# DYNAMIC MEMORY RE-ALLOCATION FOR SWIFT MODE CHANGES

Okwudire, C.G.U.


## Abstract

In this report, the design and implementation of a scalable buffer component is described. The motivation for such a component comes from the fact that in real-time multimedia systems, there is a need to maintain a desired Quality of Service (QoS) level despite variations in resource availability. The approach considered involves a component-based architecture in which each component may operate in one or more predefined modes and QoS is maintained by system-wide mode changes orchestrated by a Quality Management Component (QMC). A Resource Management Component (RMC) takes the place of a memory management unit and handles all memory related operations. Memory is considered as the constrained resource and a mode change involves relocating memory between components operating in a pipelined fashion with buffers between them.

It is desirable to have buffers which in addition to being able to dynamically resize at runtime, provide some in-buffer processing capabilities to improve performance during a mode change. The necessary interface for realizing such a component was investigated including the trade-offs between efficient memory usage and mode change latency. Finally, the component was implemented in C programming language and would be part of a prototype being developed to illustrate the novel concepts proposed for maintaining QoS in real-time multimedia systems.

# Table of Contents

## 1. Introduction

In embedded real-time systems, there is often the dilemma of maintaining a desired Quality of Service (QoS) level in an environment with constrained resources e.g. processing capabilities, network bandwidth and/or memory. A real-time application consists of a set of tasks where a task can be considered as the work needed for handling an event. These applications are characterized by deadlines which may be hard or soft. Whereas missing a hard deadline usually has catastrophic consequences such as the loss of life or complete failure of the system, missing a soft deadline results in a loss of quality/degradation of performance which in certain cases, may be tolerable as long as the desired QoS is maintained. Thus, the application developer is always faced with problem of a tradeoff between available resources and desired performance.

Being real-time systems, multimedia processing systems are characterized by the above-mentioned properties. In addition, they usually consist of a few but resource intensive tasks, communicating in a pipeline fashion. Thus, in order to maintain a desired QoS level, there is the need for resource management. One way to address the issue of resource management is the reservation-based approach in which each application is assigned a certain amount of processor time, bandwidth and/or memory to perform its tasks with the guarantee that once granted, a reservation is available until the application gives it up.

This piece of work was inspired by an on-going research, within the ITEA2 project CANTATA, on swift mode changes for memory constrained real-time systems (with no memory management unit) that supports Quality of Service [1]. Such a system is assumed to consist of one or more scalable pipelined applications, each having a number of components which can operate in any of their predefined component modes. Thus, QoS is maintained by running components in different modes. In other words, the trade-off between quality and resources (in this case only memory is considered) is achieved by mode changes. A targeted application of this concept is a video content analysis system in the surveillance domain described in [1] and reproduced in Figure 1 below.

In this application, encoded frames are received from the network and decoded after which semantic analysis is performed on the decoded frames to identify the occurrence of certain events (e.g. a robbery). When such an event is detected, the system may change into a different mode, enabling it to transmit the decoder video stream over the network to a portable device. In the context of a memory-constrained system, such a mode change will require reallocating some memory from the buffers along the video processing chain to the buffers between the Network In and Network Out components, and reducing the modes of the video processing components. Such a mode change is initiated by the Quality Management Component (QMC) which is discussed in Section 2.1.

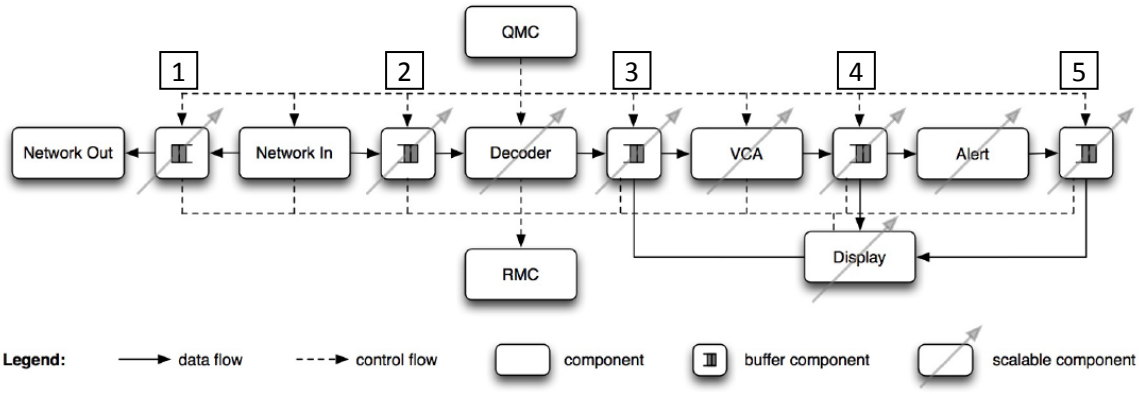



**Figure 1: Quality control flow in a video content analysis system (adapted from Figure 1 in [1])**

Another example application is a scalable video decoding in a television supporting picture-in-picture (PiP) depicted in Figure 2. This application is characterized by frequent mode changes – reallocation of resources (memory) between competing video processing chains e.g. when swapping channels displayed in the main and PiP windows.

As with most (if not all) networked embedded real-time multimedia systems, buffering is required between pipeline stages to accommodate the large gaps between worst-case and average-case resource requirements of the video processing tasks and/or to compensate for the fluctuations in network bandwidth. In our model, buffers are considered as passive components each having an associated memory capacity. From the above examples, it is clear that mode changes are associated with reallocation of memory between components. Therefore, in order to successfully realize a mode change, the buffer components must be scalable.

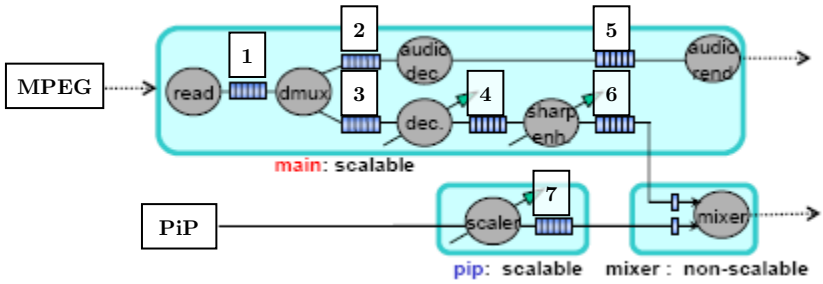



**Figure 2: MPEG decoder with PiP (adapted from RTA.D1-QoS-for-MPTs, 2008/09, slide 32[1])**

The focus of this work was to investigate, design and implement a buffer component which supports swift mode changes i.e. a scalable buffer that can dynamically be resized *at runtime*. Furthermore, it was desired that the buffer should provide interfaces for some in-buffer processing functionality. This requirement followed from the observation that during a mode change it may be desirable to preserve the work already invested in processing the contents of the buffers rather than discarding their contents. Hence, in addition to providing standard FIFO functionality, we required additional interfaces supporting:

1. in-buffer data processing e.g. up-scaling and downscaling of buffer contents in preparation for resizing, or dropping arbitrary frames to avoid stalling of multiplexed streams;
2. dynamic reallocation of buffer resources, i.e. increasing or decreasing buffer size at run time; and
3. relocation of the buffer in physical memory.

## 1.1. Contributions

The main contributions of this work are as follows:

1. We investigated the extra functionality required to support swift mode changes and the interfaces necessary to realize it.
2. We performed an analysis of the design space highlighting the options and their attendant tradeoffs in terms of internal and external fragmentation as well as latency resulting from buffer relocation.
3. We provided a well documented software implementation of the buffer which to the best of our knowledge is the first of its kind in the context of scalable video applications.

## 1.2. Organization of the Document

The rest of this document is organized as follows: Section 2 addresses the design of the buffer component. It begins with an explanation of the concepts central to the idea of swift mode changes. Next to this, an exploration of the design space in presented with emphasis on the trade-offs between the different design options as they relate to fragmentation and buffer relocation latency. The buffer interface is also motivated and explained in this section. In Section 3, the software implementation of the buffer is explained along with a description of the platform on which the prototype of the video content analysis system (where

[1] Information on how to obtain this presentation can be found online at http://www.win.tue.nl/~rbril/education/2IN25/index.htm

this buffer will be validated) is to be built. This is followed in Section 4 by an overview of the file structure of the implementation. Section 5 highlights possible experiments that can be performed when the prototype is built and Section 6 provides a summary of the work and conclusions drawn from it.

## 2. Design

Like every other engineering problem, certain choices had to be made which in one way or another affected the final design of the buffer component as presented in this document. As already stated, the scalable buffer is intended to fit within the context of a larger system. Thus, before going into the analysis, we first present a brief architectural description of the system as well as definition of important terms. A more detailed description can be found in [2].

### 2.1. Architectural Description

In the most general abstraction from details, the proposed system can be considered to consist of three kinds of components, namely the QMC, RMC and scalable components. In other words, every component apart from the QMC and RMC e.g. decoder, network interface, renderer, etc. can be considered as a scalable component. In this abstraction, a non-scalable component is simply a degenerated case of a scalable component which can operate in only one mode.

#### QMC

The QMC is "a quality management component which is responsible for managing the individual components and their modes to provide a system-wide quality of service" [1]. The QMC is also responsible for making budget requests on behalf of other components during mode changes. It is aware of the tradeoff between the desired QoS level and the memory requirements of individual components. Therefore, it is able to manage the modes of other components and change them upon a mode change request.

#### RMC

The RMC is a framework component that functions as a memory allocator and provides guarantees on all granted memory requests. The RMC takes the place of a Memory Management Unit (MMU) and manages the whole memory. Components express their memory requirements in terms of memory reservation requests. The RMC is responsible for memory-related operations such as allocation, monitoring and enforcing and provides the necessary interfaces for requesting and discarding memory reservations.

#### Scalable Components

With the exception of the QMC and RMC every other component is a scalable component (possibly with only one operation mode). These components work together in a pipelined fashion to realize the functionality of the application. They can be viewed as the means by which tasks get their work done. The buffer component functions as storage for intermediate results between pipeline stages.

### 2.2. Definition of Terms

1. *Normal operation*: The state in which the buffer is behaving as a regular FIFO, i.e. data is read in and written out in First-In-First-Out order.
2. *Mode change:* The state during which a mode change is taking place. Extra functionality of the buffer is definitely needed in this state.
3. *Budget:* An amount of memory reserved for a component and guaranteed by the RMC to be available on-demand unless and until it is discarded by the component. A budget is implemented by a *budget head*, which is a data structure containing administrative information and pointing to the *budget tail*, which is the actual space to be occupied (details can be found in [3]).

