

Dynamic Memory Allocation

Embedded Systems

Erick Eduardo Vite Lopez

Email: erick.vite@iteso.mx

Jaime Leonel Navarro Ocampo

Email: jaime.navarro@iteso.mx

[09/10/2020](#)

Embedded Systems Specialization Program

www.sistemasembebidos.iteso.mx/alumnos

ITESO A. C., Universidad Jesuita de Guadalajara

Periférico Sur Miguel Gómez Morín #8585, Tlaquepaque, Jalisco, México

Technical Report Number: ESE-O2014-001

® ITESO A.C.

Abstract: *Dynamic memory allocation (DM) is one of the most important elements of modern embedded systems engineering, the following report explains the results of the implementation of a Dynamic Memory Allocation*

Keywords: *Memory, Heap, Memory allocation, Linker*

1. Table of Contents

Table of Contents	1
Table of Figures	3
Introduction	4
Requirements	4
Functional Description	5
Results	7
Conclusions	12

2. Table of Figures

Figure 4.1 - Memory allocation data flow diagram	5
Figure 4.2 - Dependencies	5
Table 4.1 - Description of Mem alloc files	5
Table 5.1 - Description of Mem alloc files	8
Figure 5.1 - Zeroing out the heap section	8
Figure 5.2 - Zeroing out the heap section	9
Figure 5.3 - Size of SchMTaskType	9
Figure 5.4 - Assigning the heap address to our task array	9
Figure 5.5 - Debug view of the memory allocation function	10
Figure 5.6 - Memory map of our heap locations	10
Figure 5.7 - Second heap allocation.	10
Figure 5.8 - Memory view of the second task array.	11

3. Introduction

The task of fulfilling an allocation request consists of locating a block of unused memory of sufficient size. Memory requests are satisfied by allocating portions from a large pool of memory called the heap

The heap is an area of memory set aside to provide dynamically allocated storage. The size and location of the heap are application dependent and the means by which it is allocated is specific to a particular toolchain.

3.1. Requirements

Using the provided base example for the task scheduler, implement a dynamic Memory allocation which should have all the features described on the Project Specification.

After satisfying the requirements on the project Specification:

- Memory Allocation shall be invoked when memory allocation is requested by the project specific component initialization.
- Mem_Alloc shall return the initial address of the new allocated memory space.
- Current Address *currAddr* shall be updated according to the requested size.
- After allocating a new area, Mem_Alloc shall assure the current address is aligned with 32 bit address.
- The available memory in the heap freeBytes shall be updated accordingly.
- Mem_Alloc shall return a NULL pointer and the requested memory allocation shall not be handled if the size exceeds the available memory in the heap.

4. Functional Description

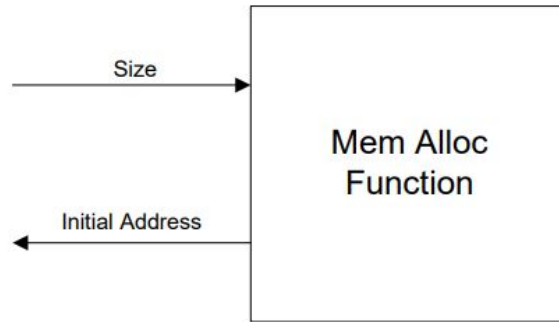


Figure 4.1 - Memory allocation data flow diagram

The figure 4.2 shows the file dependencies where $A \rightarrow B$ indicates A includes B.

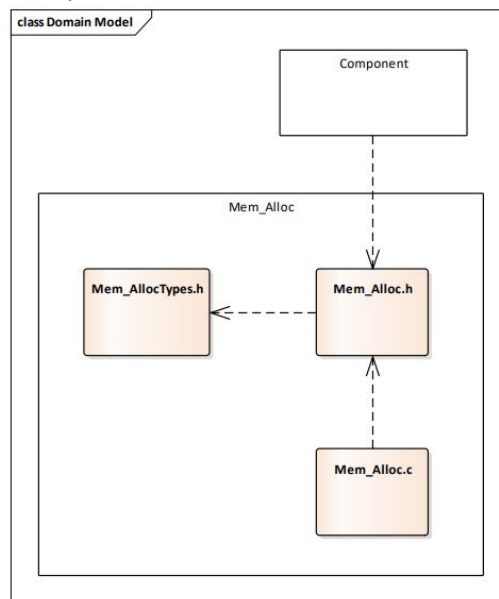


Figure 4.2 - Dependencies

The table 4.1 shows a brief description of each file:

File	Description
Mem_AllocTypes.h	Contains all the internal data types definitions use by the memory allocation handler Module
Mem_Alloc.h	Contains all the interfaces provided to the user component modules
Mem_Alloc.c	Contains the main functionality of the memory allocation handler

Table 4.1 - Description of Mem alloc files

LINKER CONFIGURATION

Memory Allocation area name shall be "heap_memalloc". The heap_memalloc space shall be allocated at the RAM location 0x20400000. The total size of this space shall be 64KB.

The heap_memalloc space and corresponding section references shall be provided from the Linker Configuration File (sam_flash.ld).

Additional heap_memsize configuration in the Linker Configuration File shall be provided. The heap_memsize is the actual heap space to be used in the project. Initial heap_memsize configuration shall be of 4KB.

The startup code shall be updated so that the heap memory is set to the value of zero.
The following labels to support this functionality shall be named as follows:

- `_heap_mem_start`
- `_heap_mem_end`

Hint! The above labels are created in `sam_flash.ld` file.

The data struct `MemHandlerType` elements (memory start address `MemStart`, memory end address `MemEnd`, memory current address `CurrAddr` and available memory bytes indicator `FreeBytes`) shall be statically initialized in the `Mem_Alloc.c` file.

MEMORY INITIALIZATION ACTIVITY

The basic steps to initialize the heap memory supported by the startup code:

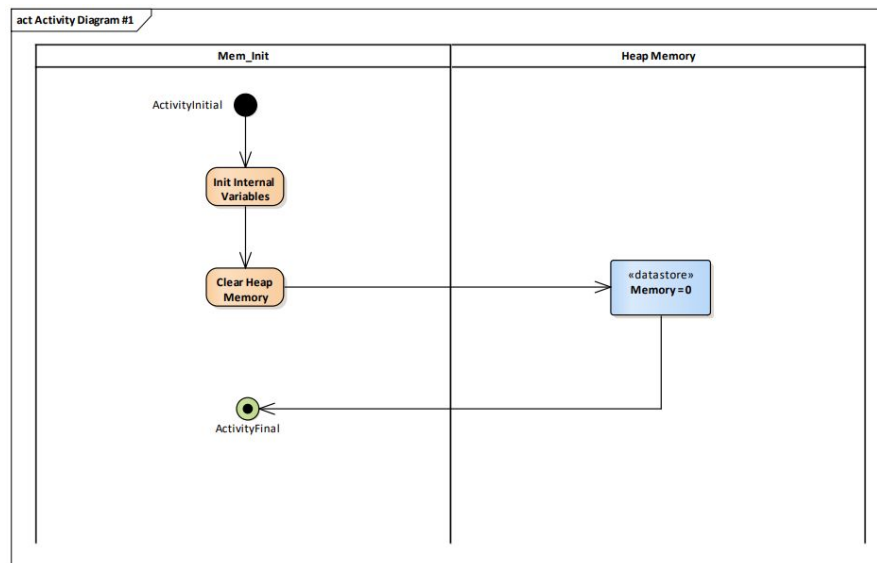


Figure 4.3 - Steps to initialize the heap memory

MEMORY ALLOCATION SEQUENCE

The basic memory allocation sequence is shown in the figure 4.4:

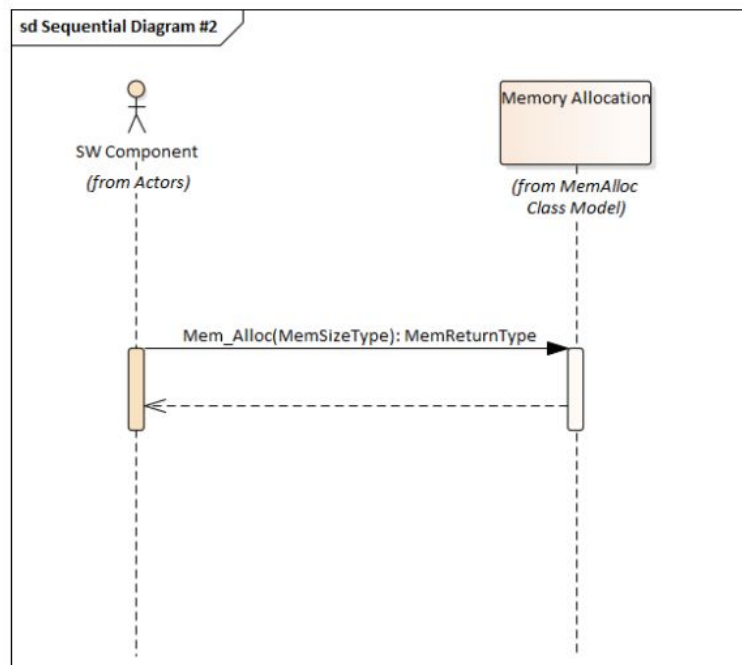


Figure 4.4 - Memory allocation sequence

MEMORY ALLOCATION ACTIVITY

The basic steps to allocate memory through the invocation of Mem_Alloc are shown in the figure 4.5:

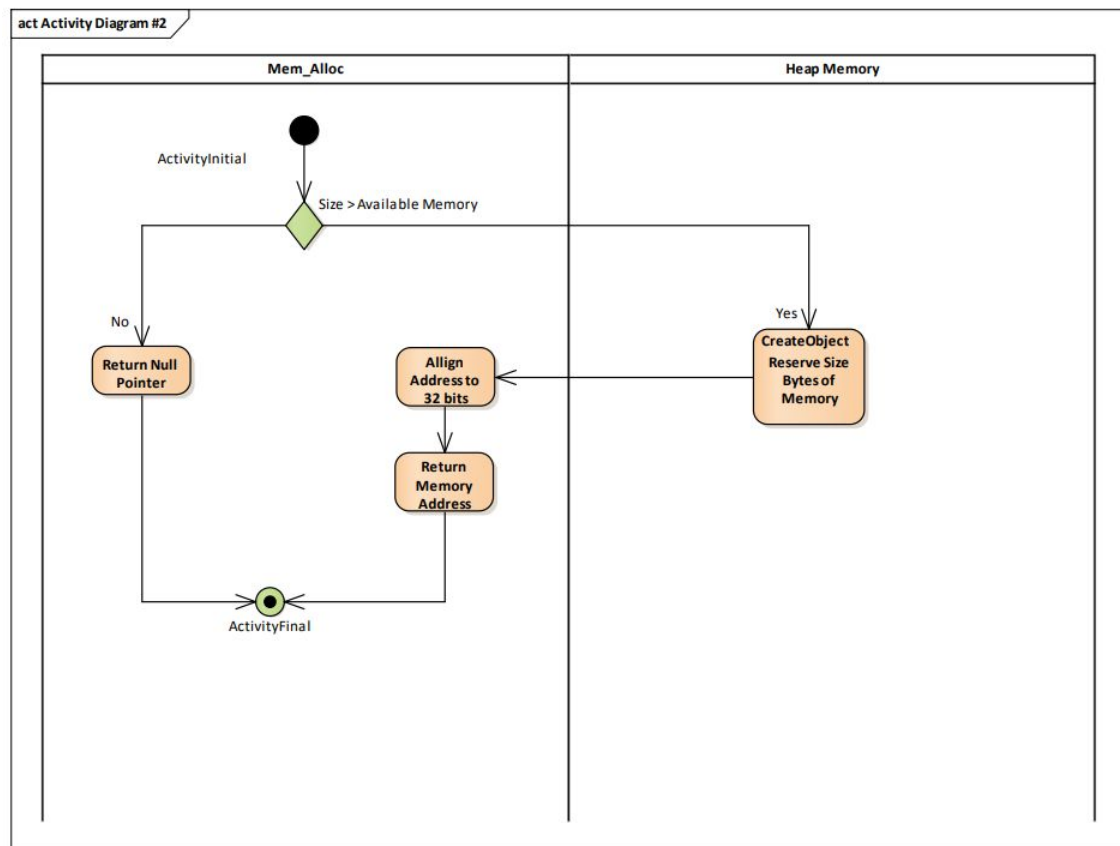


Figure 4.5 - Memory allocation activity

5. Results

Three files were created, Mem_alloc.c, Mem_alloc.h and Mem_AllocTypes.h. A brief description of each file can be found in the Table 5.1:

File	Description
Mem_AllocTypes.h	Contains all the internal data types definitions use by the memory allocation handler Module
Mem_Alloc.h	Contains all the interfaces provided to the user component modules
Mem_Alloc.c	Contains the main functionality of the memory allocation handler

Table 5.1 - Description of Mem alloc files

The main functionality of the Mem_Alloc function is in the Mem_Alloc.c file. The code implemented there consists of a **MemControl** variable of type **MemHandlerType** which contains the address of the start and the end of our heap, and it also keeps track of the current address and the bytes available to assign. If we request more bytes than the amount that can be assigned, a NULL pointer will be returned.

A function **Mem_Alloc**, which takes as a parameter the size of the heap to assign **MemSizeType**, and returns the assigned address, **MemReturntype**, is also implemented in this file. Its job is to keep track of the current heap address, starting from 0x20400000, and depending on the bytes requested, it updates the current address. If another Mem_alloc function call is done, this updated address will be returned.

The results of the implementation of our memory allocation function are as follows:

We can see that the are of memory that we defined for our memory allocation function is now present in the linker address map file which starts on the 0x20400000 address, and has a length of 0x1000, or 4KB, as seen in figure 5.1:

```

7679 .heap_alloc      0x20400000      0x1000
7680                0x20400000
7681                0x20400000      . = ALIGN (0x4)
7682                0x20401000      _heap_mem_start = .
7683 *fill*          0x20400000      0x1000      . = (. + heap_memszie)
7684                0x20401000      . = ALIGN (0x4)
7685                0x20401000      _heap_mem_end = .
7686                0x20401000      . = ALIGN (0x4)
7687                0x20401000      _end = .
7688                0x2045ffff      _ram_end_ = ((ORIGIN (ram) + 0x50000) - 0x1)
7689                0x20401000      _sdram_lma = .
7690

```

Figure 5.1 - Zeroing out the heap section

When we download and run our code into the SAM v71 development board, we enter the reset handler function, which contains the lines of code needed to write zeros to out heap_mem section, as can be seen in figure 5.2:

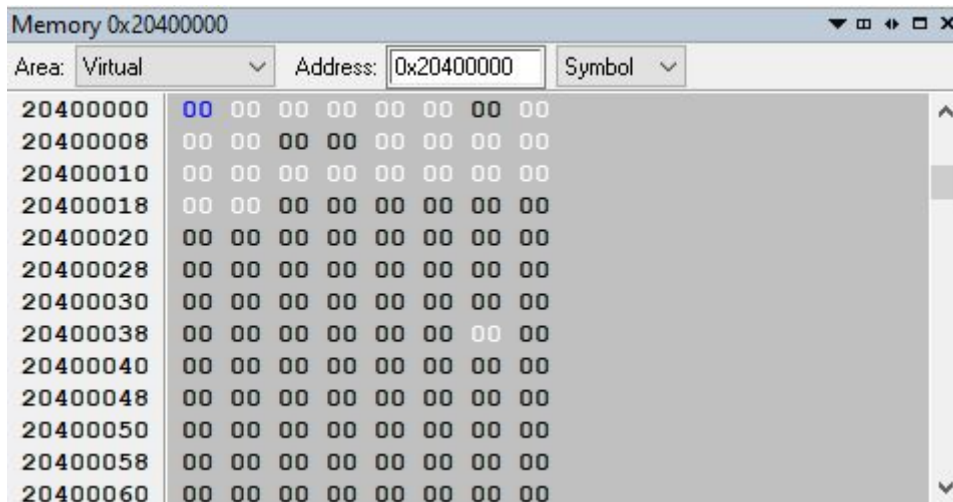


Figure 5.2 - Zeroing out the heap section

With our Heap section defined and zeroed out, we can now make use of our **Mem_Alloc** to reserve heap space. In this example, we are going to use a previous project where we created an array of 7 tasks for a task scheduler, but now we will store those tasks in our heap. This task array, called **TestTaskConfig**, of type **SchMTaskType**, contains the priority of our tasks, their ID and their function pointer.

