外文资料

An automatic driving system which is provided for automatically driving vehiclesto follow respective target running trajectories. A running instruction and a runningcourse are set based on an obstacle detecting signal, a vehicle position signal, road data and automatic drive traffic information signals, and a vehicle-to-vehicle running information signal. A running course signal indicative of the running course and a runningposition setting position signal indicative of a running position of one vehicle at the timethe running course is set are transmitted and received between the one vehicle andanother vehicle, so that a target running trajectory suitable for each vehicle can becalculated on the basis of the received running course, thereby automatically navigatingeach vehicle such that the respective vehicles run following the resultant target runningtrajectories.

A digital road map provides partial knowledge about the operating environment for a road vehicle. If a road vehicle is equipped with a video camera, machine vision approaches can provide knowledge about the actual traffic environment around the vehicle. Experiences with a combination of two such approaches during the commissioning of a van for automatic driving on a private road network are reported, including experiences gathered during subsequent driving experiments on public roads and several improvement cycles for hardware and software. Based on these experiences, a second generation vehicle for automatic driving-a sedan-has been designed and commissioned. It is currently evaluated on public roads.

Cybercars and dual mode vehicles are presently the most innovative testbeds for vehicular automation applications. The definition of standards and control architectures of the different automatic vehicle onboard systems is a necessary task to build a final prototype to be produced. Several classical architecture definitions have been made in the field of mobile robotics. These architectures are capable of dealing with sensorial inputs and environment and procedural knowledge to manage the different actuators of mobile robots in order to accomplish their missions. Autonomous vehicles are conceived as a link between mobile robotics and the field of vehicular technology, obtaining cars that may be as autonomous as a mobile robot but circulating in high demand environments and in different conditions, as compared to robots. In this paper we present the control architecture used in AUTOPIA program, used for automating mass produced cars. This architecture is to deal with sensorial information and wireless communication as main sensorial input and manages the three fundamental actuators in a car: throttle, brake and steering wheel. The final aim of this architecture is to cover an automatic driving system that can manage a set of maneuvers of a car in the same way human drivers do. At this moment, straight circulation, curve circulation, adaptive cruise control, stop and go and overtaking maneuvers are available and research continues in order to increment its number.

The Order in Which Functions Appear in This Manual

Within this document, the API functions have been split into five groups – task and scheduler related functions, queue related functions, semaphore related functions, software timer related functions and event group related functions. Each group is documented in its own chapter, and within each chapter, the API functions are listed in alphabetical order. Note however that the name of each API function is prefixed with one or more letters that specify the function’s return type, and the alphabetical ordering of API functions within each chapter ignores the function return type prefix. APPENDIX 1: describes the prefixes in more detail.

As an example, consider the API function that is used to create a FreeRTOS task. Its name is xTaskCreate(). The ‘x’ prefix specifies that xTaskCreate() returns a non-standard type. The secondary ‘Task’ prefix specifies that the function is a task related function, and, as such, will be documented in the chapter that contains task and scheduler related functions. The ‘x’ isnot considered in the alphabetical ordering, so xTaskCreate() will appear in the task andscheduler chapter ordered as if its name was just TaskCreate().

Define a set of Memory Protection Unit (MPU) regions for use by an MPU restricted task.This function is intended for advanced users only and is only relevant to FreeRTOS MPU ports(FreeRTOS ports that make use of a Memory Protection Unit).

MPU controlled memory regions can be assigned to an MPU restricted task when the task is created using the xTaskCreateRestricted() function. They can then be redefined (or reassigned) at run time using the vTaskAllocateMPURegions() function.

Definition: FreeRTOS Port

FreeRTOS can be built with approximately twenty different compilers, and can run on more than thirty different processor architectures. Each supported combination of compiler and processor is considered to be a separate FreeRTOS port.

Building FreeRTOS

FreeRTOS can be thought of as a library that provides multi-tasking capabilities to what would otherwise be a bare metal application.

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port. Build the source files as part of your project to make the FreeRTOS API available to your application. To make this easy for you, each official FreeRTOS port is provided with a demo application. The demo application is pre-configured to build the correct source files, and include the correct header files.

Demo applications should build ‘out of the box’, although some demos are older than others, and sometimes a change in the build tools made since the demo was released can cause an issue. Section 1.3 describes the demo applications.

FreeRTOSConfig.h

FreeRTOS is configured by a header file called FreeRTOSConfig.h.

FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example, FreeRTOSConfig.h contains constants such as configUSE\_PREEMPTION, the setting of which defines whether the co-operative or pre-emptive scheduling algorithm will be used1. As FreeRTOSConfig.h contains application specific definitions, it should be located in a directory that is part of the application being built, not in a directory that contains the FreeRTOS source code.

A demo application is provided for every FreeRTOS port, and every demo application contains a FreeRTOSConfig.h file. It is therefore never necessary to create a FreeRTOSConfig.h file from scratch. Instead, it is recommended to start with, then adapt, the FreeRTOSConfig.h used by the demo application provided for the FreeRTOS port in use.

The Official FreeRTOS Distribution

FreeRTOS is distributed in a single zip file. The zip file contains source code for all the FreeRTOS ports, and project files for all the FreeRTOS demo applications. It also contains a selection of FreeRTOS+ ecosystem components, and a selection of FreeRTOS+ ecosystem demo applications.

Do not be put off by the number of files in the FreeRTOS distribution! Only a very small number of files are required in any one application.

The Top Directories in the FreeRTOS Distribution

The first and second level directories of the FreeRTOS distribution are shown and described .

The zip file only contains one copy of the FreeRTOS source files; all the FreeRTOS demo projects, and all the FreeRTOS+ demo projects, expect to find the FreeRTOS source files in the FreeRTOS/Source directory, and may not build if the directory structure is changed.

FreeRTOS Source Files Common to All Ports

The core FreeRTOS source code is contained in just two C files that are common to all the FreeRTOS ports. These are called tasks.c, and list.c, and they are located directly in the FreeRTOS/Source directory, as shown in Figure 2. In addition to these two files, the following source files are located in the same directory:

queue.c provides both queue and semaphore services, as described later in this book.queue.c is nearly always required.

timers.c provides software timer functionality, as described later in this book. It need only be included in the build if software timers are actually going to be used.

event\_groups.c provides event group functionality, as described later in this book. It need only be included in the build if event groups are actually going to be used.

croutine.c implements the FreeRTOS co-routine functionality. It need only be included in the build if co-routines are actually going to be used. Co-routines were intended for use on very small microcontrollers, are rarely used now, and are therefore not maintained to the same level as other FreeRTOS features. Co-routines are not described in this book.

It is recognized that the file names may result in name space clashes, as many projects will already include files that have the same names. It is however considered that changing the names of the files now would be problematic, as to do so would break compatibility with the many thousands of projects that use FreeRTOS, as well as automation tools, and IDE plugins.

A control method for passing through an intersection for automated vehicles is proposed. If automated vehicles approaching an intersection communicate with each other and schedule the time of entering the intersection by small deceleration and acceleration, the vehicles can pass the through the intersection without stopping thereby enabling energy-saving by reducing unnecessary deceleration. This study proposes a method of categorizing intersection patterns, and extracting the required conditions for the realization of a non-stop intersection for each intersection pattern, performing control to realize such an intersection and transmitting information to vehicles entering the intersection for the creation of a virtual platoon. The proposed method is validated by experiments at an intersection of one-way traffic in single lanes.

Autonomous following of ill‐defined roads is an important part of visual navigation systems. This paper presents an adaptive method that uses a statistical model of the color of the road surface within a trapezoidal shape that approximately corresponds to the projection of the road on the image plane. The method does not perform an explicit segmentation of the images but instead expands the shape sideways until the match between shape and road worsens, simultaneously computing the color statistics. Results show that the method is capable of reactively following roads, at driving speeds typical of the robots used, in a variety of situations while coping with variable conditions of the road such as surface type, puddles, and shadows. We extensively evaluate the proposed method using a large number of datasets with ground truth . We moreover evaluate many color spaces in the context of road following, and we find that the color spaces that separate luminance from color information perform best, especially if the luminance information is discarded.

FreeRTOS Source Files Specific to a Port

Source files specific to a FreeRTOS port are contained within the FreeRTOS/Source/portable directory. The portable directory is arranged as a hierarchy, first by compiler, then by processor architecture. This is shown in Figure 3.

If you are running FreeRTOS on a processor with architecture ‘architecture’ using compiler ‘compiler’ then, in addition to the core FreeRTOS source files, you must also build the files located in FreeRTOS/Source/portable/[compiler]/[architecture] directory.