4. *Allocation:* A memory chunk within the budget of a given component. Like budgets, allocations consist of an *allocation head* and an *allocation tail* defined in a similar fashion, however, bearing in mind that **allocations are always contained within budget tails.**
5. *Memory Allocator:* A component that handles all memory-related operations. In this case, the RMC is the memory allocator and the operations it performs include reserving and discarding budgets, and creating and freeing allocations (upon requests by the components that manage these allocations).
6. *Fragmentation:* According to [4], fragmentation is defined as "the inability to reuse memory that is free". Fragmentation may be internal or external. *Internal fragmentation* is said to occur when a memory chunk that is bigger than the one requested is returned by the memory allocator for some reason(s) related to the allocator's implementation e.g. memory block round-up, memory alignment, etc. *External fragmentation* occurs when there is enough free memory, but there is not a single block large enough to fulfill the memory request. External fragmentation usually results from a combination of allocation policy and the user memory request pattern [5].

In light of the above definitions of internal and external fragmentation, we identify fragmentation at two levels namely (1) with respect to budgets and (2) with respect to allocations. These are defined as follows:

7. *Budget External Fragmentation:* This fragmentation is external with respect to the whole memory space. It occurs when a budget reservation request cannot be granted because there is no contiguous chunk of memory large enough to hold the budget although there is enough free memory. According to [1], this form of fragmentation is minimized by allowing the QMC to make reservations on behalf of components. Solving this problem may entail moving reservations around during a mode change (which is, in general, undesirable). However, the implementation is not described in that document.
8. *Budget Internal Fragmentation:* The unit of memory space in the current implementation is an integer (int) structure. Hence, this form of fragmentation occurs when requested reservation size is not a multiple of `sizeof(int)` which is typically 4 bytes but system-dependent. The resulting fragmentation is bounded by `sizeof(int) - (sizeof(budget) mod sizeof(int))` per budget if it exists and is not so interesting for this analysis.
9. *Allocation External Fragmentation:* This fragmentation is external with respect to the budget encapsulating the allocation. It occurs when the total amount of free space in the budget is enough to grant the requested allocation but there is no single contiguous chunk large enough for the allocation, hence the allocator rejects the request.
10. *Allocation Internal Fragmentation:* Similar to the budget counterpart, it occurs when the size of a requested allocation is not aligned to (i.e. not a multiple of) the unit of memory space which is `sizeof(int)` in our case. If it exists, it is bounded by `sizeof(int) - (sizeof(allocation)mod sizeof(int))` per allocation. Nevertheless, this kind of fragmentation may also result from using fixed-sized allocations. If the data (e.g. a video frame) is variable-sized and the allocation size is chosen as the largest(smallest) frame size, then (parts of) all frames smaller (larger) than the largest (smallest) frame will be (may be) placed in allocations which are larger than necessary, resulting in allocation internal fragmentation.
11. *Defragmentation:* This is the process of reclaiming wasted space resulting from fragmentation. In particular, it is used to refer to the internal algorithm in a component (e.g. buffer) which it uses to make the allocations in its budget(s) contiguous.
12. *Nonalignment:* This refers to a condition in which the size of a chunk of memory requested (whether during a budget reservation or allocation request) is not a multiple of the memory unit in the system. The allocator may reject such a request or return a larger (aligned) chunk than requested thereby creating internal fragmentation. Although the RMC provides a method, `RMC_RoundBytesToWords(),` it allows nonaligned budget/allocation request. Thus, the application developer must be careful to avoid this problem and the attendant fragmentation it creates.

## 2.3. Analysis of Possible Solutions - System Configurations

From the preceding discussion, it is evident that in addition to the buffer component's requirements which were outlined in Section 1, there are certain requirements imposed by the environment in which the buffer component is expected to function. These are outlined below, based on the M.o.S.C.o.W.[2] technique.

### Requirements

1. Minimize the mode change overhead resulting from resizing and/or moving the buffer (**Must**).
2. Maximize dynamic scaling of buffer component (i.e. to minimize fragmentation): Keep the wasted buffer capacity (i.e. non-allocable due to fragmentation within the buffer) and wasted memory budget (i.e. memory which cannot be reserved due to external fragmentation) as low as possible (**Must**).
3. Allow easy integration with a shared memory pool [6] (**Must**).
4. Comply with the architecture description contained in the swift mode changes paper [1] (**Must**).
5. Allow easy implementation of the buffer component (**Could**).

### Goals

The goal of the analysis was to investigate the tradeoffs between the requirements above, particularly (1) and (2) and to provide sufficient information needed to arrive at a choice of a system configuration which will ensure (3) and (4).

### Solution Space

By choosing between variable- and fixed-sized budgets and allocations, we identified four possible design alternatives which we call system configurations. They are:

1. Variable-sized budgets and variable-sized allocations
2. Fixed-sized budgets and variable-sized allocations
3. Variable-sized budgets and fixed-size allocations
4. Fixed-sized budgets and fixed-sized allocations

At this juncture it is necessary to explicitly state the assumptions under which the analysis was performed.

### Initial Assumptions

1. The budget and allocation sizes are both design parameters which can be changed.
2. [1] assumes mutually exclusive access to components. Furthermore, we assume buffers have a single reader and a single writer interface for the sake of simplicity. Lifting this assumption is considered as a subject for future work.
3. For fixed budget configurations, a buffer can request and manage several budgets whereas for variable budget configurations, a buffer requests and manages a single budget whose size is defined by the buffer's capacity (The definition of buffer capacity is rather dependent on the system configuration considered. We provide a definition in Section 2.5). We note that for certain applications such as scalable video it could be desirable to have several variable–sized budgets, each for a separate enhancement layer in which case this assumption will need to be changed.
4. The writer and reader operate on a frame basis i.e. complete frames are written-in and read-out by the writer and reader, respectively.
5. Single frame across multiple allocations is NOT allowed: In the cases where the size of budgets is fixed, it is chosen such that the largest frame can fit in one budget. Allowing frames to spread across multiple

[2] MoSCoW is a prioritization technique used in business analysis and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement - also known as *MoSCoW prioritization* or *MoSCoW analysis*.
Reference: MoSCoW Method. (2009, April 29). In *Wikipedia, The Free Encyclopedia*. Retrieved 09:45, May 6, 2009, from http://en.wikipedia.org/w/index.php?title=MoSCoW_Method&oldid=286758040

budgets introduces more overhead (extra headers) for writing and complicates reading. This follows from assumption 4 above; given a frame-based reader and writer, if the mapping of frames to allocations is not one-to-one, the buffer would be forced to handle the packing and unpacking of frames into allocations. Such a situation is not desirable. Nevertheless, since the budget size is a design parameter, this assumption can be satisfied by properly choosing the budget size.

6. The *defragmentation* procedure reduces allocation external fragmentation to the barest minimum obtainable in the given system configuration (and completely removes it for variable-sized budgets).
7. During a mode change, memory budgets may be moved around under the guidance of the QMC to minimize or completely eliminate budget external fragmentation.
8. In-place resizing of frames is possible during *mode change*.

**Discussion of the Four System Configurations**

Table 1 presents an overview of a comparison of the four system configurations. A more detailed derivation of the findings is contained in the Appendix for the interested reader. The major drawback of Configuration 1 is that it might require moving memory budgets during a mode change. Typically, such budget relocations are costly in terms of time and should be avoided if possible as they might increase the mode change latency bound. Also, budget relocations involve another kind of overhead: When a budget is moved in physical memory, budgets belonging to other components may be affected as well. Hence, the components that own all other affected budgets must be notified and their pointers to existing allocations updated. Such an operation is very unattractive. On the other hand, this configuration results in a more efficient use of memory space than Configuration 2, for instance. Thus, we can conclude that Configuration 1 is best when memory is highly constrained and the overhead resulting from possible budget relocations is tolerable. However, since it is desirable to minimize the mode change latency, this configuration might be less desirable in favor of Configuration 2. Nevertheless, Configuration 2 has the huge disadvantage that in the worst case, a mode change might fail due to the situation depicted in Figure A3 (in the Appendix). Configuration 3 seems to offer no clear advantages; rather, it introduces more overheads. Finally, Configuration 4 is ideal when all frames have the same size as it completely removes all forms of fragmentation (except that caused by nonalignment) while, at the same time, completely removing the necessity of budget relocations during a mode change.

Since none of the configurations investigated offered the desired tradeoff between the conflicting requirements (i.e. minimizing fragmentation and minimizing mode change latency), we revisited the assumptions earlier made, relaxed some of them and came up with the configuration which is presented next.

**An Alternative Configuration**

Although the assumption of one-to-one mapping between frames and allocations (Assumption 4) is realistic, it is not necessary. In fact, there is evidence that in some existing multimedia systems, such an assumption does not hold [9]. Thus, we removed this constraint by allowing a many-to-many mapping between frames and allocations. In other words, a single frame in multiple allocations and multiples frames in a single allocation were allowed. Next to this, we assumed a system-wide fixed allocation size. Of course, with these assumptions it is imperative that both reader and writer components must be able to pack frames into multiple allocations and reconstruct them, respectively, when necessary. This preserves the desirable buffer property of not being content-aware.

In order to completely eliminate the need for budget relocations (and thereby reduce the mode change latency), we redefined a budget. Rather than representing a contiguous chunk of memory, a budget was then defined to represent a number of (fixed-size) allocations which may be scattered around the memory. With this definition, the idea of the RMC providing reservation guarantees is reduced to simply ensuring the number of granted allocations is made available to the requesting component on-demand regardless of where they are located in physical memory. It is therefore necessary and sufficient that the RMC has a way

of identifying allocations belonging to a particular budget but not necessary that they are contiguous in memory.

In the event there is no nonalignment, the only kind of fragmentation that needs to be considered allocation internal fragmentation. However, since multiple frames can be contained in a single allocation, even this form of fragmentation does not occur. Nevertheless, it is comes at the expense of extra header overhead. For example, a frame which would have fitted in a single allocation in Configuration 1 with an overhead of `sizeof(allocationHeader)` may now spread across $n$ allocations where $n>1$, with an overhead of $n$`*sizeof(allocationHeader).`However, if the allocation size is chosen to be much larger than the header size, the resulting overhead will not result in so much memory loss. Another important consequence of these assumptions is that defragmentation is no longer necessary, since the allocations do not have to be contiguous.

Finally, we observe that at the time of writing this report, the RMC implementation and documentation are yet to reflect this change. Yet, the implementation of the buffer, discussed in Section 3, assumes that the new assumptions hold in the system.

**Revised Assumptions**

In summary, the current assumptions on the system are as follows:

1. A many-to-many mapping between frames and allocations. This also implies that the assumption that the reader and writer operate on a frame basis (Initial Assumption #4) is no longer necessary.
2. A single writer, single reader system where both write and reader are aware of the mapping of frames to allocations and responsible for their packing and reconstruction.
3. A system-wide fixed allocation size.
4. A variable budget size which is always a multiple of the fixed allocation size and a budget does not necessarily represent a contiguous chunk in physical memory.
5. Budget relocation and/or defragmentation are no longer necessary.

**FIFO or LIFO?**

So far we have implicitly assumed a FIFO to be suitable for realizing the desired functionality of the buffer. We now provide a motivation for this design choice.

A First-In-First-Out (FIFO) queue is a data structure that provides access to data in the order in which it was received. On the other hand, a Last-In-First-Out (LIFO) queue, as the name implies, provides access to data in reverse order.

