. First element in, first element out.

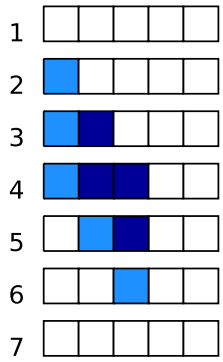

Figure 1: Basic FIFO operation. Light blue is the element that was added last and will be read first

The example starts with all 5 elements empty (white). This is step 1. In step 2 an element is written. In step 3 and 4 two more elements are written. Three places are no filled, the light blue one is the one that was written first. In step 5 a read request is made and the light blue element is read. Now the next element becomes light blue (next to read). In step 6 and 7 the element are read out. Step 7 leaves the FIFO empty.

1.2 Getting started

In the `app` directory are some simple examples that you can take a look at. There is a very simple example and one with threads. For the one with threads you need the pthread and ncurses library, so it will probably only work on unix systems. The other one should work on Windows as well. Now, how to get started ? Say we want a FIFO with 4 byte elements, and we want to have room for 10 of them.

1. We allocate 40 bytes. **char** buffer[40]; or, if you have malloc, **char** *buffer=malloc(40);
2. Next we need the object that represents our FIFO. We declare CircularBuffer cb;
3. The CircularBuffer object must be initialized with the `cb_init()` function. So you call cb_init(&cb,buffer,10,4); The first argument is always a pointer to cb. For the init function the second argument is a pointer to the buffer memory. The third argument is the number of elements your FIFO is deep. The last argument is how large (in bytes) each element is.
4. Now we can use our CircularBuffer. We will put the number 0, 1 and 2 in it.

```
int *d;
```

```
d = cb_writePacket(&cb); /* Get buffer */
*d = 0; /* Write buffer */
cb_doneWritePacket(&cb); /* Don't forget to confirm you are done! */

d = cb_writePacket(&cb); /* Get buffer */
*d = 1; /* Write buffer */
cb_doneWritePacket(&cb); /* Don't forget to confirm you are done! */

d = cb_writePacket(&cb); /* Get buffer */
*d = 2; /* Write buffer */
cb_doneWritePacket(&cb); /* Don't forget to confirm you are done! */
```

5. We can now read out these values again and print them on screen.

```
int *d,c;

for(c=1;c<=3;c++){
        d = cb_readPacket(&cb); /* Get buffer */
        printf("%d\n",*d); /* Print value */
        cb_doneReadPacket(&cb); /* Confirm done */
}
```

You can find this complete example in the `app` directory under the file `numbers.c`.

2 Compiling CircBuf

There is only one source file, called `circbuf.c` that includes two header files (`circbuf.h, types.h`). If your project uses a Makefile you need to add this file in the appropriate place, if you have a graphical environment you can simply add it to your project. This library depends on no other libraries, not even the standard C library.

3 Reference

`cb_init()`

Syntax: **void** cb_init(CircularBuffer *cb, euint8* mem, euint32 numsectors, euint32 sectorsize);

This function must be called before a CircularBuffer object can be used. The first argument (cb) is a pointer to the CircularBuffer object. The second argument (mem) is a pointer to the FIFO memory. This must be allocated by the caller of `cb_init()`. The third argument (numsectors) is the number of slots in the FIFO, and finally the last argument is the size of each slot (sectorsize).

There is no return value.

`cb_readPacket()`

Syntax: euint8* cb_readPacket(CircularBuffer *cb);

This function is used to retrieve an element from the FIFO. The only required argument is a pointer to the CircularBuffer object. This is the case for all functions, and will no be stated anymore unless there are additional arguments The return value is a pointer to the memory containing the FIFO slot. If zero is returned the FIFO has no data available for reading. With a non zero return value you can read out the memory until you call `cb_doneReadPacket` or `cb_cancelReadPacket`. After calling these functions the pointer should no longer be used. Writing in memory obtained by `cb_readPacket` or reading over the slot boundary causes undefined behavior. A failed read request (ie. return value was zero) must not be terminated by `cb_doneReadPacket()` or `cb_cancelReadPacket()` although there is no harm in doing so.