Our **Mem_Alloc** takes a parameter which tells the function how many bytes it needs to reserve, for this we can use the C built in function **sizeof()** to determine the number of bytes that the variable type **SchMTaskType** needs, and then we can multiply that number by the amount of tasks that we need to allocate. As we can see in the figure 5.3, **SchMTaskType** takes 8 bytes, and multiplying that amount by 7, which is the amount of tasks that we are going to run gives us a total size of **56 bytes** to be allocated. This is the value that will be passed to our memory allocation function.

Size of SchMTaskType: 8 bytes.

Figure 5.3 - Size of **SchMTaskType**

Now we can call our **Mem_Alloc** function, **TestTaskConfig** will be initialized to the address that it returns, in this case, since its the first time that the function is called, the assigned address will be 0x20400000, or the start of our heap, as seen in figure 5.4.

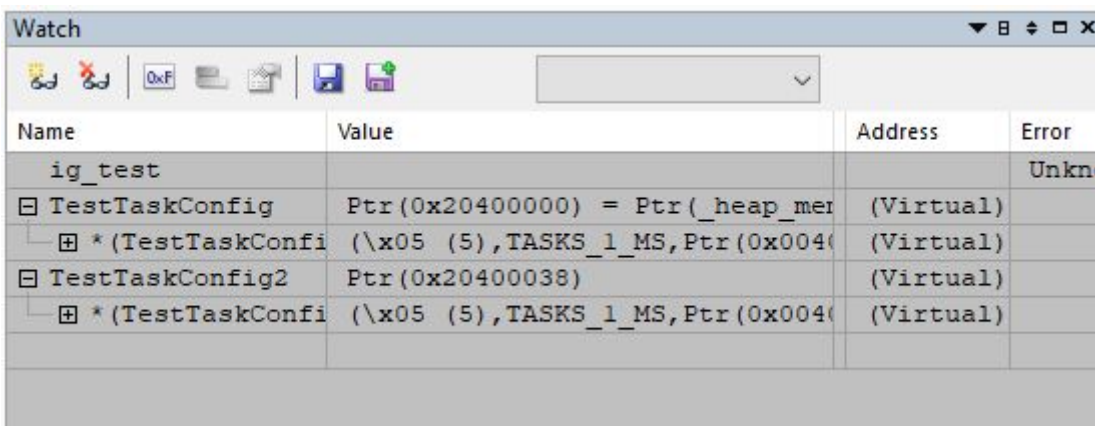


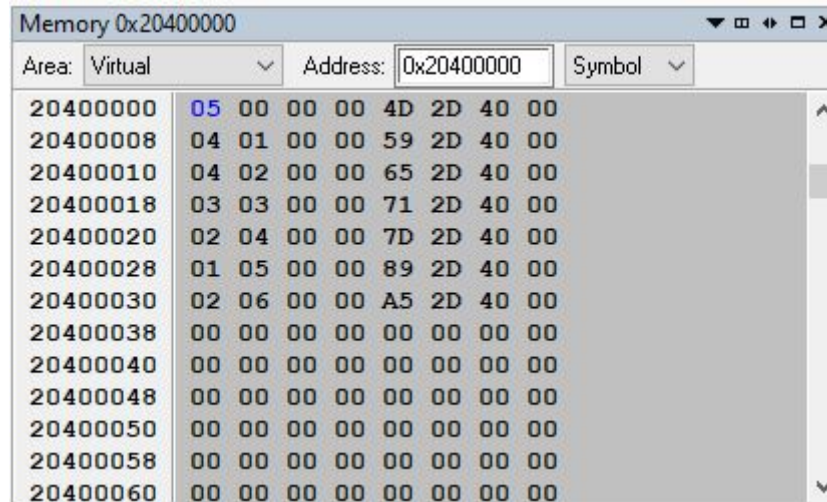
Figure 5.4 - Assigning the heap address to our task array