The applications under consideration both deal with MPEG videos for which the standard requires that an encoder must send frames in decoding order [8]. That is, frames received from the network are assumed to be arranged in the correct order for decoding. For this reason, a FIFO would be a better choice for the scalable buffer. We note that this design decision is orthogonal to the functionality of the buffer; it concerns implementation.

**Comparison of Different System Configurations**

| | | Configuration 1 | Configuration 2 | Configuration 3 | Configuration 4 |
|---|---|---|---|---|---|
| 1 | **Budget size** | Variable | Fixed | Variable | Fixed |
| 2 | **Allocation size** | Variable | Variable | Fixed | Fixed |
| 3 | **Budgets may be moved during mode change** | Yes | No | Yes | No |
| 4 | **Component may manage multiple budgets** | No | Yes | No | Yes |
| 5 | **Associated header overhead** | BH + (AH * # of frames) | (BH * # of budgets) + (AH * # of frames) | BH + (AH * # of frames) | (BH * # of budgets) + (AH * # of frames) |
| 6 | **Budget external fragmentation (after a mode change)** | Minimized by QMC | (# of components – 1) * (budget size – 1) | Minimized by QMC | Not possible |
| 7 | **Reason(s) for budget external fragmentation** | Variable-sized budgets + memory request sequence | Component unable to return budget due to single frame (see Fig. 4) | Variable-sized budgets + memory request sequence | - |
| 8 | **Budget internal fragmentation (after mode change)** | Constant | O(# of budgets) | Constant | O(# of budgets) |
| 9 | **Reason(s) for budget internal fragmentation** | Nonalignment | Nonalignment | Nonalignment | Nonalignment |
| 10 | **Allocation external fragmentation (normal operation)** | 2*(max. frame size – 1) (see Fig. A2) | (max. frame size – 1) * (1 +# of budgets) (see Fig. A3) | Not possible | Not possible |
| 11 | **Reason(s) for allocation external fragmentation** | Variable-sized allocations/memory request sequence | Variable-sized allocations/memory request sequence | - | - |
| 12 | **Allocation internal fragmentation (normal operation)** | O(# of smallest frame) | O(# of smallest frame) | O(# of smallest frame) | O(# of smallest frame) |
| 13 | **Reason(s) for allocation internal fragmentation** | Nonalignment | Nonalignment | Nonalignment/Variable-sized frames in fixed-sized allocations | Nonalignment/Variable-sized frames in fixed-sized allocations |
| 14 | **Remarks** | Involves moving budgets but better memory usage | High budget ext. frag. may impede mode change | No apparent advantage | Not applicable |

**Table 1: Comparison of System Configurations**

## 2.4. Scenarios

In order to determine the interface methods needed to realize the functionality of the scalable buffer, we considered scenarios from the two sample applications introduced in Section 1.

**Scenarios from the MPEG decoder**

Table 2 illustrates possible scenarios in the MPEG application and the buffers in Figure 2 where they may be needed. This list is not exhaustive; it is only used to justify the required functionality thus, the methods described in later sections. The "Methods" column refers to the Message Sequence Diagrams in Section 2.6 that show the relevant methods and how they interact to realize the desired functionality.

| Possible Scenarios | Example Buffer | Methods |
|---|---|---|
| Read in, write out (Standard FIFO) | 1 – 7 | MSD2/MSD3 |
| Reduce buffer size upon mode change<br>Increase buffer size upon mode change<br>Empty entire buffer contents | 1 – 7 | MSD6<br>MSD7<br>MSD8 |
| Drop arbitrary frames to enable buffer resizing | 4 | MSD5 |
| Drop arbitrary video (or audio) frames to avoid stalling | 2, 3 | MSD4 |
| Downscale/upscale frames before rendering | 5, 6, 7 | MSD6/MSD7 |

**Table 2: Scenarios from MPEG application**

**Scenarios from the Surveillance System**

A video content analysis system was presented in Figure 1 and is described in more details in Section 1 of [1]. Table 3 shows possible scenarios from this application.

As seen from Tables 2 and 3, the same methods can be used in both applications to realize the desired functionality.

| Possible Scenarios | Example Buffer | Methods |
|---|---|---|
| Read in, write out (Standard FIFO)<br>Empty entire buffer contents | 1 – 5 | MSD2/MSD3<br>MSD8 |
| Drop arbitrary frames in preparation for a mode change | 2 | MSD5 |
| Downscale/upscale frames to be displayed | 3, 4, 5 | MSD6/MSD7 |

**Table 3: Scenarios from Surveillance System**

The functionality of the buffer is divided into two interfaces, namely the standard (FIFO) interface and the additional interface necessary to realize the operations captured in the scenarios above. These are described in the following sections. However, it is necessary to note that the discussion is based on a buffer having one reader port and one writer port. This assumption is made in order to avoid dealing with issues such as synchronization and mutual exclusion that arise when multiple ports are used. It can be removed in the future (future work).

## 2.5. The Buffer Component

The buffer component is presented in Figure 3 below. It has two kinds of interfaces, namely:

1. Required interfaces – interfaces to methods which are implemented in other components and required by the buffer to provide (some of) its functionality.
2. Provided interfaces – interfaces to methods which are implemented in the buffer itself and accessible to other components that interact with the buffer

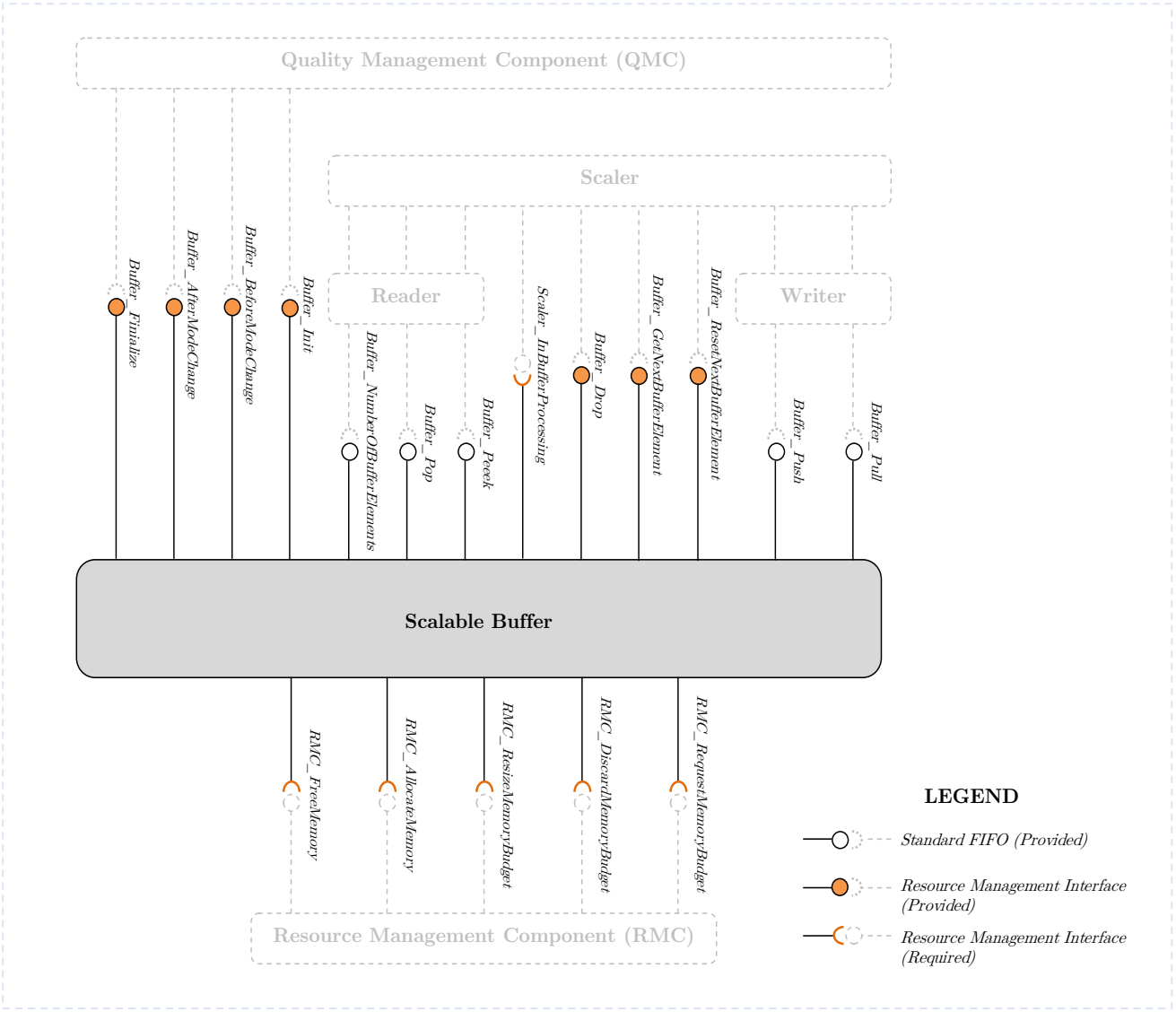



**Figure 3: The Scalable Buffer Component**

Each interface is associated with one or more components that either provide the actual implementation of some functionality (for required interfaces) or use the functionality provided by the buffer (for provided interfaces). Also note that the assumption in [1] on mutual exclusive access to components ensures that at most one of the buffer's interfaces may be active at any given time. The QMC and RMC were introduced in Section 2.1. The rest of the components in Figure 3 are briefly explained below.

*Writer* – a component that writes data into the buffer. Mapping between buffer elements and frames is assumed to be n-to-m where n≥1, m≥1. Packing frames into buffer elements is left as the responsibility of the writer.

*Reader* – a component that reads data from the buffer. The reader is also responsible for re-assembling frames from buffer elements.

*Scaler* – a component that encapsulates the in-buffer processing functionality of up-scaling (increasing the size of data in an allocation), downscaling (reducing the size of data in an allocation) and dropping (including determining which buffer elements should be dropped). The functionality of this component implies that it must have "knowledge" of the contents of buffer elements. However, it is the responsibility of the buffer to furnish this component with pointers to data elements contained in the buffer. The functionality provided by the scaler can, in reality, be handled by separate components. In this sense, the

scaler is only a conceptual component. The relevant interfaces with the buffer are to be designed such that it does not matter whether the up-scaling, downscaling and dropping are done by the same component or by different components.

In Figure 3, the dashed lines between the scaler and the reader/writer are used to indicate that the scaler reuses the interfaces provided by the buffer to the reader and writer in order to realize its functionality (of up-scaling and downscaling; see MSD6 and MSD7).

Also, in component-based design, it is often desirable that the interfaces between any two components should be all of the same kind, required or provided. This allows for independent development and updating of components. In terms of implementation, it also enables a linear include structure in a high level programming language such as C or C++. But, we observe that the interface between the scaler and buffer violates this principle. In particular, the `Scaler_InBufferProcessing()` method is the only required interface while all the others are provided interfaces. This interface is needed to trigger the scaler when its functionality is required. In implementation, the problem was solved by storing a pointer to this function in the buffer header. Hence, the call is "hidden" inside the `Buffer_BeforeModeChange()`and `Buffer_AfterModeChange()`methods.

