Memory buffers in the KLOE DAQ

H. Beker, M.L.Ferrer, W.Grandegger,
A. Martini, I.Sfiligoi

Table of Contents

Introduction

The KLOE[i] data acquisition system[ii] (DAQ) is designed as a set of independent processes that collaborate to move the data from the front–end electronics (FEE) to the final storage. The main task of these processes is to collect those data from one or more input streams, process and possibly convert them to a more suitable format and write the result in an output stream.

Two mechanisms have been put in place to allow efficient data passing between the processes:

- processes that reside on the same computing node communicate using memory buffers in the common addressing space
- processes that reside on different computing nodes communicate using TCP/IP sockets

The first mechanism is presented in this paper.

Conceptual design

Circular buffers

Memory buffers used in KLOE DAQ are designed as circular buffers[1].

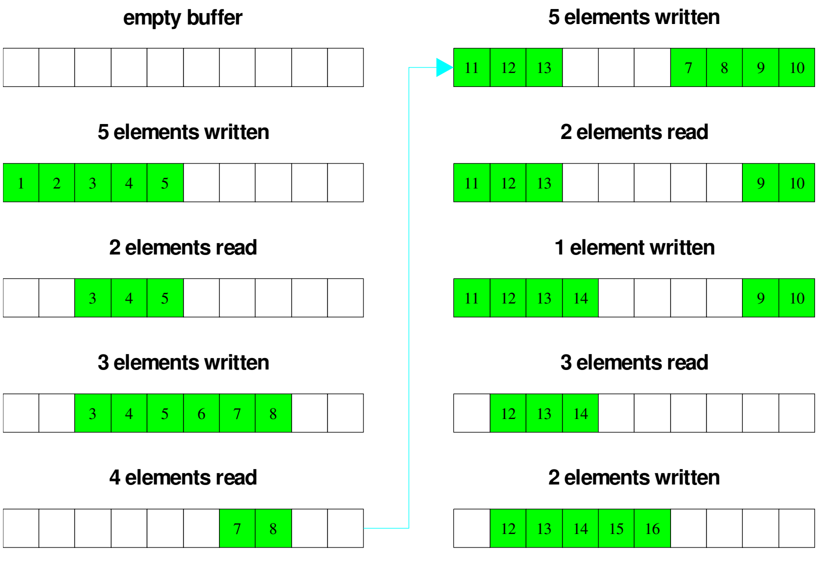

Illustration 1: Circular buffer example

Circular buffers are FIFO[2] queues mapped to fixed sized memory segments.

The memory segment get filled from left to right as long as there is some space available on the right side; once all the available space is used, the process starts again on the left border, given that some space has been freed there in the meantime. The readout works in a similar way; the first element to be read out is the one on the left border, then is the turn of the element on its right and so on, until the rightmost

---

[1] See Illustration 1 for an example of how a circular buffer works.

[2] FIFO – First–In–First–Out policy

element is read out; at that point the process restarts.

Internal data structure

The memory area of a circular buffer can be split in two functional parts; a buffer header and a buffer memory pool. Circular buffers in use in KLOE DAQ are linked lists of elements; buffer headers contain information about the position of the first and the last element of the related list plus some synchronization and monitoring elements, while the memory pool of the buffer contains the actual data and the associated information.

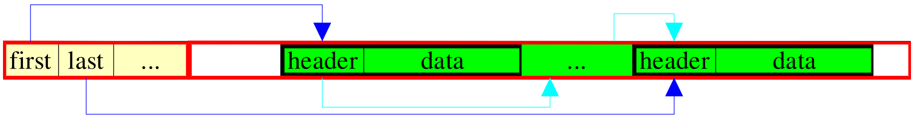

Illustration 2: Circular buffer as a linked list

Elements inside circular buffers are well separated; each element has an identifier and the data size associated to it.

Operations on buffers

The internal structure of circular buffers is not available to the processes; the only way to access the buffers is through a dedicated library. This library contains all the functions needed to access and control one or more circular buffers, maintaining them coherent at all times.