From FreeRTOS V9.0.0 FreeRTOS applications can be completely statically allocated, removing the need to include a heap memory manager: As will be described in Chapter 2, Heap Memory Management, FreeRTOS also considers heap memory allocation to be part of the portable layer. FreeRTOS provides five example heap allocation schemes. The five schemes are named heap\_1 to heap\_5, and are implemented by the source files heap\_1.c to heap\_5.c respectively. The example heap allocation schemes are contained in the FreeRTOS/Source/portable/MemMang directory. It is necessary to build one of these five source files in your project, unless yourapplication provides an alternative implementation.

Dynamic Memory Allocation and its Relevance to FreeRTOS

From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time:

Following chapters of this book will introduce kernel objects such as tasks, queues, semaphores and event groups. To make FreeRTOS as easy to use as possible, these kernel objects are not statically allocated at compile-time, but dynamically allocated at run-time; FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is deleted. This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

This chapter discusses dynamic memory allocation. Dynamic memory allocation is a C programming concept, and not a concept that is specific to either FreeRTOS or multitasking. It is relevant to FreeRTOS because kernel objects are allocated dynamically, and the dynamic memory allocation schemes provided by general purpose compilers are not always suitable for real-time applications.

Options for Dynamic Memory Allocation

From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time:

Early versions of FreeRTOS used a memory pools allocation scheme, whereby pools of different size memory blocks were pre-allocated at compile time, then returned by the memory allocation functions. Although this is a common scheme to use in real-time systems, it proved to be the source of many support requests, predominantly because it could not use RAM efficiently enough to make it viable for really small embedded systems—so the scheme was dropped.

FreeRTOS now treats memory allocation as part of the portable layer (as opposed to part of the core code base). This is in recognition of the fact that different embedded systems have varying dynamic memory allocation and timing requirements, so a single dynamic memory allocation algorithm will only ever be appropriate for a subset of applications. Also, removing dynamic memory allocation from the core code base enables application writer’s to provide their own specific implementations, when appropriate.

When FreeRTOS requires RAM, instead of calling malloc(), it calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the kernel calls vPortFree(). pvPortMalloc() has the same prototype as the standard C library malloc() function, and vPortFree() has the same prototype as the standard C library free() function.

pvPortMalloc() and vPortFree() are public functions, so can also be called from application code.

From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time: FreeRTOS comes with five example implementations of both pvPortMalloc() and vPortFree(), all of which are documented in this chapter. FreeRTOS applications can use one of the example implementations, or provide their own. The five examples are defined in the heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c and heap\_5.c source files respectively, all of which are located in the FreeRTOS/Source/portable/MemMang directory.

Heap\_1

It is common for small dedicated embedded systems to only create tasks and other kernel objects before the scheduler has been started. When this is the case, memory only gets dynamically allocated by the kernel before the application starts to perform any real-time functionality, and the memory remains allocated for the lifetime of the application. This means the chosen allocation scheme does not have to consider any of the more complex memory allocation issues, such as determinism and fragmentation, and can instead just consider attributes such as code size and simplicity.

Heap\_1.c implements a very basic version of pvPortMalloc(), and does not implement vPortFree(). Applications that never delete a task, or other kernel object, have the potential to use heap\_1. Some commercially critical and safety critical systems that would otherwise prohibit the use of dynamic memory allocation also have the potential to use heap\_1. Critical systems often prohibit dynamic memory allocation because of the uncertainties associated with nondeterminism, memory fragmentation, and failed allocations—but Heap\_1 is always deterministic, and cannot fragment memory.

The heap\_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made. The array is called the FreeRTOS heap. The total size (in bytes) of the array is set by the definition configTOTAL\_HEAP\_SIZE within FreeRTOSConfig.h. Defining a large array in this manner can make the application appear to consume a lot of RAM—even before any memory has been allocated from the array.

Each created task requires a task control block (TCB) and a stack to be allocated from the heap. Figure 5 demonstrates how heap\_1 subdivides the simple array as tasks are created.