This function will always return zero if a previous call has not ended with a call to `cb_doneReadPacket()` or its canceling variant `cb_cancelReadPacket()`.

`cb_doneReadPacket()`

Syntax: **void** cb_doneReadPacket(CircularBuffer *cb);

This function indicates that you are done reading from the memory retrieved by a previous call to `cb_readPacket()`. After a call to this function one additional slot will be available for writing. If no read was in progress this function has no effect. There is no return value.

`cb_cancelReadPacket()`

Syntax: **void** cb_cancelReadPacket(CircularBuffer *cb);

This function indicates that you are done reading from the memory retrieved by a previous call to `cb_readPacket()`. The memory however is not discarded, a next success full call to `cb_readPacket()` will give the same memory slot. This function was added to allow the creation of non-blocking transfer functions between two FIFO's. Also here there is no return value, and calling this function when no read is in progress has no effect.

`cb_writePacket()`

Syntax: euint8* cb_writePacket(CircularBuffer *cb);

This is the counterpart of `cb_readPacket()`. It makes a request to write to a slot in the FIFO. A non zero return value is a pointer to the available slot. If zero is returned there is no room in the FIFO for writing. Especially with the writing variant special care must be taken no to cross the slot memory boundary otherwise memory in an adjacent slot could be overwritten (or, other data). This call will always return zero if another write was requested and not closed by `cb_doneWritePacket()` or `cb_cancelWritePacket()`. A failed write request (ie. return value was zero) must not be terminated by `cb_doneWritePacket()` or `cb_cancelWritePacket()` although there is no harm in doing so.

cb_doneWritePacket()

Syntax: **void** cb_doneWritePacket(CircularBuffer *cb);

This function indicates that you are done writing to the memory retrieved by a previous call to `cb_writePacket()`. After a call to this function one additional slot will be available for reading. If no write was in progress this function has no effect. There is no return value.

cb_cancelWritePacket()

Syntax: **void** cb_cancelWritePacket(CircularBuffer *cb);

This function indicates that you have not written to memory retrieved by a previous call to `cb_writePacket()`. If you did write to it, you effectively changed the contents of slot that will be returned by the next call to `cb_writePacket()`. A next success full call to `cb_readPacket()` will give the same memory slot. This function was added to allow the creation of non-blocking transfer functions between two FIFO's. Also here there is no return value, and calling this function when no read is in progress has no effect.

cb_lastPacketIsIn()

Syntax: **void** cb_lastPacketIsIn(CircularBuffer *cb);

This function does not manipulate slots in the FIFO. It indicates (from the writing side of the FIFO) that no more packets will be coming. This can be checked by the companion function `cb_moreComing()` on the other side of the FIFO by the reading function.

cb_moreComing()

Syntax: euint8 cb_moreComing(CircularBuffer *cb);

This function checks if more packets will be coming. This flag is set by the `cb_lastPacketIsIn()` function. A non-zero return value means that more packets are expected to be put in the FIFO, a zero return value means that `cb_lastPacketIsIn()` has been called.

cb_lastPacketIsOut()

Syntax: **void** cb_lastPacketIsOut(CircularBuffer *cb);

This function does not manipulate slots in the FIFO. It indicates (from the reading side of the FIFO) that all packets are processed. This can be checked by the companion function `cb_readyReading()` on the other side of the FIFO by the writing function.

cb_readyReading()

Syntax: euint8 cb_readyReading(CircularBuffer *cb);

This function checks if all packets in the FIFO have been processed. This flag is set by the `cb_lastPacketIsOut()` function. A non-zero return value means that all packets are processed, a zero return value means that `cb_lastPacketIsOut()` has been called.

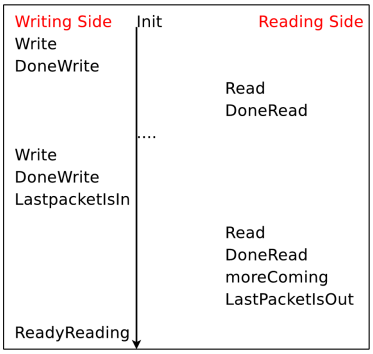

Figure 2: Order of operation in a typical setting