The operations that can be done on a buffer are:

- open/close a buffer
- insert an element in the buffer, eventually asserting that this will be the last element to be inserted
- read an element from the buffer
- look for an element in the buffer, based on its identifier
- get the number of elements in the buffer or other properties of the buffer

To minimize however the overhead of this data structure, the processes are allowed to directly write in and read out the memory buffer; the pointer to the beginning of the related memory area is given by the functions that manage the data structure, but the actual access to the buffer is left to the process command flow. This design choice rises the risk of an accidental corruption of the internal data structure, but the gain of speed and its use in a quite limited application area makes it a more attracting solution over a more safe but expensive buffer copying approach.

Multiple circular buffers

In the above text, different circular buffers were treated as separate entities. Sometimes however it is useful to group several of them together and treat them like a single memory buffer.

The concept of a multiple circular buffer[3] exploit exactly this; several circular buffers are grouped under the same logical identifier. Elements get inserted independently in different buffers, while element extraction is done on unitary basis, i.e. an *element* can be read out from a multiple circular buffer only when a *sub−element* is available on every associated circular buffer.

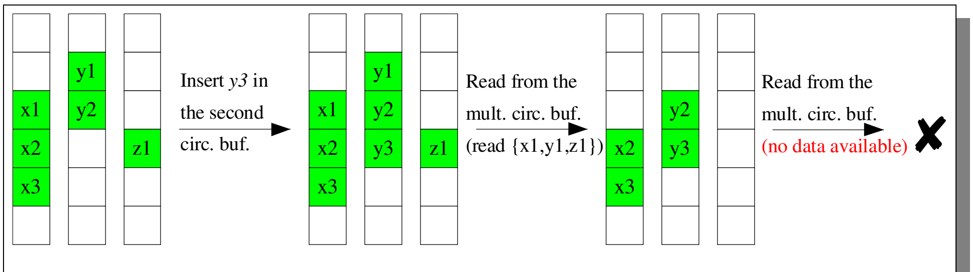

Illustration 3: Multiple circular buffer example

The library used in KLOE DAQ supports the multiple circular buffer paradigm and offers the same operators both for normal circular buffers and multiple circular buffers.

Physical design

Creation and disposal

The memory buffers are implemented using shared memory segments and semaphores, one pair for each memory buffer. The producer[4] creates both the shared memory segment and the semaphore while the consumer[5] or the consumers only map themselves to them. Since several pairs of shared memory segments and semaphores are likely to coexist on the same computing node at any given time, a mechanism based on a file name prefix has been put in place; all the interested processes are given the same text string (the file name prefix) from which they derive the system specific pair of keys. Moreover, in order to dynamically create new buffers, the producer creates the actual files a moment before it creates the shared memory segment and the semaphore.

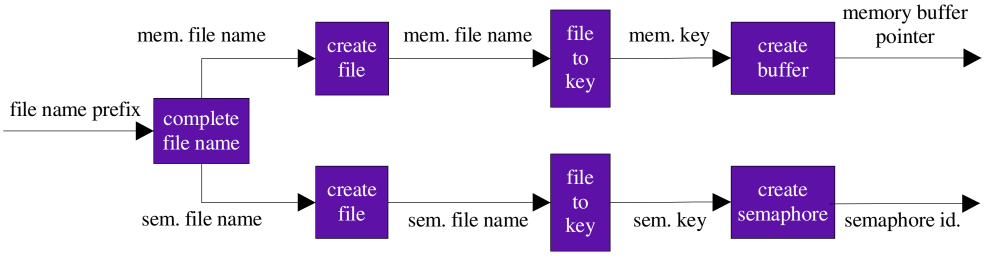

Illustration 4: Producer initialization

The cleanup itself is mainly a producer duty; once the process has finished its task, it takes care of deleting the shared memory segment, the semaphore and the associated files. The consumer cleanup is far simpler; the process must just discontinue the use

---

[3] See Illustration 3 for an example of how a multiple circular buffer works.

[4] **producer** – process that inserts data in a buffer

[5] **consumer** – process that extracts data from a buffer

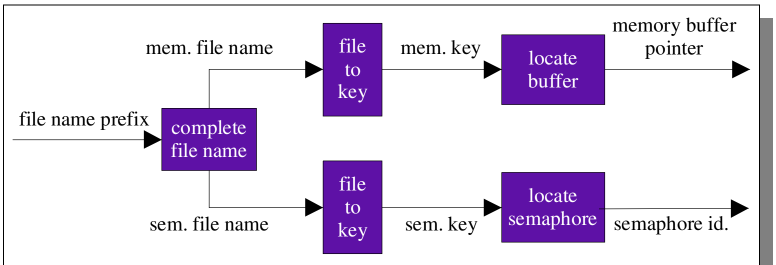

Illustration 5: Consumer initialization

of the semaphore and the related buffer.