**Buffer Element**

A buffer element is simply the memory space contained inside the tail of an allocation. Each buffer element consists of a buffer element header in which administrative information is stored and data space where the actual data is stored. Just like the allocation header is oblivious to the buffer component, the buffer element header is oblivious to the components that interact with the buffer i.e. when these components make request to store or retrieve data, they only gain access to the data part of the buffer element. This approach is akin to that used in the Internet Protocol Suite where each layer appends its own header to the incoming packet, for its own administration.

**Buffer Capacity**

Based on the fixed-sized allocation assumption, buffer capacity can simply be defined as the total amount of data space available in the buffer either in terms of maximum number of allocations in the buffer's budget or in terms of bytes. It is given by the following formula:

*Buffer Capacity (in bytes) = max. # of allocations*[sizeof(allocation tail) – sizeof(buffer element header)]*

Since header overhead is not included, this definition of buffer capacity represents the actual amount of space that can be utilized for storing data (provided all requests are aligned i.e. multiples of the word size otherwise allocation internal fragmentation may arise).

**Buffer Interfaces**

Table 4 and Table 5 give an overview of all the interfaces and their relations to the components mentioned above.

| # | Provided Interface | Required by |
|---|---|---|
| 1.<br>2.<br>3. | `Buffer_NumberOfBufferElements()`<br>`Buffer_Peek()`<br>`Buffer_Pop()` | Reader/Scaler |
| 4.<br>5. | `Buffer_Pull()`<br>`Buffer_Push()` | Writer/Scaler |
| 6.<br>7.<br>8.<br>9. | `Buffer_Init()`<br>`Buffer_BeforeModeChange()`<br>`Buffer_AfterModeChange()`<br>`Buffer_Finalize()` | QMC |

| | | |
|---|---|---|
| 10.<br>11.<br>12. | `Buffer_ResetNextBufferElement()`<br>`Buffer_GetNextBufferElement()`<br>`Buffer_Drop()` | Scaler |

**Table 4: Interfaces Provided by Buffer Component**

| # | Required Interface | Provided by |
|---|---|---|
| 2.<br>3.<br>4.<br>5.<br>6. | `RMC_RequestMemoryBudget()`<br>`RMC_AllocateMemory()`<br>`RMC_FreeMemory()`<br>`RMC_ResizeMemoryAllocation()*`<br>`RMC_DiscardMemoryBudget()` | RMC |
| 7. | `Scaler_InBufferProcessing()` | Scaler |

**Table 5: Interfaces Required by Buffer Component**
*To be added to current RMC implementation

It is important to note that the buffer methods are implemented assuming components which keep polling the buffer until their requests can be granted or rejected e.g. a reader that polls an empty buffer until some data is available in the buffer. In the event that this buffer will be used with components that need to be triggered (i.e. "woken up"), then an interface component for this purpose will be needed. Such an interface component will poll the buffer and trigger the other component when the buffer is ready. Alternatively, a "subclass" of the buffer component can be implemented based on e.g. semaphores, which will avoid polling by blocking until the request can be fulfilled.

Table 6 presents an overview of the functionality of the provided interface of the buffer.

| # | Provided Interface | Function |
|---|---|---|
| 1. | `Buffer_NumberOfBufferElements()` | Reports the number of elements contained in the buffer |
| 2. | `Buffer_Peek()` | Returns a pointer to the head element in the buffer i.e. the first allocation in FIFO order |
| 3. | `Buffer_Pop()` | Removes the head element from the buffer |
| 4. | `Buffer_Pull()` | Receives the size of the data to be stored and returns a pointer to the data space of the buffer element made for the data in the FIFO |
| 5. | `Buffer_Push()` | Receives a pointer to an element in the buffer. It indicates that the data has been successfully written into the allocation and can be read by another component |
| 6. | `Buffer_Init()` | Requests a budget for the buffer and initializes the buffer header parameters |
| 7. | `Buffer_BeforeModeChange()` | Allows the buffer time to do any processing before a mode change e.g. downscaling or arbitrary dropping |
| 8. | `Buffer_AfterModeChange()` | Allows the buffer time to do some post processing after a mode change e.g. upscaling |
| 9. | `Buffer_Finalize()` | Discards the budget requested for the buffer during `Buffer_Init()` |
| 10. | `Buffer_ResetNextBufferElement()` | Causes the buffer to reset its internal pointer which points to the next buffer element with respect to FIFO order such that it points to the head element |
| 11. | `Buffer_GetNextBufferElement()` | Returns a pointer to the next element in the buffer starting from the head element |
| 12. | `Buffer_Drop()` | Deletes the buffer element referenced by its argument, `elementPtr` |

**Table 6: Summary of the Functionality Provided by the Buffer**

**Remarks about Pre- and Post-conditions**

No formal specification is complete without pre- and post-conditions. A useful way to view these is as forming a contract between the object and its client. The pre-conditions define a state of the program which the client guarantees will be true before calling any method, whereas the post-conditions define the state of the program that the object's method will guarantee to create for you when it returns[7].

**Standard FIFO interface**

Invariant(s) for all standard FIFO operations:

1. Total free space must be non-negative i.e. removing elements from an empty buffer is not allowed.
2. Total allocated space must be less than or equal to the buffer capacity i.e. no operation should overwrite existing buffer elements or encroach other components' reservations.
3. Buffer capacity does not change.

Pre-condition(s) for all standard FIFO operations:

1. Buffer component must exist.

NOTE: As much as possible, pre-conditions will be checked i.e. enforced using extra parameters to be included in the buffer's header provided the cost (memory required) is not too much.

- `intBuffer_NumberOfBufferElements()` – Reports the number of elements contained in the buffer i.e. allocations that have been pushedhence, available for reading/processing. Illustrated in MSD3.

  *Post-condition:* Returns 0 if buffer is empty; otherwise returns a positive integer equal to the number of allocations in the buffer

- `T* Buffer_Peek()` – returns a pointer to the head element in the buffer i.e. the first element in FIFO order. Illustrated in MSD3.

  *Post-condition:* Returns `NULL` if buffer is empty, otherwise returns a pointer to the head element in the buffer.

- `boolBuffer_Pop()` – removes the head element from the buffer. See MSD3 for illustration.

  In the RMC, when an allocation is freed, it is viewed as free space and may be subsequently reallocated to new data. To avoid overwriting data between a `Buffer_Pop()` operation and when the output component actually completes reading the data contained in that allocation, the Reader executes a `Buffer_Pop()` only after it has completed reading the data (whose pointer it obtained by `Buffer_Peek()`).

  *Pre-condition:* `Buffer_Pop()` must be preceded by at least one `Buffer_Peek()` with no `Buffer_Pop()` between them.

  *Post-condition:* Returns false if buffer is empty or head buffer element could not be deleted, otherwise deletes the head buffer element, returns true,number of elements is decreased by 1, and free-space/allocated-space is incremented/decremented by the size of the removed allocation.

- `T* Buffer_Pull(int size)` – receives the size of the data to be stored and returns a pointer to the buffer element made for that data. This method allows writing arbitrarily-sized data e.g. frames of different sizes. However, from the discussion in Section 2.3, a fixed buffer element size is assumed. Nevertheless, the **`size`** parameter is retained in the event the RMC changes in the future. In the current implementation based on fixed-size allocations the **`size`** parameter is ignored. See MSD2 for illustration.

Similar to popping, it is necessary to ensure that no attempt is made to read from an allocation before the Writer has completed writing data into it. The procedure `Buffer_Push()` is used to notify the buffer when it is safe to read out data from an allocation i.e. after the input data has been completely written in. Hence, `Buffer_Push()` retains the conventional meaning in a FIFO implementation.

*Pre-condition:* **`size`**>0 && two `Buffer_Pull()` calls cannot occur without at least one `Buffer_Push()`between them.

*Post-condition:* Returns a pointer to a new buffer element of **`size`** unit or **`NULL`** if the buffer elementcould not be created (i.e. buffer does not have enough free space), and free-space/allocated-space is incremented/decremented by the size of the created buffer element.

- `Void Buffer_Push(T* elementPtr)` – receives a pointer to an element in the buffer. It indicates that the data has been successfully written into the buffer element and can be read by another component. Illustrated in MSD2.

  *Pre-condition:* **`elementPtr`** is a pointer previously returned by `Buffer_Pull()`&&`Buffer_Push()` must always be preceded by `Buffer_Pull()`.

  *Post-condition:* Number of elements in the buffer increases by 1.

In order to provide thread safe operations, this buffer component can be made a sub-component of one that provides atomic read and write operations. In other words, this is the most basic case (single reader, single writer) but can be generalized for multiple-readers, multiple-writers scenarios. But even for this most basic case the pull-push mechanism is useful as it guarantees correct operation irrespective of the priorities of the reader and writer in a preemptive system.

**Component Framework and Resource Management Interface**

Invariant(s) for all additional operations:

1. Total free space must be non-negative i.e. removing elements from an empty buffer is not allowed.
2. Total allocated space must be less than or equal to the buffer capacity i.e. no operation should overwrite existing buffer elements or encroach other components' reservations.

Pre-condition(s) for all additional operations:

1. Buffer component must exist

The extra interfaces provided by the buffer (provided interfaces) are described below.

- `Bool Buffer_Init()` - Requests a budget for the buffer from the RMC, and initializes the buffer header parameters. The size of the budget to be requested is determined by the initial mode of the buffer and stored in its Component Mode Table. The buffer header is created during the creation of the buffer by the system (see Section 2.6). Illustrated in MSD 1.

  *Post-condition:* Returns true if buffer is successfully initialized; otherwise returns false.

The buffer header contains parameters needed for the buffer's administration. These are described in the inline documentation of the code (in file **`buffer.h`**).

- `bool Buffer_BeforeModeChange()` – allows the component to do any processing before a mode change e.g. to reduce quality/resolution and thus the size of the data stored in a buffer before shrinking the buffer and reallocating its memory. It is called before the QMC reallocates memory reservations between the components involved in the mode change. The exact operations to be performed are

determined from the old and new modes stored in the buffer's Component Mode Table. See MSD5 – MSD9 for illustration.

*Post-condition:* Returns true if all necessary operations before a mode change are performed successfully; otherwise returns false.

- `bool Buffer_AfterModeChange()` – allows the component to do some post processing after a mode change e.g. to increase the resolution of decoded frames or update any memory references. See MSD5 – MSD9 for illustration.

  *Post-condition:* Returns true if all necessary operations following a mode change are performed successfully; otherwise returns false.

- `bool Buffer_Finalize()` – Discards the budget previously requested for the buffer from the RMC during `Buffer_Init()` (and possibly modified during intermediate mode changes), thereby freeing the memory space occupied by the buffer. See MSD 9 for illustration.

  *Post-condition:* Return true if buffer's budget is successfully discarded; otherwise returns false.

