前言

本文档是www.freertos.org官网的一份备份文档，复制的版本是：**FreeRTOS V8.2.1 - Copyright (C) 2015 Real Time Engineers Ltd.**

复制的主要目的是边看官网文档，边把看的复制下来，并加上自己的稍微翻译，以供自己的学习查找，同时也记录了自己的学习历程吧。

此文档仅供我自己学习使用，不作为任何商业应用。如果本文档侵权了官方文档，我立马停止。

本文很多地方参考了网络里流行的一本电子文档“FreeRTOS实时内核实用指南”，谢谢前辈的很多指点。我建议先看完这本书对FreeRTOS有个整体认识，然后再继续学习。

第一章：FreeRTOS#//本文档主要内容部分

1.1：About FreeRTOS

1.1.1 What is an RTOS/FreeRTOS?

What is An RTOS?

> "Provide a free product that surpasses the quality and service demanded by users of commercial alternatives"

Real Time Engineers Ltd. have been working in close partnership with the world's leading chip companies for more than 12 years to provide you market leading, commercial grade, and completely free high quality RTOS and tools that are free from any IP infringement risk... but what is an RTOS?

This page starts by defining an operating system, then refines this to define a real time operating system (RTOS), then refines this once more to define a real timer kernel (or real time executive).

See also the FAQ item "why an RTOS" for information on when and why it can be useful to use an RTOS in your embedded systems software design.

What is a General Purpose Operating System?

An operating system is a computer program that supports a computer's basic functions, and provides services to other programs (or applications) that run on the computer. The applications provide the functionality that the user of the computer wants or needs. The services provided by the operating system make writing the applications faster, simpler, and more maintainable. If you are reading this web page, then you are using a web browser (the application program that provides the functionality you are interested in), which will itself be running in an environment provided by an operating system.

What is an RTOS?

Most operating systems appear to allow multiple programs to execute at the same time. This is called multi-tasking. In reality, each processor core can only be running a single thread of execution at any given point in time. A part of the operating system called the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program. The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi user operating system (such as Unix) will ensure each user gets a fair amount of the processing time. As another example, the scheduler in a desk top operating system (such as Windows) will try and ensure the computer remains responsive to its user. [Note: FreeRTOS is not a big operating system, nor is it designed to run on a desktop computer class processor, I use these examples purely because they are systems readers will be familiar with]

The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as deterministic) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline). A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted (and is therefore deterministic).

Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. In FreeRTOS, a thread of execution is called a task.

What is FreeRTOS?

[see also "more about FreeRTOS"] FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications.

A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically the program is executed directly from the read only memory.

Microcontrollers are used in deeply embedded applications (those applications where you never actually see the processors themselves, or the software they are running) that normally have a very specific and dedicated job to do. The size constraints, and dedicated end application nature, rarely warrant the use of a full RTOS implementation - or indeed make the use of a full RTOS implementation possible. FreeRTOS therefore provides the core real time scheduling functionality, inter-task communication, timing and synchronisation primitives only. This means it is more accurately described as a real time kernel, or real time executive. Additional functionality, such as a command console interface, or networking stacks, can be then be included with add-on components.

1.1.2 A Compelling Free Solution

About Real Time Engineers ltd.

> "Provide a free product that surpasses the quality and service demanded by users of commercial alternatives"

FreeRTOS is solely owned, run, developed and maintained by Real Time Engineers Ltd.

Real Time Engineers Ltd. have been working in close partnership with the world's leading chip companies for more than 12 years to provide you award winning, commercial grade, and completely free high quality software that is free from any IP infringement risk.

Corporate and product summary presentation

About FreeRTOS

[Also see the "What is an RTOS?" and "Why use an RTOS" pages] FreeRTOSTM is a market leading RTOS from Real Time Engineers Ltd. that supports 35 architectures and received >113000 downloads during 2014. It is professionally developed, strictly quality controlled, robust, supported, and free to embed in commercial products without any requirement to expose your proprietary source code.

FreeRTOS has become the de facto standard RTOS for microcontrollers by removing common objections to using free software, and in so doing, providing a truly compelling free software model.