Please note that the producer cleanup trashes all the information contained in the buffer. For this reason, the producer should wait for the buffer to be empty, before starting its cleanup procedure.

Data insertion

The data insertion in the circular buffer is a multi−step operation;

1. first of all, a piece of the memory buffer must be reserved by a call to the appropriate library function, specifying the size needed
2. data get written to the given buffer region
3. the element is validated as containing valid data, specifying also the exact size of the data inside (this value can be smaller than the requested size)

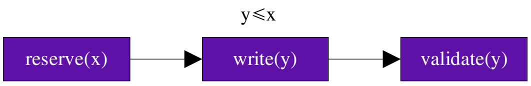

Illustration 6: Element insertion

The above functionality is implemented using a status field for each element. When a request for a piece of buffer is issued, a new element is added to the linked list, given that enough free space is available, the status field is set to **FILLING** and the pointer to the beginning of the data area is passed to the requester[6]. After the element has been validated, the field is changed to **READY** state[7].

To guaranty the consistency of the internal data structure, the associated semaphore is locked during the creation of the new element and during the changing of the status field, limiting the access to the buffer to producer only. The period of time needed for this operation is however very short, while for all the remaining period the buffer is available to all the participants, so the loss in efficiency is negligible.

The above procedure is however the most simple scenario in a data insertion strategy; sometimes, the data get corrupted and so they must not be validated, until it can be make sure they are really OK.

For this reason, the data insertion in a circular buffer at KLOE is transaction based[8]. The transactions are based on data validation; each time an element is written in the buffer, the user has the ability to leave it not validated or validate all the not validated elements in the buffer. If at a certain point the not validated data are discovered to be corrupted, they can be removed from the buffer before a consumer process has read them out. This is similar to the commit/rollback strategy used by several DBMS[iii,iv,v].

---

[6] See Illustration 7.

[7] See Illustration 8.

[8] See also Illustration 9.