Initially we had 4096 bytes to assign, now we only have 4040 bytes left after assigning the 56 bytes for our tasks, as seen in figure 5.5

```
Memory Allocation Initialization - 56 bytes to be allocated
Memory Allocation Initialization - 4096 bytes left
Memory Allocation - 4040 bytes left
Memory Allocation - 0x20400000 was the returned address
```

Figure 5.5 - Debug view of the memory allocation function

As we can see in the memory map, our tasks are now stored in our heap, shown in the figure 5.6



Address	Area	Symbol
20400000	Virtual	05 00 00 00 4D 2D 40 00
20400008	Virtual	04 01 00 00 59 2D 40 00
20400010	Virtual	04 02 00 00 65 2D 40 00
20400018	Virtual	03 03 00 00 71 2D 40 00
20400020	Virtual	02 04 00 00 7D 2D 40 00
20400028	Virtual	01 05 00 00 89 2D 40 00
20400030	Virtual	02 06 00 00 A5 2D 40 00
20400038	Virtual	00 00 00 00 00 00 00 00
20400040	Virtual	00 00 00 00 00 00 00 00
20400048	Virtual	00 00 00 00 00 00 00 00
20400050	Virtual	00 00 00 00 00 00 00 00
20400058	Virtual	00 00 00 00 00 00 00 00
20400060	Virtual	00 00 00 00 00 00 00 00

Figure 5.6 - Memory map of our heap locations

To further test the functionality of our implementation, we declared a second task array, **TestTaskConfig2**, using the memory allocation function for this new task array yields the following results:

56 bytes to be allocated, we had 4040 bytes available to allocate, so we end with 3984 after the allocation is complete. As we can see in the figure 5.7, the returned address is now 0x20400038 because our **MemControl** kept track of the “top” of the heap by adding the starting address (0x20400000), and the 56 bytes of the first allocation.

```
Memory Allocation Initialization - 56 bytes to be allocated
Memory Allocation Initialization - 4040 bytes left
Memory Allocation - 3984 bytes left
Memory Allocation - 0x20400038 was the returned address
```

Figure 5.7 - Second heap allocation.

As we can see in our memory, our second task array **TestTaskConfig2** is now also stored in our heap, starting at the address 0x20400038, shown in the figure 5.8

Memory 0x20400000

Area: Virtual

Address: 0x20400000

Symbol

20400000	05	00	00	00	4D	2D	40	00	TestTaskConfig
20400008	04	01	00	00	59	2D	40	00	
20400010	04	02	00	00	65	2D	40	00	
20400018	03	03	00	00	71	2D	40	00	
20400020	02	04	00	00	7D	2D	40	00	
20400028	01	05	00	00	89	2D	40	00	
20400030	02	06	00	00	A5	2D	40	00	
20400038	05	00	00	00	4D	2D	40	00	TestTaskConfig2
20400040	05	00	00	00	4D	2D	40	00	
20400048	05	00	00	00	4D	2D	40	00	
20400050	05	00	00	00	4D	2D	40	00	
20400058	05	00	00	00	4D	2D	40	00	
20400060	05	00	00	00	4D	2D	40	00	
20400068	05	00	00	00	4D	2D	40	00	
20400070	00	00	00	00	00	00	00	00	

Figure 5.8 - Memory view of the second task array.

6. Conclusions

In this project we implemented the Dynamic Memory Allocation and some of the advantages we learned about implementing this are:

- Data structures can grow and shrink to fit changing data requirements.
- We can allocate (create) additional storage whenever we need them.
- Thus we can always have exactly the amount of space required - no more, no less.

Why do we make use of this?

Because, although not clearly visible, not being able to allocate memory during runtime precludes flexibility and compromises space efficiency.

Especially those cases in which the input is not known in advance, the example is preceded in the fact that we can add or remove tasks to our program, initially we only have 6 tasks, then the interruption task was added and perhaps later we could add or remove more tasks