外文资料翻译

一种自动驾驶系统，用于自动驾驶车辆跟随各自的目标运行轨迹。基于障碍物检测信号、车辆位置信号、道路数据和自动驾驶交通信息信号以及车对车运行信息信号设置运行指令和运行路线。在一辆车和另一辆车之间传输和接收指示行驶路线的行驶路线信号和指示设置行驶路线时一辆车的行驶位置的行驶位置设置位置信号，以便可以根据接收到的行驶路线计算适合每辆车的目标行驶轨迹，从而自动导航每辆车，使相应的车辆按照生成的目标运行轨迹运行。

数字路线图提供了有关道路车辆运行环境的部分知识。如果道路车辆配备了摄像机，机器视觉方法可以提供有关车辆周围实际交通环境的知识。报告了在私人道路网络上自动驾驶面包车调试期间结合两种方法的经验，包括在随后的公共道路驾驶试验中收集的经验，以及硬件和软件的几个改进周期。基于这些经验，设计并调试了第二代自动驾驶汽车——轿车。目前在公共道路上对其进行评估。

赛博汽车和双模车辆是目前最具创新性的车辆自动化应用试验台。定义不同自动车载系统的标准和控制架构是构建最终原型的必要任务。在移动机器人领域，已经做出了一些经典的架构定义。这些体系结构能够处理传感器输入、环境和程序知识，以管理移动机器人的不同执行器，从而完成其任务。自动车辆被认为是移动机器人技术和车辆技术领域之间的一个纽带，可以获得与移动机器人一样具有自主性，但与机器人相比，在高需求环境和不同条件下流通的汽车。在本文中，我们介绍了AUTOPIA程序中使用的控制体系结构，用于批量生产汽车的自动化。该体系结构将传感器信息和无线通信作为主要的传感器输入进行处理，并管理汽车中的三个基本执行器：油门、制动器和方向盘。该体系结构的最终目标是覆盖一个自动驾驶系统，该系统可以像人类驾驶员那样管理汽车的一系列操纵。目前，直行循环、曲线循环、自适应巡航控制、停行和超车操作都可用，研究仍在继续，以增加其数量。

提出了一种自动车辆通过交叉口的控制方法。如果接近交叉口的自动车辆相互通信，并通过较小的减速和加速来安排进入交叉口的时间，车辆可以通过交叉口而不停车，从而通过减少不必要的减速来实现节能。本研究提出了一种对交叉口模式进行分类的方法，为每个交叉口模式提取实现不停车交叉口所需的条件，执行控制以实现此类交叉口，并将信息传输给进入交叉口的车辆，以创建虚拟排。通过单车道单向交通交叉口的试验验证了该方法的有效性。

模糊道路的自动跟踪是视觉导航系统的重要组成部分。本文提出了一种自适应方法，该方法使用梯形形状内路面颜色的统计模型，该形状近似于道路在图像平面上的投影。该方法不执行图像的显式分割，而是横向扩展形状，直到形状和道路之间的匹配恶化，同时计算颜色统计。结果表明，该方法能够在各种情况下以所用机器人的典型行驶速度反应性地跟踪道路，同时应对道路的各种条件，如路面类型、水坑和阴影。我们使用大量具有地面真实性的数据集对所提出的方法进行了广泛的评估。此外，我们在道路跟踪的背景下评估了许多颜色空间，发现将亮度与颜色信息分离的颜色空间表现最好，尤其是在亮度信息被丢弃的情况下。

在本文档中，API 函数分为五组——任务和调度程序相关函数、队列相关函数、信号量相关函数、软件定时器相关函数和事件组相关函数。每个组都记录在自己的章节中，在每一章中，API 函数按字母顺序列出。但请注意每个 API 函数的名称都以一个或多个字母为前缀，指定函数的返回类型，并且每章中 API 函数的字母顺序忽略了函数返回类型前缀。附录 1：更详细地描述了前缀。例如，考虑用于创建 FreeRTOS 任务的 API 函数。它的名字是xTaskCreate()。 “x”前缀指定 xTaskCreate() 返回非标准类型。这辅助“任务”前缀指定该函数是任务相关函数，因此，将记录在包含任务和调度程序相关功能的章节中。 “x”是不考虑字母顺序，因此 xTaskCreate() 将出现在任务中调度程序章节的排序就好像它的名字只是 TaskCreate()。

定义一组内存保护单元 (MPU) 区域以供 MPU 受限任务使用。此功能仅供高级用户使用，仅与 FreeRTOS MPU 端口相关（使用内存保护单元的 FreeRTOS 端口）。

MPU 控制的内存区域可以分配给 MPU 受限任务，当该任务是使用 xTaskCreateRestricted() 函数创建。然后可以重新定义它们（或重新分配）在运行时使用 vTaskAllocateMPURegions() 函数。

定义：FreeRTOS 端口

FreeRTOS可以使用大约20种不同的编译器来构建，并且可以在30多种不同的处理器架构上运行。每个受支持的编译器和处理器的组合都被认为是一个单独的FreeRTOS端口。

