

Resource Management Component: Design and Implementation

ITEA-CANTATA

.....



CANTATA

Project number: ITEA05010
Document version no.: 0.5
Status: Draft
Edited by: Mike Holenderski

ITEA Roadmap domains:

Major: Services & Software creation
Minor: Cyber Enterprise

ITEA Roadmap technology categories:

Major: Content
Minor: Data and content management

Document history

Document version #	Modifier	Date	Remarks
v0.1	TU/e SAN	25 May 2008	Initial implementation description
v0.2	TU/e SAN	7 Jun 2008	More elaborate description, after remarks from Logica.
v0.3	TU/e SAN	30 Jun 2008	Minor changes, after remarks from Logica.
v0.4	TU/e SAN	8 Jul 2008	Added a remark in the Remarks section
v0.5	TU/e SAN	4 Sep 2008	Extended section on fragmentation with preventive strategies.

1 Introduction

This document describes the working of the Resource Management Component (RMC). It is aimed at system integrators who may need to look at the code and fix bugs or extend the existing functionality. This document, together with inline documentation, should help them understand the code.

1.1 The Resource Management Component

The Resource Manager Component is a part of the CANTATA Runtime Environment. It is responsible for managing the memory for the components ¹. It provides interfaces to request and discard memory budgets, and to allocate and free memory within an allotted budget.

The RMC provides guaranteed memory access by granting memory budgets to components. Upon a budget request the RMC checks whether there is sufficient space to accommodate it. The component then can allocate memory within the budget. Upon each allocation request the RMC checks whether there is enough space available in the budget and makes sure that no other component “steals” the allocated budget. A component can be sure that what it has requested and was granted by the RMC will indeed be available.

Memory within the budgets can be freed, as can the complete budgets be discarded. When memory within a budget is freed it can be reallocated by the component (the owner of the budget). When a budget is discarded it can be requested again, also by other components.

Budgets can be shared between components by embedding them inside special components, such as a buffer or Quality of Service component. These components can control access to a shared memory by requesting several budgets and then controlling access from other components to the memory allocated inside these budgets, e.g. by synchronizing reading and writing to a buffer, or reassigning budgets between components to control the QoS.

The intended use for the RMC component is the following:

Initialization phase:

- The RMC is initialized with a call to `RMC_init()`, which allocates and initializes the internal RMC data structures. This is usually performed by the Runtime Environment. After the RMC is initialized each component can request (several) budgets with `RMC_RequestMemoryBudget()`. `RMC_init()` must be called before any other RMC interface method. It is meant to be called by the Runtime Environment.

Operation phase:

- Each component allocates memory within a budget with `RMC_AllocateMemory()`.
- When it wants to change the allocation within a budget, e.g. due to a mode change, then it resets the complete budget with `RMC_ResetMemoryBudget()`, or frees a particular memory allocation within a budget with `RMC_FreeMemory()`. It can then allocate the freed memory again.
- A component can claim additional budgets during the operation phase similarly to the initialization phase, by calling `RMC_RequestMemoryBudget()`.

¹ Note that the current version of the RMC manages only the memory. It is intended to manage other resources, e.g. the processor and network, in the second CANTATA demonstrator.

- A component can also discard a budget with `RMC_DiscardMemoryBudget()` (e.g. because of a lower memory demand due to a mode change) making the freed space available to other components, which can be reclaimed by calling `RMC_RequestMemoryBudget()`.

Finalization phase:

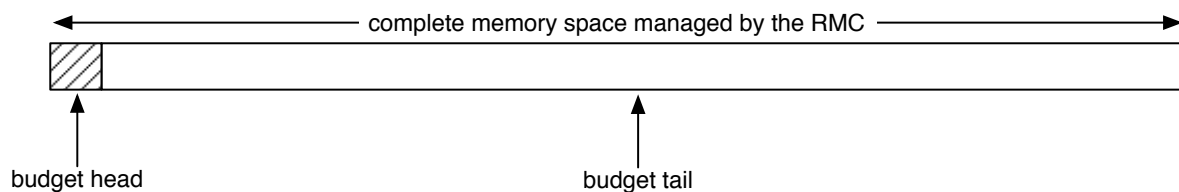
- The `RMC_finalize()` function frees all the memory, which was allocated by the RMC during the `RMC_init()` call. It is meant to be called by the Runtime Environment.