Value Proposition

1. High quality C source code under strict configuration management
2. Safety critical version ensures dependability
3. Cross platform support secures time investment
4. Tutorial books and training to educate engineers
5. Pre-configured example projects for all supported ports
6. Free support, quoted as better than some commercial alternatives
7. Large and growing user base and community
8. Peace of mind - low cost commercial options can be taken at any time
9. = A low total cost of ownership, risk free, & compelling solution

RTOS Technology Highlights

1. Pre-emptive scheduling option
2. Easy to use message passing
3. Co-operative scheduling option
4. Round robin with time slicing
5. ROMable
6. Mutexes with priority inheritance
7. 6K to 10K ROM footprint
8. Recursive mutexes
9. Configurable / scalable
10. Binary and counting semaphores
11. Compiler agnostic
12. Very efficient software timers
13. Some ports never completely disable interrupts
14. Easy to use API

Partners ARM connected community RTOS partnerXilinx Microblaze and Zynq partnerRenesas Gold Alliance Partner RTOS partnerPartner of NXP supplying RTOS to all ARM microcontrollersMicrochip premier third party partnerTexas Instruments MCU Developer Network for RTOS on ARM and MSP430 microcontrollersFreescale Alliance Member supplying RTOS for ARM Kinetis and ColdFire microcontrollerInfineon XMC ARM Cortex MicrocontrollersAtmel RTOS partner for SAM3 ARM Cortex Microcontrollers and AVR32Cypress RTOS supplier for ARM Cortex-M3STmicroelectronics RTOS partner for STM32 ARM Cortex microcontrollersFujitsu RTOS partner for Cortex-M3 and FM3 microcontrollersMicrosemi RTOS partner for ARM Cortex-M3 embedded processorsAtollic RTOS partner for all real time embedded systemsIAR Embedded Workbench RTOS partner for all real time embedded systemsARM Keil RTOS partner for real time embedded systemsEmbedded Artists RTOS partner for ARM based microcontroller embedded hardware

Implementation Quality Management

FreeRTOS is very strictly quality managed, not just in software coding standards and look and feel, but also in implementation. For example:

1. FreeRTOS never performs a non-deterministic operation, such as walking a linked list, from inside a critical section or interrupt.
2. We are particularly proud of the efficient software timer implementation that **does not use any CPU time unless a timer actually needs servicing**. Software timers do not contain variables that need to be counted down to zero.
3. Likewise, lists of Blocked (pended) tasks do not require time consuming periodic servicing. The FreeRTOS queue usage model manages to combine simplicity with flexibility (in a tiny code size) - attributes that are normally mutually exclusive.
4. FreeRTOS queues are base primitives on-top of which other communication and synchronisation primitives are build. The code re-use obtained

dramatically reduced overall code size, which in turn **assists testing and helps ensure robustness**.

In addition, the TÜV SÜD certified SIL 3 SafeRTOS real time kernel was originally derived from FreeRTOS, and has undergone the most stringent analysis and test process - the results of which were fed back into the FreeRTOS code base (when commonality still existed).

Mission

> "Provide a free product that surpasses the quality and service demanded by users of commercial alternatives"

The original mission of the FreeRTOS project was to provide a free RTOS solution that was easy to use. That is, easy to build and deploy, on a Windows (or Linux) host computer, without having to figure out which source files are required, which include paths are required, or how to configure the real time debugging environment. This has been achieved through the provision of pre-configured, build-able, example projects for each officially support port.

Naturally, as the FreeRTOS started circa 2003, how these projects are created has evolved for the better, and some original projects remain that don't demonstrate all of the RTOS functionality, or have become stale. However, each project is fully tested before it is added to the FreeRTOS zip file distribution, and many RTOS demo projects undergo active maintenance before each new release. Responding to user feedback, each new demo added to the distribution now also includes a simple "blinky" style getting started configuration to compliment the comprehensive examples.

Design Goals

The primary design goals are: Easy to use Small footprint Robust

FreeRTOS Founder

The FreeRTOS project was founded by Richard Barry. Richard graduated with 1st Class Honours in Computing for Real Time Systems. He's been directly involved in the start up of several companies, primarily working in the industrial automation and aerospace and simulation markets. Richard is currently a director of Real Time Engineers Ltd., owners and maintainers of the FreeRTOS project.