建立自由RTOS

FreeRTOS可以被认为是一个库，它为裸金属应用程序提供了多任务功能。

FreeRTOS是作为一组C源文件提供的。有些源文件对所有端口都是通用的，而其他文件是特定于端口的。将源文件作为项目的一部分来构建，以便使应用程序可以使用FreeRTOSAPI。为了方便您做到这一点，每个官方的FreeRTOS端口都提供了一个演示应用程序。演示应用程序被预配置为构建正确的源文件，并包含正确的头文件。

演示应用程序应该“开箱即用”地构建，尽管一些演示比其他的更老，有时自演示发布以来所进行的构建工具的更改可能会导致问题。第1.3节描述了演示应用程序。FreeRTOSConfig.h

FreeRTOS是由一个名为FreeRTOSConfig.h的头文件配置的。

FreeRTOSConfig.h用于定制FreeRTOS。例如，FreeRTOSConfig.h包含像configUSE\_PREEMPTION这样的常量，它的设置定义了是使用合作调度算法还是先发制人调度算法1。由于FreeRTOSConfig.h包含特定于应用程序的定义，因此它应该位于正在构建的应用程序的一部分的目录中，而不是位于包含FreeRTOS源代码的目录中。

为每个FreeRTOS端口提供了一个演示应用程序，并且每个演示应用程序都包含一个FreeRTOSConfig.h文件。因此，从来没有必要从头开始创建一个FreeRTOSConfig.h文件。相反，建议从为正在使用的FreeRTOS端口提供的演示应用程序所使用的FreeRTOSConfig.h开始，然后进行调整。

官方免费提供服务和操作系统发行版

FreeRTOS分布在单个zip文件中。zip文件包含所有FreeRTOS端口的源代码，以及所有FreeRTOS演示应用程序的项目文件。它还包含了FreeRTOS+生态系统组件，以及FreeRTOS+生态系统演示应用程序。

不要被FreeRTOS发行版中的文件数量所推迟！在任何一个应用程序中，只需要非常少量的文件。

FreeRTOS发行版中的顶级目录

FreeRTOS发行版的第一级和第二级目录显示和描述.

zip文件只包含FreeRTOS源文件的一个副本；所有FreeRTOS演示项目和所有FreeRTOS+演示项目都希望在FreeRTOS/源目录中找到FreeRTOS源文件，如果目录结构发生更改，可能不会构建。

所有端口共有的免费RTOS源文件

核心的FreeRTOS源代码仅包含在两个C文件中，这是所有FReeRTOS端口通用的。这些任务被称为tasks.c和list.c，它们直接位于FreeRTOS/Source目录中，除这两个文件外，以下源文件还位于同一目录中：

quue.c同时提供队列和信号服务，正如后面描述的，book.queue.c几乎总是必需的。

计时器。c提供了软件计时器功能，如本书后面所述。只有要使用软件计时器，它才需要包含在构建中。

event\_groups.c提供了事件组功能，如本书后面所述。只有在实际上要使用事件组时，才需要将其包含在构建中。

croutine.c实现了FreeRTOS的协同例程功能。只有在实际上要使用协同例程时，它才只需要包含在构建中。协同例程原本打算用于非常小的微控制器，现在很少使用，因此没有维护到与其他FreeRTOS特性相同的级别。在这本书中没有描述共同例程。

我们可以认识到，文件名可能会导致名称空间冲突，因为许多项目将已经包含了具有相同名称的文件。然而，人们认为现在更改文件的名称是有问题的，因为这样做会破坏与数千个使用FreeRTOS、自动化工具和IDE插件的项目的兼容性。

特定于端口的FreeRTOS源文件

特定于FreeRTOS端口的源文件包含在FreeRTOS/源文件/可移植目录中。可移植目录被安排为一个层次结构，首先由编译器，然后由处理器架构。这一点如图3所示。

如果您在使用编译器“编译器”的具有架构“架构”的处理器上运行FreeRTOS，那么除了核心的FreeRTOS源文件外，您还必须构建位于FreeRTOS/Source/portable/[compiler]/[architecture]目录中的文件。