- `void Buffer_ResetNextBufferElement()` – causes the buffer to reset its internal pointer which points to the next buffer element with respect to FIFO order such that it points to the head element. Illustrated in MSD4 and MSD5.

  *Post-condition:* Internal pointer to the next buffer element points to the head element if the buffer is non-empty; otherwise, it is made a **`NULL`** pointer.

- `Const T* Buffer_GetNextBufferElement()` – returns a pointer to the next element in the buffer starting from the head element(this is the same pointer which was returned by `Buffer_Push()` when the data was written into the buffer element).With each subsequent call, an internal pointer is incremented to point to the next element in FIFO order. Illustrated in MSD4 and MSD5.

  *Post-condition:* Returns current value of the next pointer and updates it to point to the next element in FIFO order or **`NULL`** if empty or next pointer previously pointed to the tail element.

- `void Buffer_Drop(const T* elementPtr)`– deletes a buffer element at the location referenced by **`elementPtr`** (this is the same pointer which was returned by `Buffer_Push()` when the data was written into the buffer element, or the pointer returned by `Buffer_GetNextBufferElement()`). See MSD4 and MSD 5 for illustration.

  *Pre-condition:* **`elementPtr`** references an element currently in the buffer

  *Invariant:* Buffer capacity remains unchanged

  *Post-condition:* The element referenced by **`elementPtr`** is removed from buffer, number of elements in the buffer is decremented by 1, and free-space/allocated-space is incremented/decremented by the size of the removed element.

Since the required interface methods are implemented in other components, their implementation details (e.g. pre- and post-conditions) must be provided by those components. Hence, they are not addressed here. For those methods already defined, the buffer is assumed to adhere completely to the interface provided.

## 2.6. Buffer Functionality Captured in Message Sequence Diagrams

The Message Sequence Diagrams (MSD) in this section are expected to help illustrate the way the different buffer methods will work together to realize the desired functionality. It also puts the buffer in the

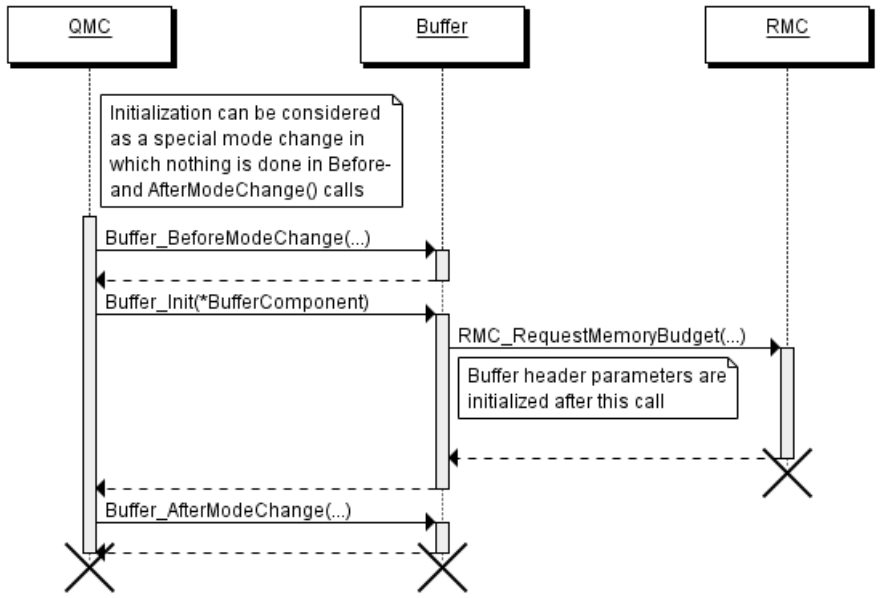
context of mode changes described in Section VI of [1]. However, a few important differences exist in the current perception of a mode change when compared to what was expressed in that paper:

1. When the buffer component is created, a *BufferComponent* structure is associated with it. Before the component is started, the QMC already initializes its Component Mode Table (including the initial mode). Thus, during the initialization phase, the QMC calls `Buffer_Init()` with a pointer to the *BufferComponent* as parameter. In this method, the buffer looks up its initial mode, requests a budget accordingly and initializes the parameters it later uses for its own administration. This approach also removes the cyclic dependency between the buffer and QMC as all interfaces towards the QMC are now provided by the buffer which would not be the case if the `QMC_RegisterComponent()` approach proposed in [1] is used.
2. Instead of discarding and requesting new budgets during a mode change (step 5), a new RMC method is assumed to exist (though not yet implemented at the time of this report) which allows for resizing an existing budget. Depending on the method adopted by the RMC in defining and managing budgets, defragmentation (and update of budget handles) might be required as part of this stage. However, the following MSDs assume that allocations are fixed-sized and a budget is a collection of (possibly) non-contiguous allocation, hence defragmentation is not needed.
3. The method to get (and update) the pointer to an allocation that is used across a mode change is unnecessary since (i) it is assumed that a mode change never occurs between `Buffer_Pull()` and `Buffer_Push()`; and (ii) if new RMC budget concept is adopted, nothing is moved so allocation pointers are never invalidated by a mode change.

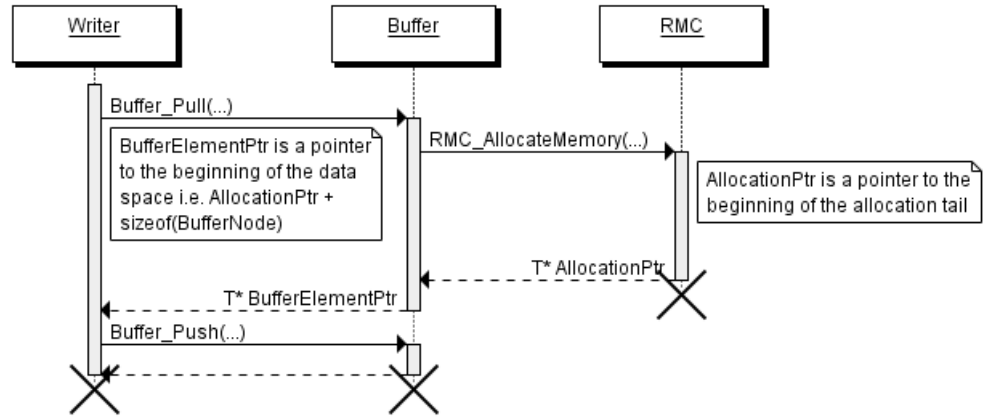
The following are a few notes about the notation used in the sequence diagrams:

- The dashed arrow represents a return of a method call initiated by the corresponding solid arrow. Return values are generally omitted except in cases where they are necessary to illustrate a point.
- `method(…)` – although the method has parameters, we are not interested in them.
- `method(…, param)` – we ignore all other arguments except param.
- `method()` – the method has no arguments.
- Comments are added in certain places for clarification.
- `[whileNextBufferElement != ()]` – loop condition i.e. loop until all elements in buffer have been processed.
- The "×" indicates the end of the particular sequence of messages being illustrated i.e. it does not have the more conventional meaning of thread termination.