1.1.3 A Better Type of Open Source

Open source software is the source of frequent debate - where often the same pro and con arguments are repeatedly raised. Every effort is made to ensure FreeRTOS is as open and easy to use as possible, so this page is provided to demonstrate how the FreeRTOS licensing model removes the objections people might otherwise have to including open source components in their products.

> Argument Counter-argument when using FreeRTOS

1. "Open source software is badly supported"
2. Real Time Engineers Ltd. directly support FreeRTOS through an active and free support forum. It is also possible to obtain commercial support from a large engineering company - providing choice and complete peace of mind. "Incorporating open source means you risk having to open source your entire application" FreeRTOS is licensed such that only the RTOS kernel is open source. Application code that uses the RTOS kernel can remain closed source and proprietary - provided the functionality it provides is distinct from that provided by the RTOS kernel itself.

3. "Open source software ends up costing much more (the total cost of ownership argument)"

4. FreeRTOS is completely free to download, experiment with and deploy. Each port comes with a pre-configured demo application to ensure you start with a known good and working project that can then be tailored to meet your needs - getting you up and running very quickly. Should at some point you require commercial licensing or support then packages are available at very competitive prices, so you have nothing to loose and everything to gain.

5. "Open source software is badly written"
6. FreeRTOS is commercial grade, stable and reliable. There are even safety critical versions based on it, with improvements from the safety critical certification being fed back into the open source code base (although not the new safety related features). FreeRTOS conforms to a strict coding standard, and a coding philosophy that ensures non-deterministic actions never executes in an interrupt or in a critical section.
7. "Open source code becomes fragmented, with many different versions available"
8. The FreeRTOS release procedure is very tightly controlled with all official ports being updated simultaneously. Current and past releases are available in .zip files. The head revision is available from a publicly accessible SVN repository. Naturally, occasionally errors are made, but these are quickly spotted by the large user base (more than 6000 downloads per month [very conservative figure given]) and are documented as soon as they are brought to our attention.
9. "Use of open source code leaves you at risk of IP infringement"
10. Only code of known origin is included in official versions. If you are still concerned about IP infringement, purchase a commercial license to receive standard indemnification.
11. "Open source projects have no longevity"
12. Neither do some commercial products! Unlike commercial equivalents, the FreeRTOS license allows you to continue to use FreeRTOS software forever, at no cost. FreeRTOS has been around since before 2003, and is still growing!

1.1.4 Features Overview

FreeRTOS Features:

See also

- About FreeRTOS
- The Memory Usage, Boot Times and Context Switch Times FAQ

FreeRTOS is a scale-able real time kernel designed specifically for small embedded systems. Highlights include:

- Free RTOS kernel - preemptive, cooperative and hybrid configuration options.
- The SafeRTOS derivative product provides a high level of confidence in the code integrity. Includes a tickless mode for low power applications.
- Tiny footprint.
- Official support for >30 embedded system architectures (counting ARM7 and ARM Cortex-M3 as one architecture each).
- FreeRTOS-MPU supports the ARM Cortex-M3 Memory Protection Unit (MPU).
- Designed to be small, simple and easy to use. Typically a RTOS kernel binary image will be in the region of 4K to 9K bytes.
- Very portable source code structure, predominantly written in C.
- Supports both real time tasks and co-routines.
- Direct to task notifications, queues, binary semaphores, counting semaphores, recursive semaphores and mutexes for communication and synchronisation between tasks, or between real time tasks and interrupts.
- Innovative event group (or event flag) implementation.
- Mutexes with priority inheritance.
- Efficient software timers.
- Powerful execution trace functionality.
- Stack overflow detection options.
- Pre-configured RTOS demo applications for selected single board computers allowing 'out of the box' operation and fast learning curve.
- Free monitored forum support, or optional commercial support and licensing.
- No software restriction on the number of real time tasks that can be created.

- No software restriction on the number of task priorities that can be used.
- No restrictions imposed on task priority assignment - more than one real time task can be assigned the same priority.
- Free development tools for many supported architectures.
- Free embedded software source code.
- Royalty free.
- Cross development from a standard Windows host.

plus ... some older links

- Embedded Ethernet example applications.
- 'How FreeRTOS.orgTM works': Source code explained.
- Embedded software application design: Tips for using FreeRTOS on small embedded systems.