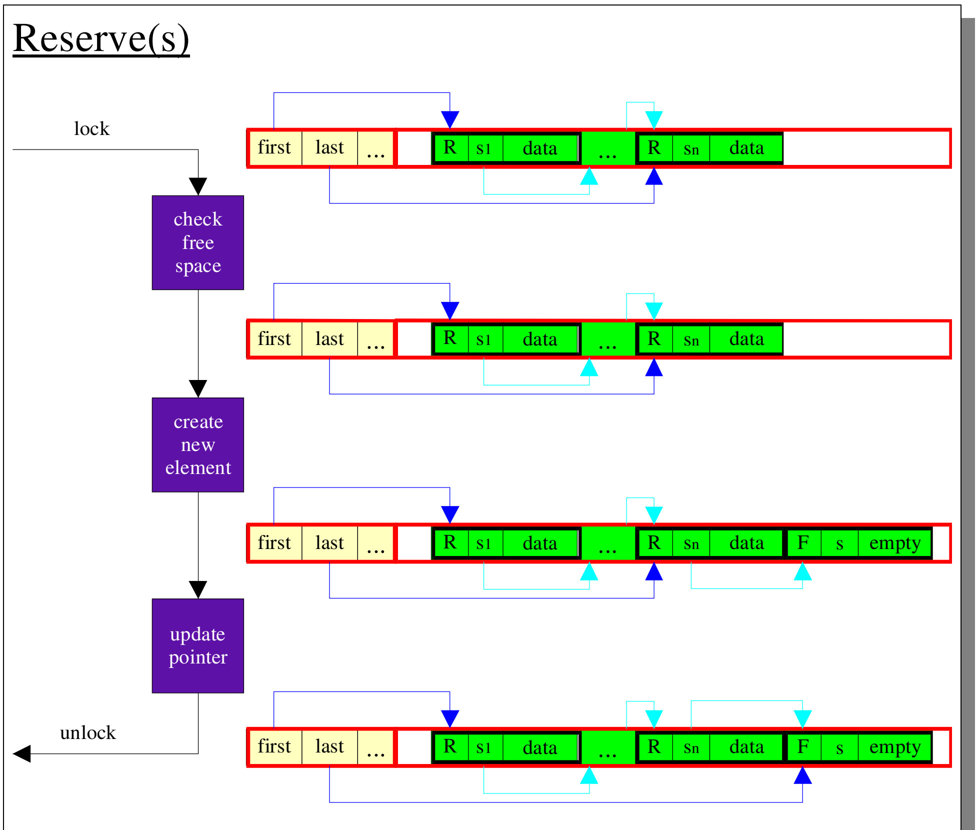

Illustration 7: Reserve a memory buffer

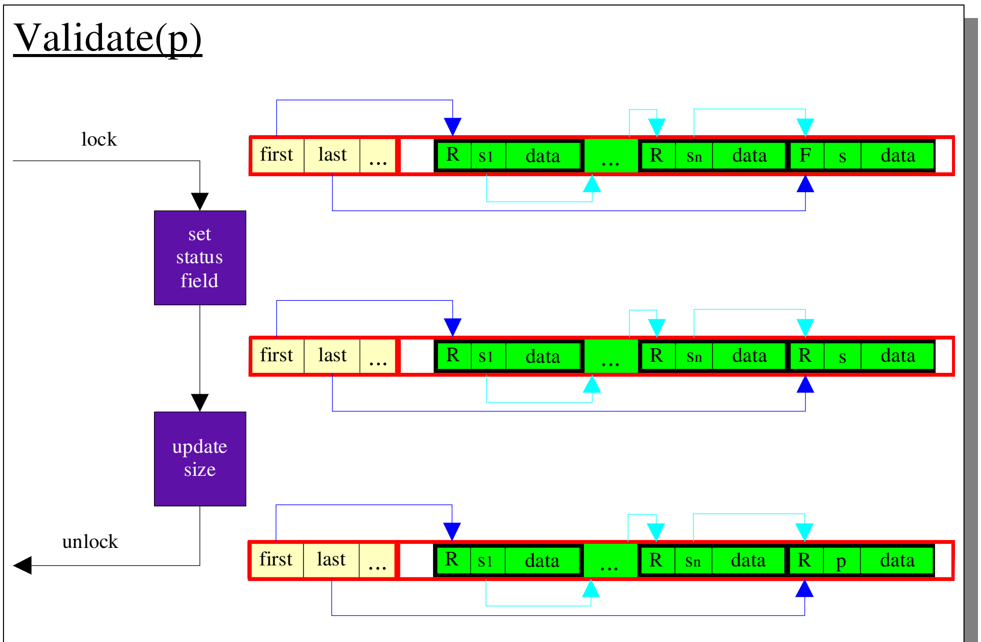

Illustration 8: Validate the buffer

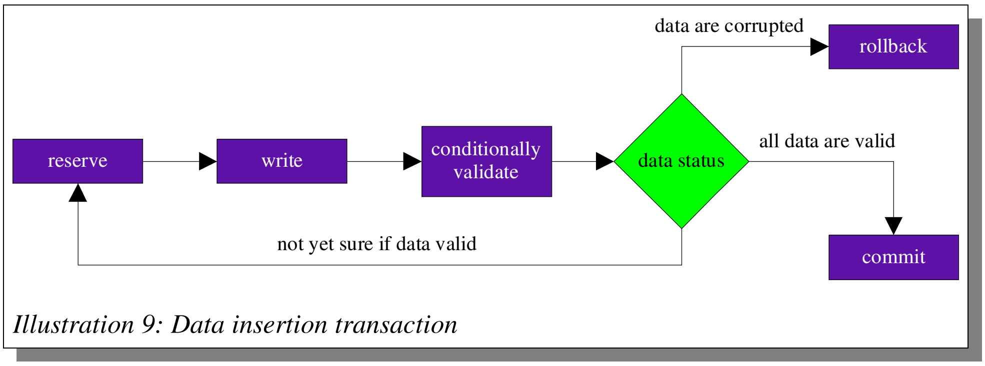

Illustration 9: Data insertion transaction

Data extraction

Like the data insertion, also the data extraction task is a multi–step operation:

1. pointer to the first element in the queue is returned, if one exist and is not in use by another process
2. the data located in the given buffer get used
3. the element is marked as read and removed from the circular buffer

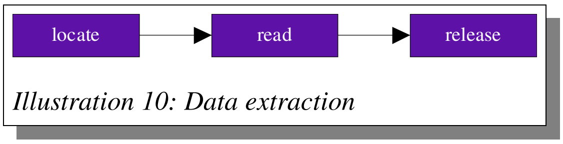

Illustration 10: Data extraction

As in the case of data insertion the status field is used. When an eligible element is found, the status field is set to **EMPTYING** and the pointer to the beginning of the data area is passed to the requester[9]. After the element is not used anymore, it is removed from the linked list[10].

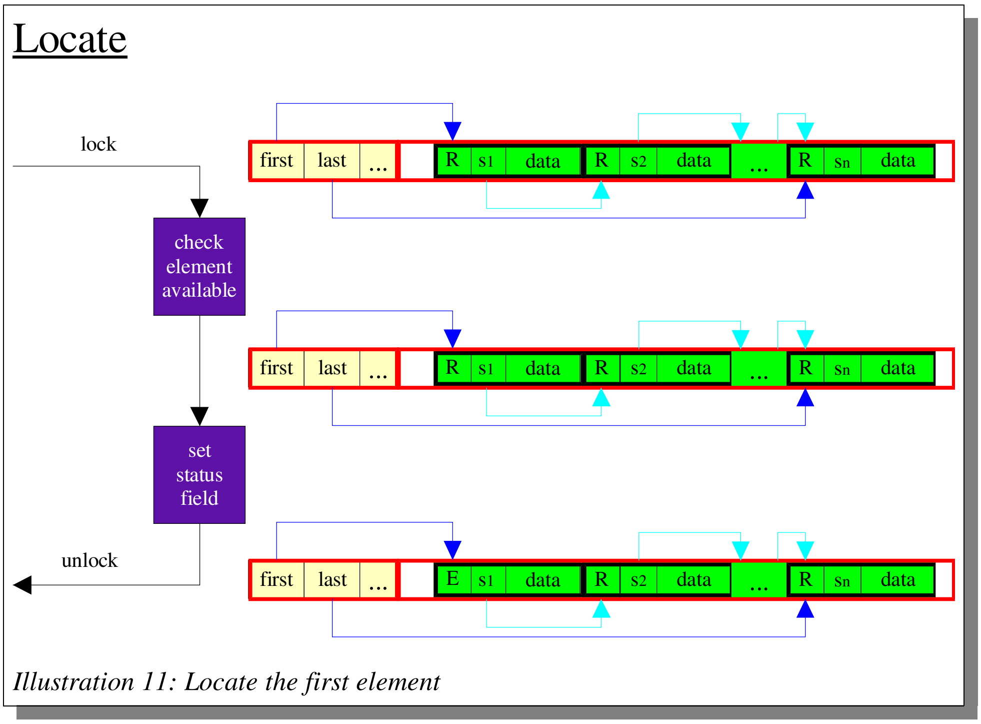

Illustration 11: Locate the first element

To guaranty the consistency of the internal data structure, the associated semaphore is locked during the changing of the status field and during the element removal,

---

[9] See Illustration 11.

[10] See Illustration 12.

limiting the access to the buffer to this consumer only. The period of time needed for this operation is however very short, while for all the remaining period the buffer is available to all the participants, so the loss in efficiency, measured as the fraction of time the buffer cannot be accessed, is negligible. However, since only one element can be the first one in the queue, at any given time, only one consumer at a time can extract data from a circular buffer in KLOE, making parallel in–place data extraction impossible. This limitation can however be easily overcame simply by copying the located element to a private area before using it, if concurrent data extraction from the buffer is needed. In case of a single consumer, instead, the in–place operation is quite faster and have no drawbacks.