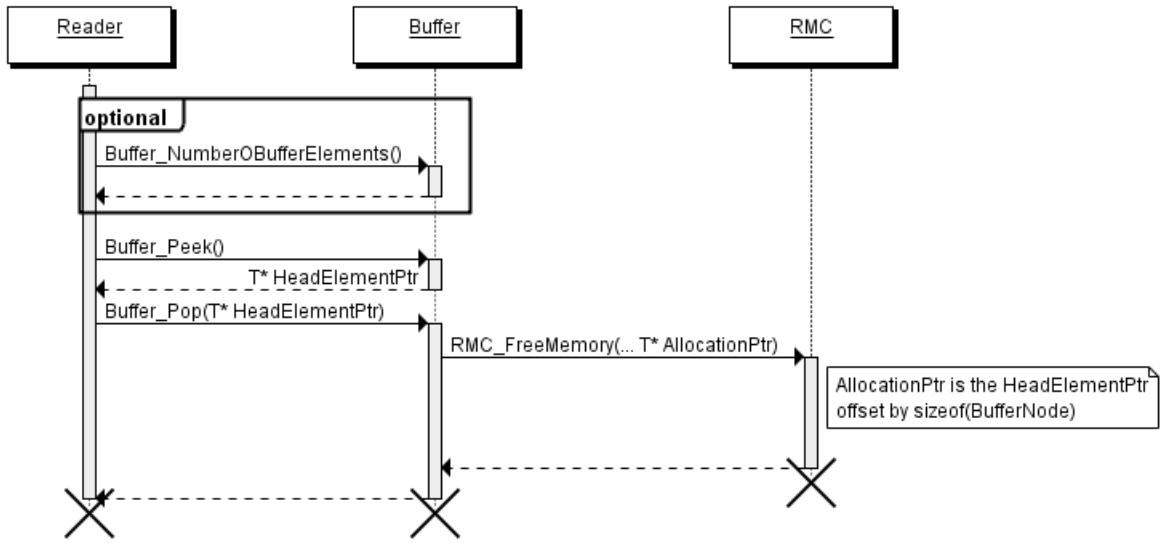
**MSD1: Initialization**




**MSD2: Read-in (Normal Operation)**

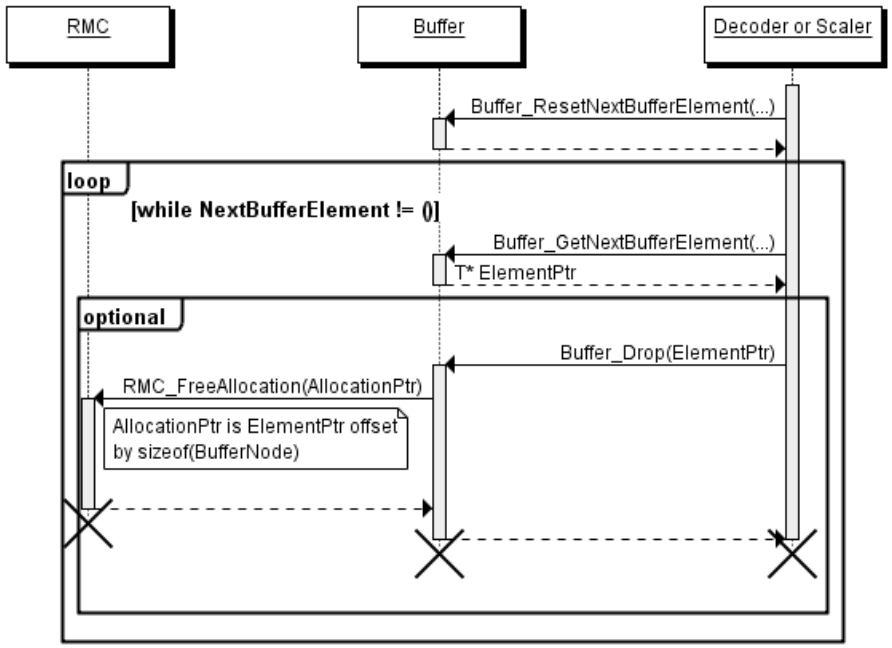



**MSD3: Write-out (Normal Operation)**



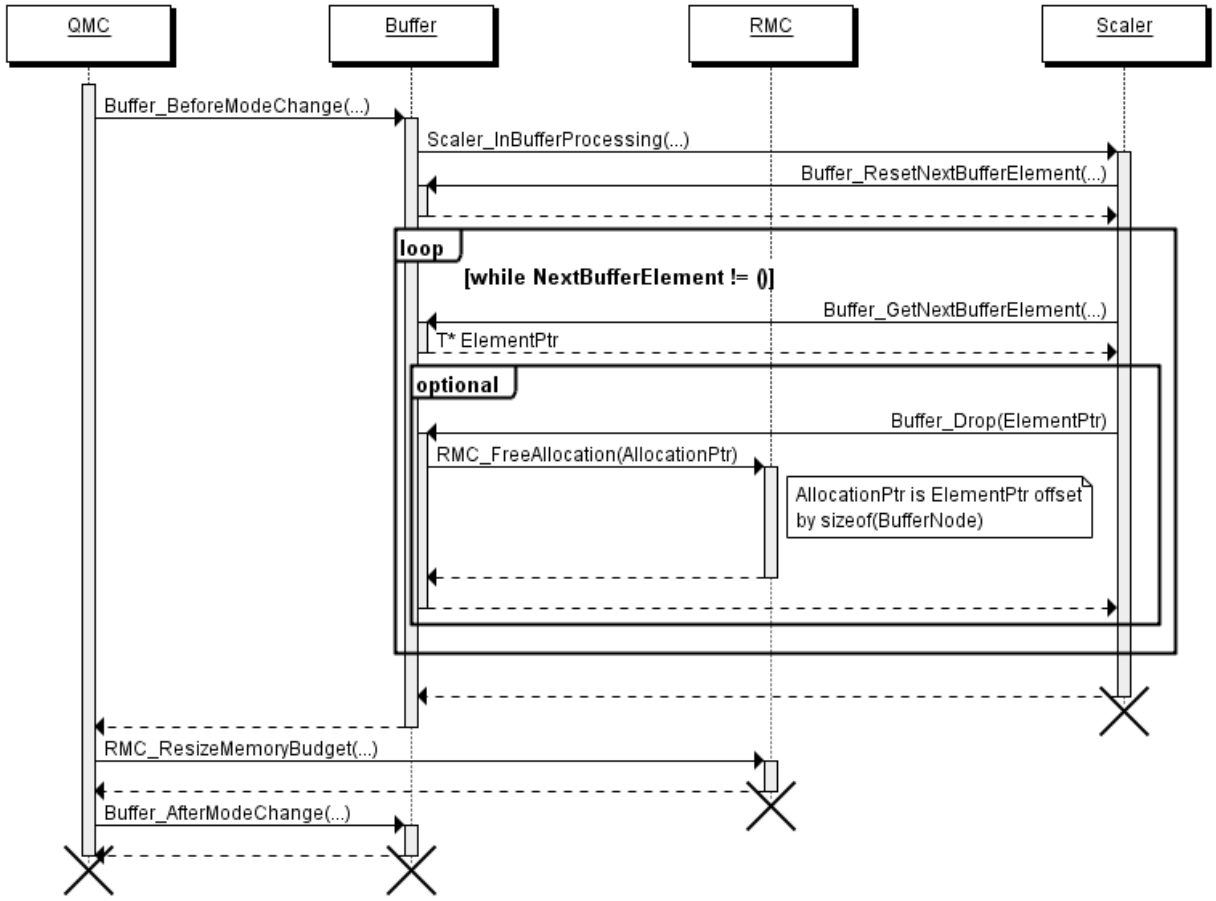
**MSD4: Drop (to avoid stalliing)**

**(Since dropping is performed to avoid stalling and no budget resizing is involved, it is considered a normal mode operation. Hence, QMC is not involved)**

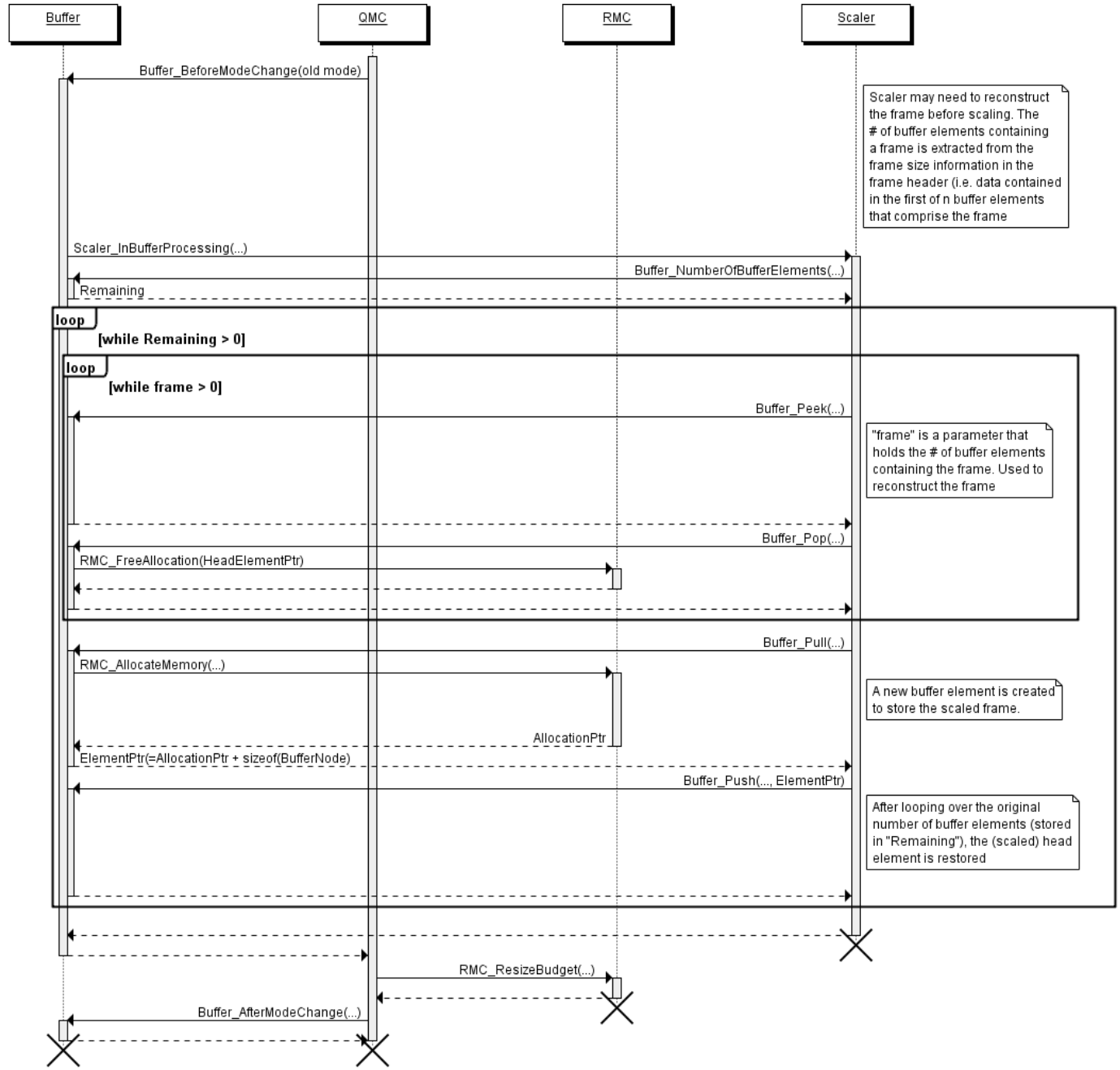



**MSD5: Drop (to enable mode change)**

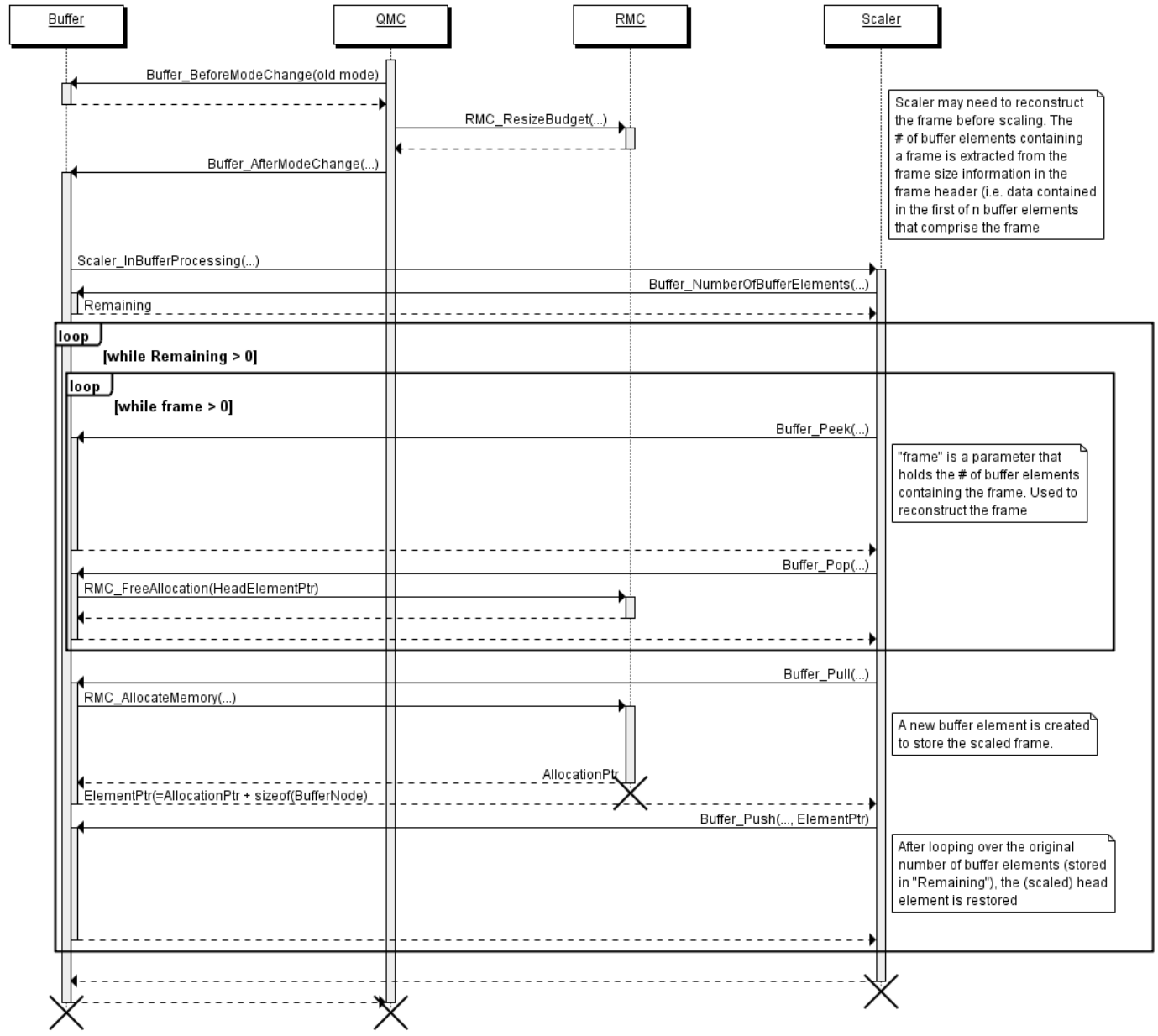
**(Since dropping is performed to avoid stalling and no budget resizing is involved, it is considered a normal mode operation. Hence, QMC is not involved)**



