第 5 章

CHAPTER 5

FreeRTOS 内存管理

FreeRTOS 内存管理包括内存的申请和释放以及内存分配。

本章通过对 FreeRTOS 内存管理和相关原理的介绍，并以实例开发帮助读者掌握 FreeRTOS 内存的管理。

视频

5.1 内存申请和释放

通过学习本节内容，读者应掌握 FreeRTOS 的内存申请和释放函数的用法。

5.1.1 开发原理

1. FreeRTOS 的内存申请函数

函数原型：

```
void * pvPortMalloc(size_t xWantedSize);
```

功能：从 FreeRTOS 堆栈申请内存。申请成功后会返回所申请内存的首地址；如果失败则会调用内存分配失败回调函数，并返回 NULL。内存申请函数会自动进行字节对齐，在对齐时只会增大申请的内存，不会减小。申请函数自身会消耗 8 字节大小的内存。

参数描述：

xWantedSize——申请的内存大小，单位是字节。函数会以 8 字节大小自动对齐。也就是说，此参数函数会自动计算成 8 字节的倍数。

返回值：

申请成功返回申请的内存首地址，申请失败返回 NULL。

2. FreeRTOS 的内存释放

函数原型：

```
void vPortFree(void * pv);
```

功能：从用户指定的内存地址开始释放内存。

参数描述：

pv——释放的内存首地址。

5.1.2 开发步骤

(1) 在 main.c 中创建一个任务：任务 vTaskGD 完成按键检测和打印任务信息功能。

```
void vTaskGD(void * pvParameters)          // 定义任务
```

```
{
    while(1)
    {
        if(!Key0_Read)
        {
                printf ( " The remaining unused memory in the current system is % d \ n",
xPortGetFreeHeapSize());                    // 打印当前 FreeRTOS 系统未使用的堆栈大小
        }
        if(!Key1_Read)
        {
            Memory_AOA = (uint32_t * )pvPortMalloc(16);     // 申请内存
            // 打印申请指定大小内存后 FreeRTOS 系统未使用的堆栈大小
            printf("The remaining % d bytes of memory in the system\n",xPortGetFreeHeapSize());
            vPortFree(Memory_AOA);                          // 释放内存
        }
        if(!Key2_Read)
        {
            Memory_AOA = (uint32_t * )pvPortMalloc(32);     // 申请内存
            // 打印申请指定大小内存后 FreeRTOS 系统未使用的堆栈大小
            printf("The remaining % d bytes of memory in the system\n",xPortGetFreeHeapSize());
            vPortFree(Memory_AOA);                          // 释放内存
        }
        vTaskDelay(10/portTICK_PERIOD_MS);
    }
}
```

(2) 定义任务创建函数,代码如下:

```
void Task_Cerate(void)
{
    xTaskCreate(vTaskGD,                                    // 任务指针
                "vTaskGD",                                  // 任务描述
                100,                                        // 堆栈深度
                NULL,                                       // 给任务传递的参数
                3,                                          // 任务优先级
                &TaskGD_Handle                              // 任务句柄
                );
}
```

(3) 在 main()函数中调用任务创建函数和启动调度器函数,代码如下:

```
# include "FreeRTOS.h"
# include "task.h"
# include "bsp_clock.h"
# include "Bsp_Key.h"
# include "bsp_uart.h"

uint32_t * Memory_AOA;                                      // 存储申请内存的首地址

TaskHandle_t TaskGD_Handle;                                 // 定义任务句柄

void vTaskGD(void * pvParameters);                          // 声明任务
void Task_Cerate(void);

int main(void)
{
    CLOCLK_Init();
```

```
    Key_Init();
    UART1_Init();
    Task_Cerate();
    vTaskStartScheduler();                                    // 启动调度器函数
    while(1);
}
```

5.1.3 运行结果

下载程序,按下 KEY0 键后通过串口打印出当前 FreeRTOS 系统剩余未使用的堆栈大小。按下 KEY1 键后通过串口打印出申请 16 字节内存后 FreeRTOS 系统剩余未使用的堆栈大小。按下 KEY2 键后通过串口打印出申请 32 字节内存后 FreeRTOS 系统剩余未使用的堆栈大小。

练习

(1) 简述内存申请函数的功能。

(2) 如果每次申请完内存不释放会出现什么结果?


