文章编号：2096-1472(2016)-05-21-02

# μC/OS与FREERTOS动态内存管理机制的分析与比较

肖 蕾，刘克江
（广东技术师范学院自动化学院，广东 广州 510635）

**摘 要**：在嵌入式系统领域，如何合理地分配和管理系统内存RAM资源是程序员必须面对的问题，能否高效、可靠、实时地管理动态内存分区决定了整个系统的稳定性和可靠性。本文以μC/OS与FREERTOS两种操作系统为例，在深入研究其动态内存管理机制的基础上，对其优缺点和适用场合进行了分析比较，便于软件开发人员在实际应用中根据产品不同需求进行针对性的选择。

**关键词**：μC/OS FREERTOS；嵌入式系统；动态内存；管理机制
**中图分类号**：TP316.2 **文献标识码**：A

# Analysis and Comparison of μC/OS and FREERTOS Dynamic Memory Management Mechanism

XIAO Lei,LIU Kejiang
( *College of Automation Engineering,GuangDong Polytechnic Normal University,Guangzhou* 510635,*China*)

**Abstract:**In the embedded system field,every programmer has to face the problem about how to reasonably allocate and manage the RAM resource.Stability and reliability of a whole system are determined by the programmer's capability to carry out efficient,reliable and real-time management of dynamic memory partition.Based on in-depth study of μC/OS and FREERTOS dynamic memory management mechanism,the paper comparatively analyzes the advantages,disadvantages and the application occasions,which facilitates developers to make targeted selection of operating systems based on different product requirements in practical applications.

**Keywords:**μC/OS;FREERTOS;embedded systems;dynamic memory;management mechanism

## 1 引言(Introduction)

在嵌入式系统领域中，内存RAM一直是一种稀缺资源。如何合理地分配和管理系统的内存资源是嵌入式软件程序员必须面对的问题，特别是在产品必须使用到动态内存分配时，能否高效、可靠、实时地管理动态内存分区决定了整个系统的稳定性和可靠性[1,2]。

针对这种情况，程序员常用的解决办法主要包括下面三种：在系统启动时就按最坏考虑分配足够大的数组、根据实际需求自行编写内存管理程序或直接使用编译器提供的malloc和free函数、基于嵌入式操作系统内存管理机制来处理。如果程序员能够充分掌握整个系统所有可能出现的情况，根据最坏需求在系统启动时就给每一项作业分配一块足够大的数组是最简单和直接的方法，但势必会造成内存浪费，且如果作业需要数组类型是多种的情况很可能会陷入内存空间不足的困境[3]。根据产品软件设计的需求自行编写简单短小的内存管理程序，对于有经验程序员来说不成问题，但是这种方法也存在着不同平台通用性较差、管理程序稳定性和可靠性因人而异，另外，malloc和free函数并不是所有平台都可以使用且代码不可见。移植现成的嵌入式操作系统，基于操作系统下的内存管理机制来处理系统的动态内存问题是比较方便而且可靠的解决办法，但程序员必须深入了解所使用操作系统动态内存管理机制的特点和区别，并能够针对不同的处理器资源对操作系统进行一定程度裁剪。

上述三种方法中程序员大多比较倾向于移植现有成熟的嵌入式操作系统来处理，本文以μC/OS与FREERTOS两种操作系统为例，在深入研究其动态内存管理机制的基础上，对其优缺点和适用场合进行了分析比较，便于软件开发人员在实际应用中根据产品不同需求进行针对性的选择。

## 2 内存管理算法(Memory management algorithm)

### 2.1 μC/OS动态内存管理

在μC/OS操作系统中，使用动态分配内存时必须先调用OSMemCreate函数建立并初始化一个内存区，该内存区会被分割成n块固定大小的内存块。OSMemCreate函数传递参数指定内存区起始地址、每块内存块大小以及内存块数量。初始化后的结构在每一块空闲块开头存放着指向下一块空闲块指针。初始化后μC/OS为每一个动态内存区定义一个“内存控制块”来记录和跟踪该区的使用情况，包含内存分区指针、空闲块链表指针、每块内存大小、内存块数目和空闲内存块数目，其结构为：

```
typedef      struct
{
    void     *OSMemAddr;
    void     *OSMemFreeList;
    INT32U OSMemBlkSize;
        INT32U       OSMemNBlks;
        INT32U       OSMemNFree;
```

基金项目：广州市科技和信息化局科普计划资助项目，“搭建原创智能车平台，培养中小学生工程实践意识”（项目编号：2013KP054）；广东省科技计划项目，“非开挖地下管道三维探测关键技术研究及装备研制”（项目编号：2015A030401099）.