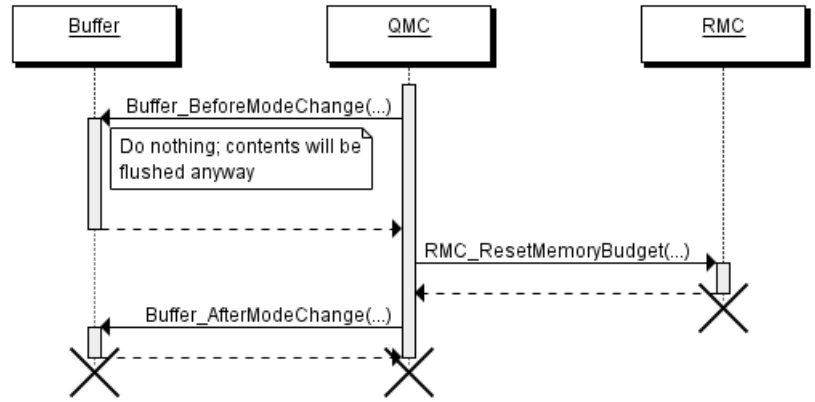
**MSD6: Downscale**



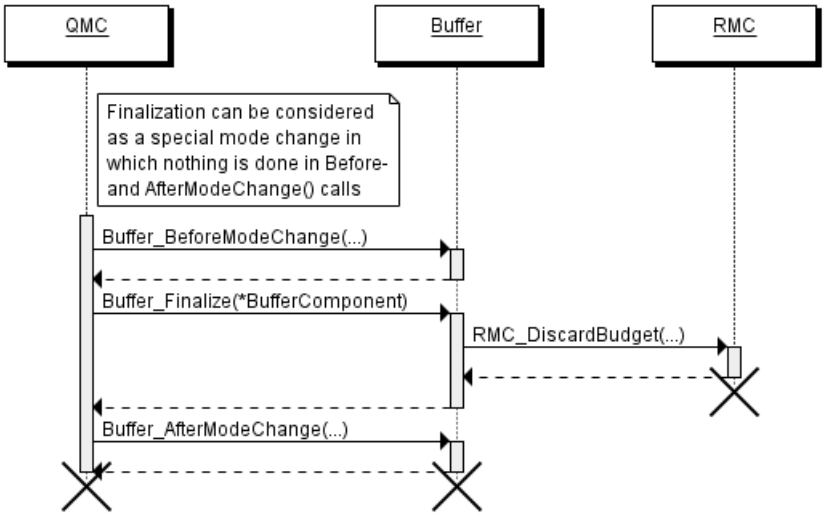
**MSD7: Upscale**




**MSD8: Flush**



**MSD9: Finalize**



## 3. Implementation

The buffer was implemented in C programming language just like the existing RMC. In this section, the actual (software) implementation of the buffer component is highlighted and related to the specification described earlier.

### 3.1. Target Platform and Operating System

The buffer was implemented as a C library, and therefore can be used in any C program. It was intended to run on top of the uC/OS-II real-time operating system, in the experimental setup for swift mode changes [1].

### 3.2. Relating Implementation to Specification

In realizing the methods specified in Section 2.5, a few assumptions and changes were made which need to be clarified.

#### Changes

1. The buffer has a number of pre-defined status codes (see **`buffer.h`**). Thus, each of the methods returned a status code from which its success or failure can directly be determine. Hence, those methods that return an output parameter did so by employing calls by reference. Calling by reference is a concept in C that allows the caller to send a pointer to an output parameter so that it can be modified by the called method.
2. Every method call takes a pointer to the header of the buffer upon which the method is being called. This is necessary to check pre-conditions and ensure correct buffer administration.

Thus, the syntax for all method prototypes was as follows where parameters in square brackets are optional:

```
StatusCode Method_Name(*BufferComponent, [Input Parameters], [Output Parameters]);
```

#### Assumptions

1. The implementation makes no assumptions on whether allocation sizes are fixed or variable. However, as stated earlier, the current impression is in favor of fixed-sized allocations. So, although some parameters may appear redundant under this fixed-sized allocation assumption, they have been included in case this assumption is lifted in the future. In a sense, we have tried to make the buffer as generic as possible.
2. When the shared memory pool is integrated with the buffer components, the interfaces it provides for memory management would the same as those currently provided by the RMC (even though some parameters may be redundant). As long as this assumption holds, the buffer will nicely fit into the system with or without a shared memory pool with no modifications whatsoever.

Finally, we note that regardless of how budgets are defined in the RMC (contiguous chunks versus a collection of scattered allocations), the buffer still provides the desired functionality.

## 4. File Structure

The buffer component is split among a number of files. In addition to these, the complete RMC implementation and a dummy scaler component were included to enable testing.

**buffer.h** specifies the interface methods offered by the buffer to other components (i.e. provided interface). This file also contains a comprehensive documentation of each method and serves as a good reference to supplement the description provided in this report.

**buffer.c** contains the implementation of the buffer interface described in the preceding section. Inline commenting was used extensively. It was intended that the implementation can be fully understood even without referring to this report.

**buffer_types.h** contains type definitions of structures internal to the buffer namely, the *BufferNode* and *BufferComponent*. These are used in **buffer.c** and **test_buffer.c**

**test_buffer.c** contains a comprehensive test suite, running in four different modes and testing all interface standard FIFO methods (Mode 1) and all methods for extra functionality (Mode 2) of the buffer. It also includes a simulation of normal operation (Mode 2) and mode changes (Mode 4).

**test_buffer.h** specifies the methods used in the test suite.

**scaler.h** contains the interface method provided by the scaler (and passed to the buffer via a pointer to function to break cyclic dependency in the include structure).

**scaler.c** implements a dummy scaler which was used to in the test suite to enable mode change simulation.

## 5. Experimental Results

It was not possible to perform any experiments with the buffer components. This is because the prototype in which the buffer is expected to function is currently being developed in the context of [1]. Once this prototype is completed, experiments will be performed to verify that the buffer actually provides the desired functionality. Nevertheless, following the exhaustive unit-tests performed on the buffer, we are confident that it indeed meets the requirements of a scalable buffer with in-buffer processing capabilities for swift mode changes.

The actual experiments to be performed to illustrate the buffer functionality are expected to closely match those performed in the test suite using a dummy context (i.e. dummy QMC and Scaler).

## 6. Conclusion

The goal of this assignment was to investigate, design and implement a scalable buffer with interfaces allowing for in-buffer processing and the capability of dynamic resizing at run time. It was required to run within the context of an on-going research on swift mode changes in memory constrained real-time systems.

From our analysis of the design space, we noted that relocation of the buffer in physical memory should be avoided because it has the disadvantage of increasing the mode change latency. We came up with a configuration in which no relocations of the buffer were necessary while providing an efficient use of memory given certain realistic assumptions.

Our preliminary study (literature review) revealed that no such effort has been undertaken in the past for which we consider this an original work and a valuable contribution to illustrating the feasibility of our novel concepts in [1].

### 6.1. Future Work

Currently, we have assumed a buffer with a single writer and single reader interface. Extending this to one that supports multiple writers and readers while providing mutual exclusive access to data and preserving data integrity is considered a possible improvement.

Also, the buffer methods are implemented assuming components that polling the buffer until their requests can be granted or rejected e.g. a reader that polls an empty buffer until some data is available in the buffer. In the event that this buffer will be used with components that need to be triggered (i.e. "woken up") the buffer implementation could be extended with semaphores or similar synchronization primitives. Developing such a component would even increase the adaptability of the buffer component to different systems.

Finally, the actual integration of the buffer with the shared memory pool is another possible direction.

## 7. REFERENCES


[1] M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrainedreal-time systems*. In *Proc. IEEE/IFIP International Conference on Embedded and UbiquitousComputing (EUC09)*. 2009b.

[2] M. Holenderski, R. J. Bril, J. J. Lukkien. *Swift mode changes in memory constrainedreal-time systems*. Tech. Rep. CS-09-08, Eindhoven University of Technology, 2009a. URL http://w3.win.tue.nl/en/research/research_computer_science/.

[3] M. Holenderski, "Resource management component: Design and implementation," 2008. [Online]. Available: http://www.win.tue.nl/san/projects/cantata/

[4] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles. *Dynamic storage allocation: a survey and critical review.* In: Baker H (ed) Proc. of the international workshop on memory management, vol. 986, pp. 1 – 116, 1995.

[5] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems,* vol. 40, no. 2, pp, 149-179, 2008.

[6] R.J. Bril and M. Holenderski, *Memory sharing for queues of media processing chains,* TU/e, WIN, SAN, Assignment description, Version 0.3, February 2009.

[7] J. Morris, "Data Structures and Algorithms: Objects and ADTs," 1998. [Online]. Retrieved: 13.05.09, Available: http://www.cs.auckland.ac.nz/software/AlgAnim/objects.html, Section 2.2.4

[8] J. Watkinson. *The MPEG Handbook*, p. 259. 2nd Edition. Focal Press, 2004.

[9] M. A. Albu, "Behavioral Analysis of Real-Time Systems with Interdependent Tasks," Ph.D dissertation, Faculty of Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e), Eindhoven, The Netherlands, April, 2008. http://alexandria.tue.nl/extra2/200810687.pdf

## 8. APPENDIX

In this appendix, the analysis of the system configurations presented in Table 1 is discussed in somewhat more details. In what follows, it is assumed that the selected input sequence illustrating the worst-case scenario in each of the configurations can actually occur in a decoder otherwise, it would be useless to consider it. We also emphasize that this discussion are based on the initial assumption in Section 3.3 as well as the definition of a budget as a contiguous chunk in memory.

Symbols in curly brackets are used. Whereas {+} indicates a relative advantage, {–} indicates a relative disadvantage. It is also important to note that although omitted in the figures, each buffer will have a header containing some additional information for administration. This also has to be taken into account when defining buffer capacity.

**Configuration 1: Variable-sized budgets and variable-sized allocations**

- Variable-sized budgets → budget external fragmentation may occur → QMC may need to move budgets around during mode change (and memory is typically slow especially external memory). Hence, budget external fragmentation is always minimized by moving budgets around during a mode change.
- If it exists, budget internal fragmentation = sizeof(int) – (sizeof(budget) mod sizeof(int)) ⩽ sizeof(int)→ constant; otherwise zero.
- Variable-sized allocations → allocation external fragmentation possible.
- Maximum allocation external fragmentation during normal operation is limited (assuming MPEG video encoding) to 2(I–1) where I, P, B are the worst-case sizes of an I-frame, P-frame and B-frame, respectively, and I ≥max(B, P). The worst-case scenario illustrated in Figure A1. Assuming an I-frame is ready at the input but there are I–1 units at both ends of the budget tail.