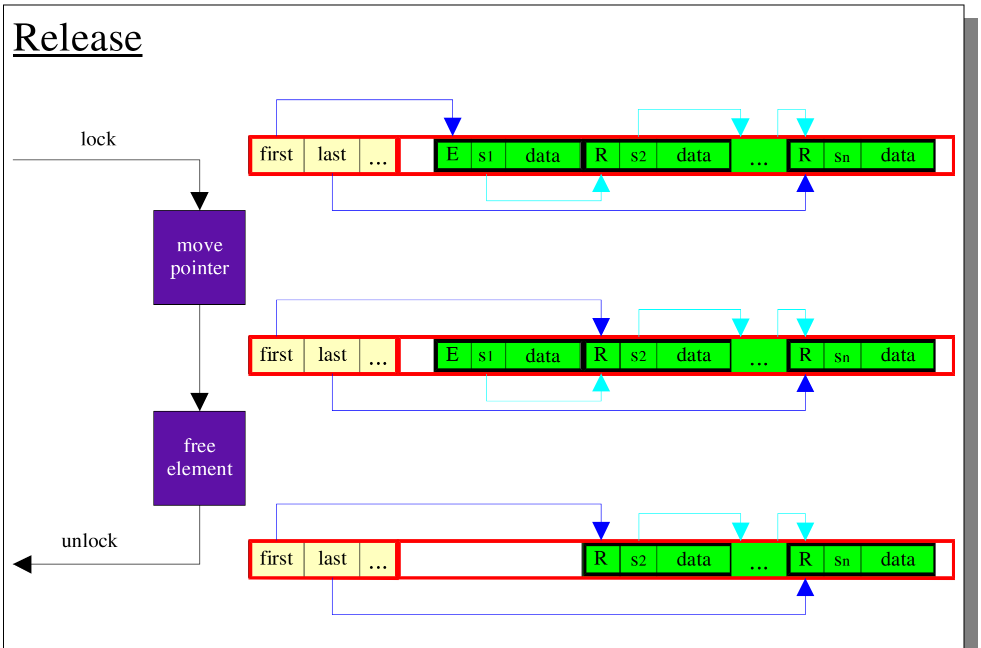

Illustration 12: Release a read element

Data inspection

Some processes, not involved in the data mainstream, are interested to pick some data from a circular buffer, without removing it after the reading. Moreover, some of them are also interested in looking at events that are behind the first element in the queue.

As the previous two tasks, also the data inspection is a multi–part operation:

1. a not yet inspected element is found in the queue, if one exist
2. the data located in the given buffer get copied in a private area
3. the element is released
4. the data in the private area get used

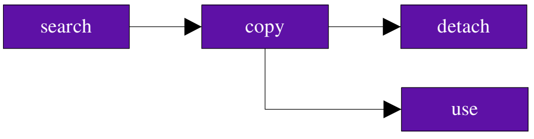

Illustration 13: Data inspection

When an element, eligible for inspection, is found, its status field is set to **COPYING** and is switched back to **READY** only when the copying phase is over.

The copying of data from the circular buffer to a private area is not strictly necessary but is strongly recommended. This requirement is due to the implementation of the circular buffer, that uses the same status field to keep track both the element readout and the element inspection by an arbitrary number of consumers. In order to minimize the dead time of the mainstream consumers, the time in which the inspected element is unavailable must be as short as possible; this suggests the copying to a private area unless the time needed to elaborate the element is shorter than the copying time[11].

Buffer properties

Another functionality of the KLOE memory buffers is the possibility to retrieve some of the buffer properties:

- number of elements ready to be read–out
- size of the memory still available for insertion
- a still–alive flag
- usage statistics (how many calls has been made to the insertion, extraction and inspection routines and how many of them were successful)

The above properties are maintained at all times in the internal data structure, so their retrieval is very fast and can be efficiently used to monitor the status of the memory buffer.

---

[11] There are cases when the elaboration of the content of the element is faster than its copying. This is especially true when only few fields of an element are needed to take a decision.

References:

i *The KLOE Collaboration*,
KLOE, A General Purpose Detector For DAΦNE,
LNF–92/019 (1992)

ii *The KLOE Collaboration*,
The KLOE Data Acquisition System, Addendum To The KLOE Technical Proposal,
LNF–95/014 (1995)

iii *T.Connolly et. al.*,
Database Systems, A Practical Approach to Design, Implementation and Management,
Addison–Wesley (1996),
ISBN 0–201–42277–8

iv **IBM DB2 Universal Database, Application Development Guide**,
IBM Technical Library,
SC09–2845–00

v **IBM DB2 Universal Database, Call Level Interface Guide and Reference**,
IBM Technical Library,
SC09–2843–00