视频

5.2 内存分配

通过学习本节内容,读者应掌握 FreeRTOS 内存分配原理。

1. 动态内存管理介绍

FreeRTOS 操作系统将内核与内存管理分开实现。操作系统内核仅规定了必要的内存管理函数原型,而不关心这些内存管理函数是如何实现的。这样做大有好处——可以增加系统的灵活性,不同的应用场合可以采用不同的内存分配实现,选择对自己更有利的内存管理策略。例如,对于安全型的嵌入式系统,通常不允许动态内存分配,那么可以采用非常简单的内存管理策略,一经申请的内存,甚至不允许被释放。在满足设计要求的前提下,系统越简单越容易做得更安全。再例如一些复杂应用,要求动态地申请、释放内存操作,那么也可以设计出相对复杂的内存管理策略,允许动态分配和动态释放。

FreeRTOS 内核规定的几个内存管理函数原型如下:

```
void * pvPortMalloc(size_t xSize)                    // 内存申请函数
void vPortFree(void * pv)                            // 内存释放函数
void vPortInitialiseBlocks(void)                     // 初始化内存堆函数
size_t xPortGetFreeHeapSize(void)                    // 获取当前未分配的内存堆大小
size_t xPortGetMinimumEverFreeHeapSize(void)         // 获取未分配的内存堆历史最小值
```

FreeRTOS 提供了 5 种内存管理方案,有简单的也有复杂的,可以应用于绝大多数场合。它们位于下载包目录...\FreeRTOS\Source\portable\MemMang 中,文件名分别为 heap_1.c、heap_2.c、heap_3.c、heap_4.c、heap_5.c。FreeRTOS 提供的内存管理都是基于内存堆进行的。默认情况下,FreeRTOS 内核创建任务、队列、信号量、事件组、软件定时器都是借助内存管理函数从内存堆中分配内存。FreeRTOS v9.0.0 以上的版本可以完全使用静态内存分配方法,也就是不使用任何内存堆。对于 heap_1.c、heap_2.c 和 heap_4.c 这 3 种内存管理方案,内存堆实际上是一个很大的数组,定义为:

```
static uint8_t ucHeap[configTOTAL_HEAP_SIZE];
```

其中，宏 configTOTAL_HEAP_SIZE 用来定义内存堆的大小，这个宏在 FreeRTOSConfig.h 中设置。

heap_3.c 只是简单地包装了标准库中的 malloc()和 free()函数，包装后的 malloc()和 free()函数具备线程保护。因此，内存堆需要通过编译器或者启动文件设置堆空间。

heap_5.c 比较有趣，它允许程序设置多个非连续内存堆，例如，需要快速访问的内存堆设置在片内 RAM，稍微慢速访问的内存堆设置在外部 RAM。每个内存堆的起始地址和大小由应用程序设计者定义。

2. 动态内存管理方案 1——heap_1

这是所有实现中最简单的一个。一旦分配内存之后，它甚至不允许释放分配的内存。尽管这样，heap_1.c 还是适用于大部分嵌入式应用程序。这是因为大多数深度嵌入式(deeply embedded)应用只是在系统启动时创建所有任务、队列、信号量等，并且直到程序结束都会一直使用它们，永远不需要删除。当需要分配 RAM 时，这个内存分配方案只是简单地将一个大数组细分出一个子集来。大数组的容量大小通过 FreeRTOSConfig.h 文件中的 configTOTAL_HEAP_SIZE 宏来设置。API 函数 xPortGetFreeHeapSize()返回未分配的堆栈空间总大小，可以通过这个函数返回值对 configTOTAL_HEAP_SIZE 进行合理的设置。

heap_1 功能简介：

(1) 用于从不会删除任务、队列、信号量、互斥量等的应用程序(实际上大多数使用 FreeRTOS 的应用程序都符合这个条件)。

(2) 执行时间是确定的并且不会产生内存碎片。

(3) 实现和分配过程非常简单，需要的内存是从一个静态数组中分配的，意味着这种内存分配通常只是适用于那些不进行动态内存分配的应用。

3. 动态内存管理方案 2——heap_2

这个内存分配方案使用一个最佳匹配算法，它允许释放之前分配的内存块。它不会把相邻的空闲块合成一个更大的块(换句话说，这会造成内存碎片)。有效的堆栈空间大小由位于 FreeRTOSConfig.h 文件中的 configTOTAL_HEAP_SIZE 宏来定义。API 函数 xPortGetFreeHeapSize()返回剩下的未分配堆栈空间的大小(可用于优化设置 configTOTAL_HEAP_SIZE 宏的值)，但是不能提供未分配内存的碎片细节信息。