}OS_MEM；

当用户程序调用函数OSMemGet申请一块动态内存时，系统便通过“内存控制块”将空闲块链表指向的第一块空闲内存块分配给程序，同时将空闲块链表指针指向下一个空闲块并更新空闲块数目。在程序需要释放内存时调用OSMemPut函数，系统根据“内存控制块”把回收的内存块插入到空闲块链表表头，并更新空闲块数目。由此可见，μC/OS每次分配和回收动态内存的时间是确定的，每一次分配和回收的内存大小已知，且没有内存碎片的存在。

μC/OS为了保证内存管理时间确定性和解决内存碎片问题，在用户程序初始化一块动态内存区时便指定了该内存区每一块内存块长度，申请动态内存时只能得到初始化时指定的大小，OSMemGet这个函数并不需要用户程序指定需要申请的内存长度，内存块长度由图2内存控制块中的OSMemBlkSize决定。如果出现多个需要动态使用内存的任务，且每个任务所需的内存块长度都不一样，程序员可以多次调用OSMemCreate函数创建包含不同大小内存块的动态内存区，最大创建的内存区数目可通过OS_MAX_MEM_PART设定，然后程序需要多大的内存块就到对应内存控制块中申请即可；另一种做法就是调用OSMemCreate时根据最大内存块大小去初始化一块动态内存区，所有程序都在该内存区中申请和释放动态内存，在调用OSMemGet得到一块较大的内存块时强制转换成具体程序所需的数据格式即可。

通过分析可知μC/OS处理动态内存的方法具有如下优点：

(1)μC/OS对动态内存分区的管理机制在操作上时间是可确定的。

(2)不会产生所谓动态内存碎片，可以最大限度保证系统的稳定性和可靠性。

(3)每一动态内存块大小固定，每个空闲内存块顶部只需存储下一个空闲块指针，减小了系统额外开支。

但是，该算法的缺点也是显而易见的：

(1)不能灵活充分利用整个内存空间。无论是创建多个内存区还是根据最大内存块创建一个内存区，都会造成严重的内存空间浪费。

(2)缺乏灵活性。用户程序之所以使用动态内存分配目的便是为了提高灵活性，但是μC/OS在初始化动态内存区时便将其划分为固定大小的连续存取区，这样在某些时候不能确定某个内存块大小时便无法通过该算法解决问题。

(3)不检查回收内存块的合法性。μC/OS的内存回收函数OSMemPut在回收内存时并不检查所回收的内存块是否是本动态内存区的空间，用户程序调用OSMemPut函数时传递任何参数，只要内存控制块中OSMemNFree小于OSMemNBlks便将该参数指向的空间作为空闲块链表节点插入到空闲块链表中，这种情况导致的后果是将是不可预估的。

### 2.2 FreeRTOS动态内存管理

FreeRTOS是一个微型嵌入式操作系统内核，具有源码公开、免费、可裁剪、调度策略灵活和简单易用等特点，被很多嵌入式开发人员所选用[4,5]。对于内存管理，FreeRTOS根据使用者实际需求提供三种策略，每种策略对应独立的源文件，需要将对应的文件移植到工程中[6–8]。

策略一是三个方案中最简单的，系统根据onfigTOTAL_HEAP_SIZE设定的大小划分一块内存作为动态内存区，同时定义变量xNextFreeByte标志空闲区域的位置，初始值为0。当用户程序申请动态内存时便返回当前xNextFreeByte代表的内存地址，并将xNextFreeByte加上所申请内存块长度，且内存一旦分配便不允许释放，策略一中没有提供内存回收方法。

策略二中建立空闲分区链表采用首次适应算法分配动态内存，允许分配后的动态内存调用释放函数进行回收，然而，它不具备将邻近空闲块合并成一个大空闲块的功能。FreeRTOS为动态内存分区中每一块空闲块建立一个“空闲分区节点”，并将该节点存放于空闲块顶部。该节点数据结构包含指向下一节点指针和本空闲分区大小。同时定义一个开始节点和终止节点作为空闲分区链表的表头和结尾，当用户程序调用函数pvPortMalloc申请动态内存时，便从空闲分区链表表头节点开始查找合适大小的内存块(即该空闲内存分区大于或等于所申请内存)，找到则返回该空闲分区存储地址并修改该块“空闲分区节点”内容，判断该块剩余空间是否可以创建“空闲分区节点”，可以则将该块剩余空间划分为新的空闲块并建立新的“空闲分区节点”，最后更新空闲分区链表，FreeRTOS空闲分区节点数据结构如下所示。

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;
    size_t xBlockSize;
}xBlockLink;
```

通过上述分析可以得出FreeRTOS策略二具有下列优点：

(1)根据用户程序申请的每一块动态内存大小建立一个空闲分区节点记录该动态内存的信息，真正意义上实现了动态分配。

(2)用户不必在系统启动时初始化动态内存区，对外接口函数只有pvPortMalloc和vPortFree，很大程度上降低了使用动态内存的难度。

(3)系统采用首次适应算法减小了分配和回收动态内存时的查找时间。

同样，该策略也存在着如下缺点：

(1)每次分配和回收动态内存的时间不固定，即存在着时间不确定性。

(2)如果用户程序需要频繁分配和回收大小不同的动态内存块时，随着系统运行，可能会出现空闲分区链表越来越大，整个动态内存分区会被分割成很多个细小的内存碎片，且每一个内存碎片都附带一个空闲分区节点，造成内存空间的大量浪费。

(3)一旦出现(2)所述的现象，整个系统的稳定性将会降低，甚至会引发系统的崩溃，尽管这种情况并不是开发人员编程逻辑算法错误造成的，但是FreeRTOS并没提供方法或者试图去阻止这种情况发生。

(4)FreeRTOS在回收内存时同样没有检查用户程序所释放内存块的合法性，这是因为FreeRTOS采用从小到大排列空闲分区块，这样便找不到一种很好的算法去判别所释放内存块是否属于动态内存区和该动态内存块结构有没有遭到破坏。

(下转第3页)