2 Memory management

In real-time applications, budgets are usually allocated during the initialization phase. RMC, however, allows to request and discard budgets, as well as allocate and free memory within the budgets, dynamically during runtime. Once memory is freed (or a budget discarded) it can be reallocated again.

2.1 Memory structure

The whole memory distributed by the RMC among the requested budgets is stored in a single contiguous chunk of memory. Each budget consists of a *head* and a *tail*, as shown below.



Initially the complete memory space is occupied by a single *free* budget. The budget head is a data structure keeping track of the *size* of the tail, pointers to two linked lists: the `freeList` and the `allocatedList`, and a pointer to the `next` budget.

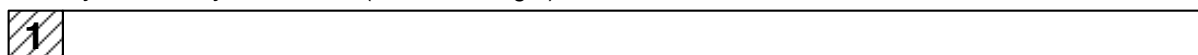
- **Size:** The size of the tail, i.e. the size of memory available for allocation.
- **Free list:** The free list keeps track of the available space within the budget. Initially it points to the beginning of the tail and occupies the complete tail. The free list is always sorted, i.e. node *A* in the free list precedes node *B* if and only if the memory address referred to by node *A* is smaller than the memory address referred to by node *B*. The free list is sorted to facilitate the merging of neighboring free memory spaces, as described later.
- **Allocated list:** The allocated list keeps track of the allocated memory within the budget. Initially it is empty. Unlike the free list it is not necessarily sorted.
- **Next:** A pointer to the next budget. If the current budget is free (i.e. it is a member of the global free list) then the next pointer points to the next available budget. If the current budget is allocated (i.e. it is a member of the global allocated list) then the next pointer points to the next allocated budget. If this is the last budget in the list, then the next pointer is NULL.

2.2 Requesting budgets

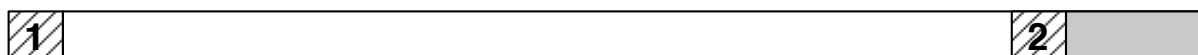
Budgets are requested with `RMC_RequestMemoryBudget()`. Upon a request, the RMC traverses the free list of all available budgets, starting at the first node in the global free list, and checks whether the requested budget can fit. If so, the requested budget is taken from the **end** of the free budget and the free budget is reduced accordingly.

The figure below shows the memory structure after several budget requests, starting with the initial configuration with all memory still available.

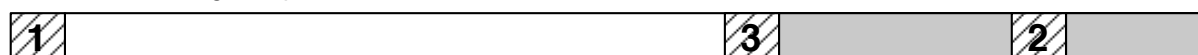
Initially all memory is available (one free budget)



After first budget request

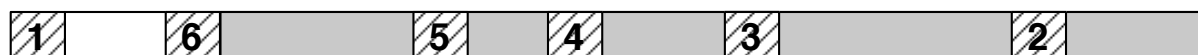





After second budget request



⋮

After five budget request

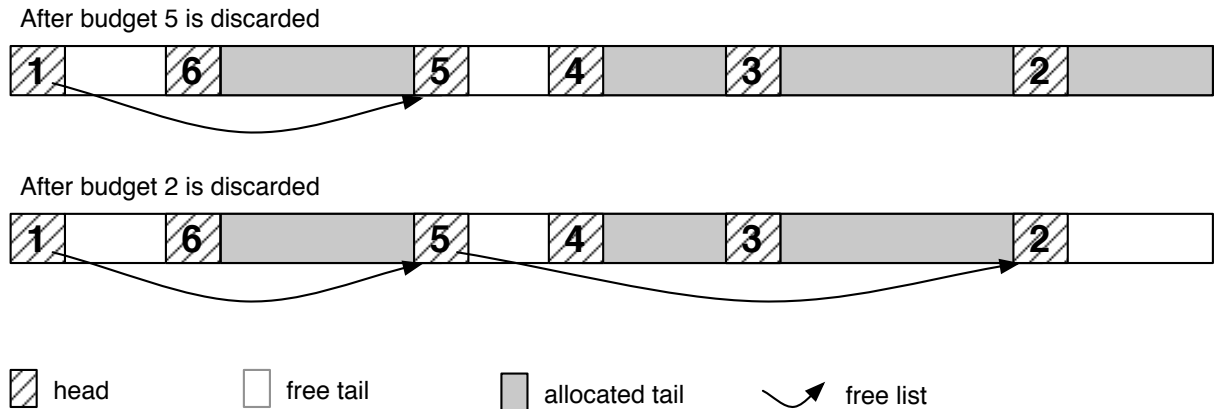


 head
  free tail
  allocated tail

If the requested budget does not fit in the tail of a free budget, but fits within the head and tail, then the complete free budget is allocated and removed from the free list.