heap_2 功能简介：

(1) 可以用于重复地分配和删除具有相同堆栈空间的任务、队列、信号量、互斥量等，并且不考虑内存碎片的应用程序。

(2) 不能用在分配和释放随机字节堆栈空间的应用程序。

(3) 如果一个应用程序动态地创建和删除任务，并且分配给任务的堆栈空间总是同样大小，那么大多数情况下 heap_2.c 是可以使用的。但是，如果分配给任务的堆栈不总是相等，那么释放的有效内存可能碎片化，形成很多小的内存块。最后会因为没有足够大的连续堆栈空间而造成内存分配失败。应用程序直接调用 pvPortMalloc() 和 vPortFree()函数，而不仅是通过 FreeRTOS API 间接调用。

(4) 如果应用程序中的队列、任务、信号量、互斥量等以人无法预测的顺序存在，则可能会导致内存碎片问题，虽然这是小概率事件，但必须牢记。不具有确定性，但是它比标准库中的 malloc()函数具有高得多的效率。heap_2.c 适用于需要动态创建任务的大多数小型实时(small real time)系统。

4. 动态内存管理方案 3——heap_3

heap_3.c 简单地包装了标准库中的 malloc()和 free()函数,包装后的 malloc()和 free()函数具备线程保护功能。

heap_3.c 功能简介:

(1) 需要链接器设置一个堆栈,并且编译器库提供 malloc()和 free()函数。

(2) 不具有确定性。

(3) 可能明显地增大 RTOS 内核的代码大小。

(4) 注意,使用 heap_3 时,FreeRTOSConfig.h 文件中的 configTOTAL_HEAP_SIZE 宏定义没有作用。

5. 动态内存管理方案 4——heap_4

这个方案使用一个最佳匹配算法。和动态内存管理方案 2 不同的是,它会将相邻的空闲内存块合并成一个更大的块(包含一个合并算法)。有效的堆栈空间大小由位于 FreeRTOSConfig.h 文件中的 configTOTAL_HEAP_SIZE 来定义。API 函数 xPortGetFreeHeapSize()返回剩下的未分配堆栈空间的大小(可用于优化设置 configTOTAL_HEAP_SIZE 宏的值),但是不能提供未分配内存的碎片细节信息。

heap_4.c 功能简介:

(1) 可用于重复分配、删除任务、队列、信号量、互斥量等的应用程序。

(2) 可以用于分配和释放随机字节内存的情况,并不像 heap_2.c 那样产生严重碎片。

(3) 不具有确定性,但是它比标准库中的 malloc()函数具有高得多的效率。

(4) heap_4.c 还特别适用于移植层代码,可以直接使用 pvPortMalloc()和 vPortFree()函数来分配和释放内存。

6. 动态内存管理方案 5——heap_5

有时候我们希望 FreeRTOSConfig.h 文件中定义的 heap 空间可以采用不连续的内存区,例如,我们希望可以将其定义在内部 SRAM 一部分、外部 SRAM 一部分,此时就可以采用 heap_5 动态内存管理方式。这个方案同样实现了动态内存管理方案 4 中的合并算法,并且允许堆栈跨越多个非连续的内存区。heap_5 通过调用 vPortDefineHeapRegions()函数实现初始化,在内存初始化函数执行完成前不允许使用内存分配和释放。创建 RTOS 对象(任务、队列、信号量等)会隐含地调用 pvPortMalloc(),因此必须注意,使用 heap_5 创建任何对象前,要先执行 vPortDefineHeapRegions()函数。

5 种动态内存管理方案总结。

(1) heap_1:5 种方式里面最简单的,但是申请的内存不允许释放。

(2) heap_2:支持动态内存的申请和释放,但是不支持内存碎片的处理,并将其合并成一个大的内存块。

(3) heap_3:将编译器自带的 malloc()和 free()函数进行简单的封装,以支持线程安全,即支持多任务调用。

(4) heap_4:支持动态内存的申请和释放,支持内存碎片处理,支持将动态内存设置在一个固定的地址。

(5) heap_5:在 heap_4 的基础上支持将动态内存设置在不连续的区域上。