从FreeRTOSV9.0.0中，FreeRTOS应用程序可以完全静态分配，消除了包含堆内存管理器的需要：正如将在第2章堆内存管理中描述的，FreeRTOS还将堆内存分配视为可移植层的一部分。FreeRTOS提供了5个堆分配方案示例。这五种方案分别命名为heap\_1到heap\_5，并分别由源文件heap\_1.c到heap\_5.c实现。堆分配方案包含在FreeRTOS/源/可移植/MemMang目录中。必须在项目中构建这五个源文件中的一个，除非您的应用程序中提供了一个替代实现。

动态内存分配及其与freertos的相关性

从FreeRTOSV9.0.0中，内核对象可以在编译时静态分配，或在运行时动态分配：

本书的以下章节将介绍内核对象，如任务、队列、信号量和事件组。为了使FreeRTOS尽可能容易使用，这些内核对象不是在编译时静态分配，而是在运行时动态分配；FreeRTOS在每次创建内核对象时分配RAM，并在每次删除内核对象时释放RAM。此策略减少了设计和规划工作，简化了API，并最小化了RAM占用。

本章讨论了动态内存分配。动态内存分配是一个C语言编程的概念，而不是一个特定于FreeRTOS或多任务处理的概念。它与FreeRTOS相关，因为内核对象是动态分配的，而且由通用编译器提供的动态内存分配方案并不总是适合于实时应用程序。

用于动态内存分配的选项

从FreeRTOSV9.0.0中，内核对象可以在编译时静态分配，或在运行时动态分配：

FreeRTOS的早期版本使用了一种内存池分配方案，即在编译时预先分配不同大小的内存块的池，然后由内存分配函数返回。尽管这是在实时系统中使用的一种常见方案，但它被证明是许多支持请求的来源，主要是因为它不能足够有效地使用RAM，使其适用于非常小的嵌入式系统——因此该方案被放弃了。

FreeRTOS现在将内存分配视为可移植层的一部分（而不是核心代码库的一部分）。这是为了认识到不同的嵌入式系统具有不同的动态内存分配和时间要求，因此一个单一的动态内存分配算法将永远只适用于应用程序的一个子集。此外，从核心代码库中删除动态内存分配可以使应用程序编写器能够在适当的时候提供它们自己的特定实现。

当FreeRTOS需要RAM时，它不是调用Malloc()，而是调用pvportMalloc()。当释放RAM时，内核不是调用免费的()，而是调用vPortFree()。pvPortMalloc()与标准C库malloc()函数具有相同的原型，而vPortFree()与标准C库free()函数具有相同的原型。

pvPortMalloc()和vPortFree()是公共功能，所以也可以从应用程序代码中调用。

从FreeRTOSV9.0.0中，内核对象可以在编译时静态分配，或者在运行时动态分配：FreeRTOS提供了五个pv()Malloc()和vPortFree()的实现示例，所有这些都在本章中记录。FreeRTOS应用程序可以使用其中一个示例实现，或者提供它们自己的实现。这五个例子分别在heap\_1.c、heap\_2.c、heap\_3.c、heap\_4.c和heap\_5.c源文件中进行了定义，所有这些文件都位于FreeRTOS/源文件/便携式/MemMang目录中。

Heap\_1

对于小型专用嵌入式系统，通常在调度程序启动之前只创建任务和其他内核对象。在这种情况下，只有在应用程序开始执行任何实时功能之前，内核才能动态地分配内存，并且在应用程序的生命周期中仍然分配内存。这意味着所选择的分配方案不必考虑任何更复杂的内存分配问题，如决定论和碎片化，而可以只考虑代码大小和简单性等属性。

Heap\_1.c实现了一个非常基本的()Malloc版本，而没有实现vHeap\_1Free()。从未删除任务或其他内核对象的应用程序有可能使用heap\_1。一些原本会禁止使用动态内存分配的商业关键和安全关键系统也有可能使用heap\_1。关键系统通常禁止动态内存分配，因为不确定性，内存碎片化和分配失败——但Heap\_1总是确定性的，不能将内存碎片分割。

heap\_1分配方案将一个简单的数组细分为更小的块，作为对pvPortMalloc()的调用。该数组被称为FreeRTOS堆。数组的总大小（以字节为单位）由FreeRTOSConfig.h中的定义configTOTAL\_HEAP\_SIZE设置。以这种方式定义一个大数组可以使应用程序似乎消耗大量RAM——甚至在从数组分配任何内存之前。

每个已创建的任务都需要从堆中分配一个任务控制块(TCB)和一个堆栈。图5演示了heap\_1如何在创建任务时细分简单数组。