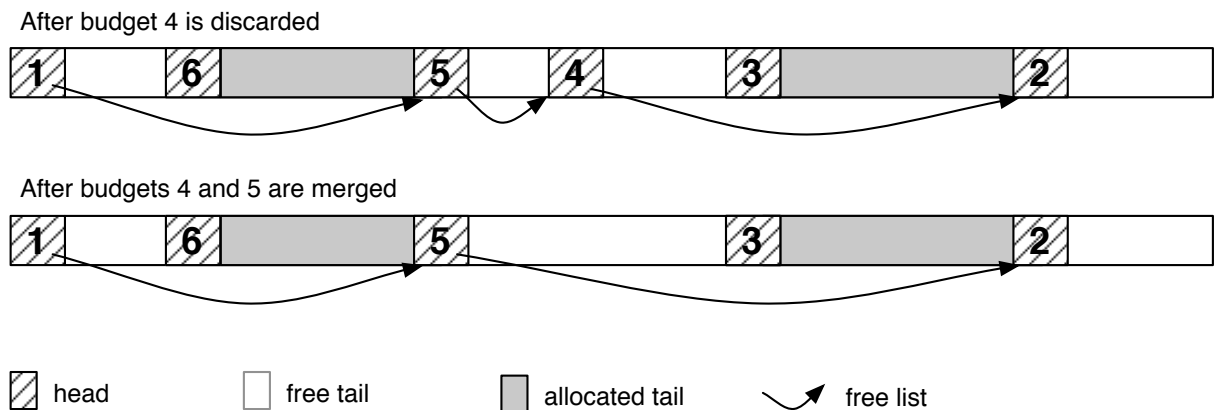
2.3 Discarding budgets

Budgets are discarded with `RMC_DiscardMemoryBudget()`. Following our example above, the following figure shows discarding budgets 5 and 2.



As budgets are discarded, they are added to the global free list. Note that the above figure ignores the allocated list. It is used by the RMC only to check if the budget handle parameter (passed as an argument to one of the interface functions) is a valid budget id, and therefore it is not relevant from the the memory management point of view.

When a budget is discarded the RMC scans the whole free list and merges any neighboring budgets into larger budgets. This allows to grant larger budgets upon following budget requests. The following figure shows discarding budget 4 and merging it with the available budget 5.



2.4 Memory Allocation

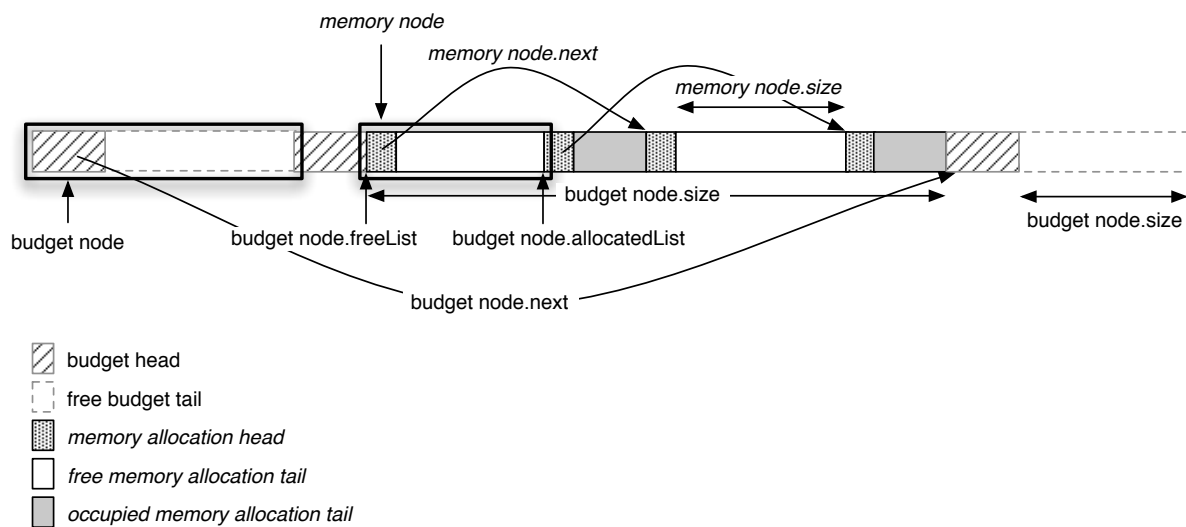
Memory is allocated within budgets. Each budget head stores a `freeList` pointer, referring to the beginning of the free list containing available memory allocations. Memory allocations within budgets are managed in the same way that budgets are managed on the global level, with the only difference that they are limited to the tail of the corresponding budget, and the headers for memory allocations and budgets have different sizes.