The RMC always starts from the first (i.e. leftmost) free space in the budget tail and scans through the sorted free list until it finds a free chuck that can accommodate the allocation. It then takes the allocation from the rightmost end of this free space (see [3] for illustrative figures). Furthermore, the RMC implementation does not support combining free spaces at both ends of the budget into a single chunk. Thus, since we also assume FIFO access during normal operation, the total unusable free space at any given time cannot exceed 2(I–1). However, the actual distribution of this free space may differ from what is shown in Figure A1 depending on the allocation request sequence.

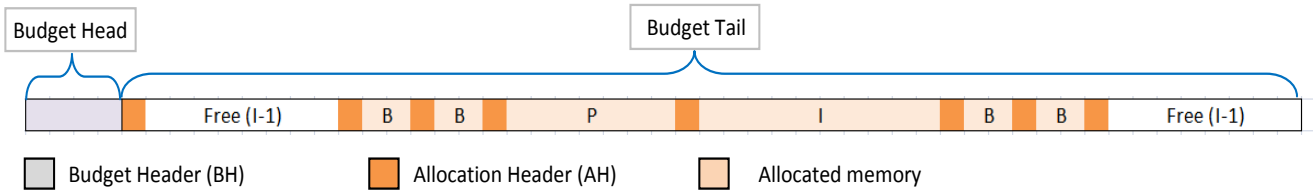



**Figure A1: Worst-case Scenario for Allocation External Fragmentation for Configuration 1**

Solution: Wait until the next frame is consumed by the next component in the pipeline i.e. the reader with respect to that buffer. The solution proposed is valid for all of the buffers in the proposed applications (see Section 1). Note that this situation cannot lead to reader starvation because there must be frames to be consumed in the buffer otherwise it will not occur in the first place.

- If it exists, allocation internal fragmentation = `# of` unaligned allocations * [sizeof(int) – (sizeof(allocation) mod sizeof(int))] ⩽ n * sizeof(int) where n is the number of frame of the smallest size that can fit in the budget tail. Since allocation size is variable, the number of allocations in the buffer is dynamic. However, a worst case bound can be derived using the smallest possible frame size if it is known.
- After mode change: Budget external fragmentation is completely removed. {+}

- Budget may be moved by QMC during mode change thereby invalidating existing references to memory allocations within the moved budgets (i.e. time loss both for moving budgets and updating invalidated pointers). {–}
- Header overhead = BH + (AH * # of frames) = Achievable minimum. {+}

**Summary:** Best when memory is highly constrained (since allocation internal fragmentation is completely removed upon mode change, no memory space is wasted) and some overhead of (possibly) moving some budgets during a mode change is tolerable.

**Configuration 2: Fixed-sized budgets and variable-sized allocations**

- Fixed-sized budgets → (budget external fragmentation does not occur → No movement of budgets) but buffer may have to manage several budgets.
- Budgets do not necessarily have to be contiguous in memory.
- If it exists, budget internal fragmentation = # of unaligned budgets * [sizeof(int) – (sizeof(budget) mod sizeof(int))] $\leqslant m * \text{sizeof(int)}$ where $m$ is the total number of budget ($m \geqslant 1$).
- Maximum allocation external fragmentation during normal operation is limited to (1 + # of budgets)*(I-1).

This is illustrated in Figure A2 using a buffer having two budgets. Assume that an I-frame is ready to be input but cannot fit into any of the free chunks in all budgets managed by the buffer and that budget 1 is the budget currently being read from. As seen in the figure, budgets of the same component need not be contiguous in memory.

The bound is obtained by observing that based on the first-fit allocation scheme used by the RMC in which it finds the first free space that can hold the allocation, only one budget, namely the one currently being read from, can suffer maximum allocation external fragmentation of 2*(I-1) as discussed in Configuration 1. The extra (I-1) unusable space in this budget arises from the last read operations which are assumed to have created the space. All other budgets have at most (I-1) units of unusable space. If any of these other budgets has more than (I-1) unusable space, FIFO access must have been violated at some point. We further assume that whenever the buffer receives a new allocation request it searches it budgets for free space in the same order e.g. in our example, budget 1 first then budget 2. Thus, when requested to discard (a) budget(s) during a mode change it attempts to free them in reverse order i.e. in the example, it first checks budget 2 and tries to move its contents to budget 1 so as to return budget 2 to the RMC.

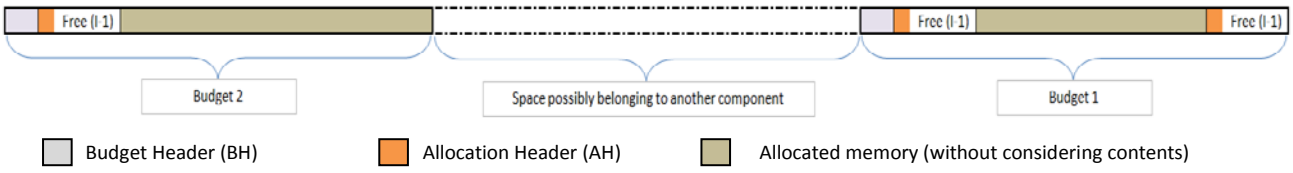



**Figure A2: Worst-case Scenario for Allocation External Fragmentation for Configuration 2**

Solution: Wait until the next frame is output. Again this situation cannot lead to reader starvation because there must be frames to be consumed in the buffer.

- If it exists, allocation internal fragmentation = # of unaligned allocations in all budgets * [sizeof(int) – (sizeof(allocation) mod sizeof(int))] $\leqslant n * \text{sizeof(int)}$ where $n$ is the number of frame of the smallest size that can fit in a budget tail multiplied by the number of budgets.
- After mode change: Budget external fragmentation is not completely removed since a component may retain one extra budget on account of a single frame. Assume the QMC requests the buffer to return one budget and after defragmentation, Figure 4 results. The buffer cannot return Budget 2 due to a single frame X (which may be an I-, P- or B-frame) whose size is at least 1 unit larger than the free space in Budget 1. In the worst case this situation may occur in all buffers and possibly inhibit a mode change. {–}

- Maximum budget external fragmentation occurs when one component requests a new budget but all other components suffer from the situation depicted in Figure A3. Thus, although the total amount of free space across budgets of different components is more than the requested single budget, none of them is large enough. From Figure 4, we observe that each of the affected components wastes **`(budget size - 1)`** units resulting in a total of **`(# of components - 1) * (budget size - 1)`** units.

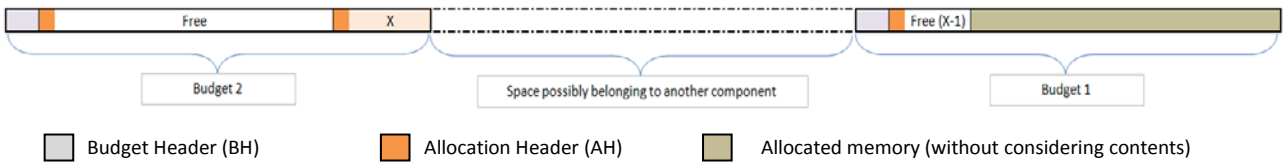



**Figure A3: Showing Possible Internal Fragmentation During Mode Change**

- Budgets are not moved by QMC during *mode change* {+} and they do not have to be contiguous in memory.
- Header overhead = (BH * # of budgets) + (AH * # of frames). [{–} but bearable if # of budgets is not too large]

**Summary:** Best when moving budgets by the QMC during mode change is intolerable; comes at the cost of extra memory in the event the situation illustrated in Figure A3 occurs (in the worst case, this may occur in all buffer components and could possible inhibit a mode change).

**Configuration 3: Variable-sized budgets and fixed-sized allocations**

- Variable-sized budgets → budget external fragmentation may occur → QMC may need to move budgets around during mode change (and memory is typically slow especially external memory).
- If it exists, budget internal fragmentation = sizeof(int) – (sizeof(budget) mod sizeof(int)) ⩽ sizeof(int)→ constant; otherwise zero.
- Fixed-size allocations: → One of two cases

**Case 1**: All frames are the same size

  - Choose allocation size equal to frame size → No allocation external fragmentation is possible during *normal operation* or after *mode change*.
  - If it exists, allocation internal fragmentation = **`n * [sizeof(int) - (sizeof(allocation) mod sizeof(int))]`** where **`n`** is the number of allocations in the budget.
  - Also, selecting budget size as a multiple of the (fixed) frame size → Configuration 4, Case 1.

**Case 2:** Frames have arbitrary sizes

  - Fixed allocation size must be large enough to contain the largest frame (follows from the assumption that single frame across multiple allocations is NOT allowed) → unavoidable allocation internal fragmentation.
  - Worst-case allocation internal fragmentation = **`n * [sizeof(allocation) - (sizeof(smallest frame)]`** where **`n`** is the number of smallest-sized frames that can fit in the budget. This scenario occurs when the buffer is filled up with only smallest-sized frames. We assume that the memory lost due to unaligned allocations is much less than the difference between the largest and smallest frames.
  - No allocation external fragmentation is possible during *normal operation* or after *mode change* since each allocation has the same size = *sizeof(largest frame)*.
- After *mode change*: Budget external fragmentation may be (is?) completely removed. {+}
- Budget may be moved by QMC during *mode change*. {–}
- Header overhead = BH + (AH * # of frames) = Achievable minimum {+}

**Summary:** This configuration appears to offer no benefits compared to the others. Instead it suffers extra overheads (both allocation internal fragmentation and memory moves by QMC).

**Configuration 4: Fixed-sized budgets and fixed-sized allocations**

- This is a simplification of Configuration 3 when budget size is a multiple of fixed allocation size.
- Fixed-sized budgets → no budget external fragmentation (i.e. no budget moves during mode changes)
- Fixed-sized allocations → no allocation external fragmentation during *normal operation*
- If it exists, budget internal fragmentation = `# of unaligned budgets * [sizeof(int) - (sizeof(budget) mod sizeof(int))]` ≤ `m * sizeof(int)` where *m* is the total number of budget
- Allocation internal fragmentation:
  - Case 1: All frames have the same size → allocation internal fragmentation, if it exists = `n * [sizeof(int) - (sizeof(allocation) mod sizeof(int))]` ≤ `n * sizeof(int)` where `n` is the total number of allocations in all budgets.
  - Case 2: Frames have different sizes → worst-case allocation internal fragmentation = `n * [sizeof(allocation) - (sizeof(smallest frame)]` where `n` is the number of smallest-sized frames that can fit in the budget.
- Header overhead = (BH * # of budgets) + (AH * # of frames). [{–} but bearable if # of budgets is not too large]

**Summary:** This is THE configuration when all frames are fixed-size. Unfortunately, this assumption does not hold for our target applications which involve decoding.