1.1.5 Coding Standard and Style Guide

1.1.6 License Details

1.1.7 FreeRTOS™ Site Map

1.2 Features / Getting Started...//初步使用认识FreeRTOS

1.2.1 Quick Start Guide + Videos

1.2.2 Tasks & Co-routines

1.2.3 More About Tasks...//关于任务认识

1.2.3.1 Task States//任务状态的转换

A task can exist in one of the following states:

- Running//运行状态

  When a task is actually executing it is said to be in the Running state. It is currently utilising the processor.

- Ready//准备状态

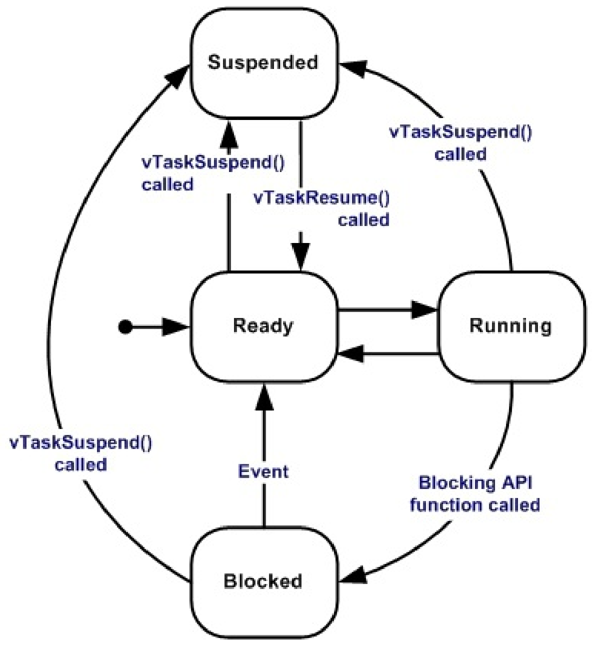
  Ready tasks are those that are able to execute (they are not blocked or suspended) but are not currently executing because a different task of equal or higher priority is already in the Running state.

- Blocked//阻塞状态

  A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block waiting for queue and semaphore events. Tasks in the Blocked state always have a 'timeout' period, after which the task will be unblocked. Blocked tasks are not available for scheduling.

- Suspended//挂起状态

  Tasks in the Suspended state are also not available for scheduling. Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively. A 'timeout' period cannot be specified.



Valid task state transitions//状态转换图

1.2.3.2 Task Priorities//任务优先级

Each task is assigned a priority from 0 to (configMAX*PRIORITIES - 1), where configMAX*PRIORITIES is defined within FreeRTOSConfig.h.

If the port in use implements a port optimised task selection mechanism that uses a 'count leading zeros' type instruction (for task selection in a single instruction) and configUSE*PORT*OPTIMISED*TASKSELECTION is set to 1 in FreeRTOSConfig.h, then configMAXPRIORITIES **cannot be higher than 32**. In all other cases configMAX*PRIORITIES can take any value within reason - but for reasons of RAM usage efficiency should be kept to the minimum value actually necessary.

Low priority numbers denote low priority tasks. The idle task has priority zero (tskIDLE_PRIORITY).

The FreeRTOS scheduler ensures that tasks in the Ready or Running state will always be given processor (CPU) time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed into the Running state is always the highest priority task that is able to run.

Any number of tasks can share the same priority. If configUSE*TIME*SLICING is not defined, or if configUSE*TIME*SLICING is set to 1, then Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme.