Memory is allocated with the `RMC_AllocateMemory()` method, which takes as one of the arguments the target budget.

Memory is freed with the `RMC_FreeMemory()` method. The freed memory is inserted in the free list of the corresponding budget and merged with neighboring free nodes similarly to merging budgets.

A convenience method is provided to discard all the memory allocations within a budget, called `RMC_ResetMemoryBudget()`.

The memory organization of budgets and memory allocations within budgets is shown in the following figure.



Note that budgets and memory allocations are stored back-to-back, occupying contiguous memory locations. Therefore special care has to be taken by the component developers not to write outside the boundaries of their memory allocations. Otherwise the complete memory structure managed by the RMC can be corrupted.

2.5 Fragmentation

RMC does not address fragmentation, which may result from repeated allocation and deallocation of memory and budgets. However, if freed budgets or memory allocations happen to be neighbors, then they are merged.

Fragmentation can be avoided by applying the following strategies:

- *Split component's memory requirements across several budgets of the same size.* For example, given a system with total memory of 2000 bytes and an application with two components A and B requiring 1000 bytes each, both components can request 10 budgets of 100 bytes each. During runtime, when component A discards several budgets (e.g. because of a lower memory demand due to a mode change), component B can request new budgets (also of size 100 each). Since the budget sizes are the same, this will leave no gaps between the budgets in the memory, and thus prevent fragmentation.
- *Request and discard budgets in stack or FIFO order.* For example, a component requesting three budgets A, B and C (in that order), should discard them in stack order (C, B, A) or FIFO order (A, B, C). It should not discard budget B before discarding the other budgets, to prevent creating gaps in the memory between the budgets.

Note that similar strategies hold for allocating memory within budgets.

3 Monitoring

RMC provides two functions for monitoring the available memory:

`RMC_getAvailableMemory()`: Get the total number of bytes currently available to new budgets requests. Note that since the RMC does not address fragmentation, the number of bytes returned by this function may not be allocable in one contiguous chunk, i.e. a single budget.

`RMC_getAvailableBudget()`: Get the number of bytes currently free within a budget. Note that since the RMC does not address fragmentation, the number of bytes returned by this function may not be allocable in one contiguous chunk, i.e. a single memory allocation within the budget.

4 File structure

The RMC is split among several files:

<code>rmc.h</code>	specifies the interface methods offered to other components.
<code>rmc.c</code>	is the implementation of the interface, as described above.
<code>rmc_types.h</code>	contains type definitions internal to <code>rmc.c</code> , but which are shared with <code>rmc_debug.c</code> and <code>test_rmc.c</code> .
<code>rmc_debug.c</code>	defines several debugging methods used throughout <code>rmc.c</code> and <code>test_rmc.c</code> .
<code>test_rmc.c</code>	is the unit test suit, testing all the interface methods of the RMC declared in <code>rmc.h</code> .

5 Remarks

- The budget size requested by `RMC_RequestMemoryBudget()` must include the overhead due to the memory allocation heads. The size of the head is defined by the global constant `RMC_MEMORY_HEAD_SIZE`. Similarly, `RMC_BUDGET_HEAD_SIZE` defines the budget head size, which must be taken into account when dividing the total memory space between the budgets (usually done by the system integrator).
- `RMC_init()` and `RMC_finalize()` must be called in alternating fashion. In particular, `RMC_finalize()` must not be called twice without a call to `RMC_init()` in between, because this will give rise to a “malloc: double free” error. `RMC_init()` and `RMC_finalize()` are intended to be called by the runtime environment (and not by the components).