1.2.3.3 Implementing a Task//任务函数主体格式

A task should have the following structure:

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }

    /* Tasks must not attempt to return from their implementing
    function or otherwise exit.  In newer FreeRTOS port
    attempting to do so will result in an configASSERT() being
    called if it is defined.  If it is necessary for a task to
    exit then have the task call vTaskDelete( NULL ) to ensure
    its exit is clean. */
    vTaskDelete( NULL );
}
```

The type TaskFunction_t is defined as a function that returns void and takes a void pointer as its only parameter. All functions that implement a task should be of this type. The parameter can be used to pass information of any type into the task - this is demonstrated by several of the standard demo application tasks.

Task functions should never return so are typically implemented as a continuous loop. Again, see the RTOS demo application for numerous examples.

Tasks are created by calling **xTaskCreate()** and deleted by calling **vTaskDelete()**.

1.2.3.4 Idle Task and Idle Hook//空闲任务和空闲钩子函数

The Idle Task//空闲任务

The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run. It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state.

The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the vTaskDelete() function to ensure the idle task is not starved of processing time. The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions.

It is possible for application tasks to share the idle task priority (tskIDLE*PRIORITY). See the configIDLE*SHOULD_YIELD configuration parameter for information on how this behaviour can be configured.

The Idle Task Hook//空闲钩子函数

An idle task hook is a function that is called during each cycle of the idle task. If you want application functionality to run at the idle priority then there are two options:

1. Implement the functionality in an idle task hook.

   There must always be at least one task that is ready to run. It is therefore imperative that the hook function does not call any API functions that might cause the idle task to block (vTaskDelay(), or a queue or semaphore function with a block time, for example). It is ok for co-routines to block within the hook function.

2. Create an idle priority task to implement the functionality.

   This is a more flexible solution but has a higher RAM usage overhead.

See the Embedded software application design section for more information on using an idle hook.

To create an idle hook:

1. Set configUSE*IDLE*HOOK to 1 in FreeRTOSConfig.h.
2. Define a function that has the following name and prototype:

   ```
   void vApplicationIdleHook( void );
   ```

It is common to use the idle hook function to place the microcontroller CPU into a power saving mode.

1.2.4 More About Co-routines...

1.2.5 Queues, Mutexes, Semaphores...

1.2.6 Direct To Task Notifications

1.2.7 Software Timers

1.2.8 Event Groups (or 'flags')

1.2.9 Source Code Organisation

1.3 More Advanced...//高级特性

1.3.1 Creating a New Project

1.3.2 FreeRTOSConfig.h//配置文件

FreeRTOS is customised using a configuration file called FreeRTOSConfig.h. Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path.

FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.

Each demo application included in the RTOS source code download has its own FreeRTOSConfig.h file. Some of the demos are quite old and do not contain all the available configuration options. Configuration options that are omitted are set to a default value within an RTOS source file.

Here is a typical FreeRTOSConfig.h definition, followed by an explanation of each parameter:

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
```

```
/* Here is a good place to include header files that are required across
your application. */
#include "something.h"

#define configUSE_PREEMPTION                    1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE                 0
#define configCPU_CLOCK_HZ                      60000000
#define configTICK_RATE_HZ                      250
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configTOTAL_HEAP_SIZE                   10240
#define configMAX_TASK_NAME_LEN                 16
#define configUSE_16_BIT_TICKS                  0
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  0
#define configUSE_NEWLIB_REENTRANT              0
#define configENABLE_BACKWARD_COMPATIBILITY     0
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0

/* Run time and task stats gathering related definitions. */
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                0
#define configUSE_STATS_FORMATTING_FUNCTIONS    0

/* Co-routine related definitions. */
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1

/* Software timer related definitions. */
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               3
#define configTIMER_QUEUE_LENGTH                10
#define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE

/* Interrupt nesting behaviour configuration. */
```

```
#define configKERNEL_INTERRUPT_PRIORITY         [dependent of processor]
#define configMAX_SYSCALL_INTERRUPT_PRIORITY    [dependent on processor and application]
#define configMAX_API_CALL_INTERRUPT_PRIORITY   [dependent on processor and application]

/* Define to trap errors during development. */
#define configASSERT( ( x ) )     if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )

/* FreeRTOS MPU specific definitions. */
#define configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS 0

/* Optional functions - most linkers will remove unused functions anyway. */
#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_xResumeFromISR                  1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTaskGetCurrentTaskHandle       1
#define INCLUDE_uxTaskGetStackHighWaterMark     0
#define INCLUDE_xTaskGetIdleTaskHandle          0
#define INCLUDE_xTimerGetTimerDaemonTaskHandle  0
#define INCLUDE_pcTaskGetTaskName               0
#define INCLUDE_eTaskGetState                   0
#define INCLUDE_xEventGroupSetBitFromISR        1
#define INCLUDE_xTimerPendFunctionCall          0

/* A header file that defines trace macro can be included here. */

#endif /* FREERTOS_CONFIG_H */
```

'config' Parameters

1.3.2.1 configUSE_PREEMPTION

Set to 1 to use the preemptive RTOS scheduler, or 0 to use the cooperative RTOS scheduler.

```
设为 1则采用抢占式调度器, 设为 0则采用协作式调度器。
```

1.3.2.2 configUSE*PORTOPTIMISEDTASK*SELECTION

Some FreeRTOS ports have two methods of selecting the next task to execute - a generic method, and a method that is specific to that port.

The Generic method:

- Is used when configUSE*PORT*OPTIMISED*TASK*SELECTION is set to 0, or when a port specific method is not implemented.
- Can be used with all FreeRTOS ports.
- Is completely written in C, making it less efficient than a port specific method.
- Does not impose a limit on the maximum number of available priorities.

A port specific method:

- Is not available for all ports.
- Is used when configUSE*PORT*OPTIMISED*TASK*SELECTION is set to 1.
- Relies on one or more architecture specific assembly instructions (typically a Count Leading Zeros [CLZ] of equivalent instruction) so can only be used with the architecture for which it was specifically written.
- Is more efficient than the generic method.
- Typically imposes a limit of 32 on the maximum number of available priorities.

1.3.2.3 configUSE*TICKLESS*IDLE

Set configUSE*TICKLESS*IDLE to 1 to use the low power tickless mode, or 0 to keep the tick interrupt running at all times.

```
设为1则使能 tick hook，设为0则禁止tick hook。
```

1.3.2.4 configUSE*IDLE*HOOK

Set to 1 if you wish to use an idle hook, or 0 to omit an idle hook.

```
设为1则使能 idle hook，设为0则禁止idle hook。
```

1.3.2.5 configUSE_MALLOC*FAILED*HOOK

The kernel uses a call to pvPortMalloc() to allocate memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes four sample memory allocation schemes for this purpose. The schemes are implemented in the heap*1.c, heap*2.c, heap*3.c, heap4.c and heap5.c source files respectively. configUSE*MALLOC*FAILED*HOOK is only relevant when one of these three sample schemes is being used.

The malloc() failed hook function is a hook (or callback) function that, if defined and configured, will be called if pvPortMalloc() ever returns NULL. NULL will be returned only if there is insufficient FreeRTOS heap memory remaining for the requested allocation to succeed.

If configUSE*MALLOC*FAILED*HOOK is set to 1 then the application must define a malloc() failed hook function. If configUSEMALLOCFAILED*HOOK is set to 0 then the malloc() failed hook function will not be called, even if one is defined. Malloc() failed hook functions must have the name and prototype shown below.

```
void vApplicationMallocFailedHook( void );
```

1.3.2.6 configUSE*TICK*HOOK

Set to 1 if you wish to use an tick hook, or 0 to omit an tick hook.

1.3.2.7 configCPU*CLOCK*HZ

Enter the frequency in Hz at which the internal clock that driver the peripheral used to generate the tick interrupt will be executing - this is normally the same clock that drives the internal CPU clock. This value is required in order to correctly configure timer peripherals.

```
设置为 MCU 内核的工作频率，以Hz为单位。配置FreeRTOS的时钟Tick时会用到。
```

```
对不同的移植代码也可能不使用这个参数。如果确定移植代码中不用它就可以注释掉这行。
```

1.3.2.8 configTICK*RATE*HZ

The frequency of the RTOS tick interrupt.

The tick interrupt is used to measure time. Therefore a higher tick frequency means time can be measured to a higher resolution. However, a high tick frequency also means that the RTOS kernel will use more CPU time so be less efficient. The RTOS demo applications all use a tick rate of 1000Hz. This is used to test the RTOS kernel and is higher than would normally be required.

More than one task can share the same priority. The RTOS scheduler will share processor time between tasks of the same priority by switching between the tasks during each RTOS tick. A high tick rate frequency will therefore also have the effect of reducing the 'time slice' given to each task.

```
FreeRTOS的时钟Tick的频率，也就是FreeRTOS用到的定时中断的产生频率。
这个频率越高则定时的精度越高，但是由此带来的开销也越大。
FreeRTOS 自带的Demo 程序中将TickRate 设为了1000Hz只是用来测试内核的性能的。
实际的应用程序应该根据需要改为较小的数值。

当多个任务共用一个优先级时，内核调度器回来每次时钟中断到来后轮转切换任务（round robin），
因此，更高的Tick Rate 会导致任务的时间片“time slice”变短。
```

1.3.2.9 configMAX_PRIORITIES

The number of priorities available to the application tasks. Any number of tasks can share the same priority. Co-routines are prioritised separately - see configMAX*CO*ROUTINE_PRIORITIES.

Each available priority consumes RAM within the RTOS kernel so this value should not be set any higher than actually required by your application.

```
程序中可以使用的最大优先级。FreeRTOS 会为每个优先级建立一个链表，
因此没多一个优先级都会增加些RAM 的开销。所以，要根据程序中需要多少种不同的优先级来设置这个参数。
```

1.3.2.10 configMINIMAL*STACK*SIZE

The size of the stack used by the idle task. Generally this should not be reduced from the value set in the FreeRTOSConfig.h file provided with the demo application for the port you are using.

Like the stack size parameter to the xTaskCreate() function, the stack size is specified in words, not bytes. If each item placed on the stack is 32-bits, then a stack size of 100 means 400 bytes (each 32-bit stack item consuming 4 bytes).

```
任务堆栈的最小大小，FreeRTOS根据这个参数来给idle task 分配堆栈空间。
这个值如果设置的比实际需要的空间小，会导致程序挂掉。因此，最好不要减小Demo 程序中给出的大小。
```

1.3.2.11 configTOTAL*HEAP*SIZE

The total amount of RAM available to the RTOS kernel.

This value will only be used if your application makes use of one of the sample memory allocation schemes provided in the FreeRTOS source code download. See the memory configuration section for further details.

```
设置堆空间（Heap）的大小。只有当程序中采用FreeRTOS 提供的内存分配算法时才会用到。
```

1.3.2.12 configMAX_TASK*NAME*LEN

The maximum permissible length of the descriptive name given to a task when the task is created. The length is specified in the number of characters including the NULL termination byte.

```
任务名称最大的长度，这个长度是以字节为单位的，并且包括最后的 NULL 结束字节。
```

1.3.2.13 configUSE*TRACE*FACILITY

Set to 1 if you wish to include additional structure members and functions to assist with execution visualisation and tracing.

```
如果程序中需要用到TRACE功能，则需将这个宏设为1。否则设为0。
开启TRACE功能后，RAM占用量会增大许多，因此在设为1之前请三思。
```

1.3.2.14 configUSE_STATS*FORMATTING*FUNCTIONS

Set configUSE*TRACE*FACILITY and configUSE_STATS*FORMATTING*FUNCTIONS to 1 to include the vTaskList() and vTaskGetRunTimeStats() functions in the build. Setting either to 0 will omit vTaskList() and vTaskGetRunTimeStates() from the build.

1.3.2.15 configUSE_16*BIT*TICKS

Time is measured in 'ticks' - which is the number of times the tick interrupt has executed since the RTOS kernel was started. The tick count is held in a variable of type TickType_t.

Defining configUSE*16*BIT*TICKS as 1 causes TickTypet to be defined (typedef'ed) as an unsigned 16bit type. Defining configUSE16*BIT*TICKS as 0 causes TickType*t to be defined (typedef'ed) as an unsigned 32bit type.

Using a 16 bit type will greatly improve performance on 8 and 16 bit architectures, but limits the maximum specifiable time period to 65535 'ticks'. Therefore, assuming a tick frequency of 250Hz, the maximum time a task can delay or block when a 16bit counter is used is 262 seconds, compared to 17179869 seconds when using a 32bit counter.

```
将 configUSE_16_BIT_TICKS设为 1后portTickType 将被定义为无符号的16位整形类型，
configUSE_16_BIT_TICKS 设为0 后portTickType 则被定义为无符号的32位整型。
```

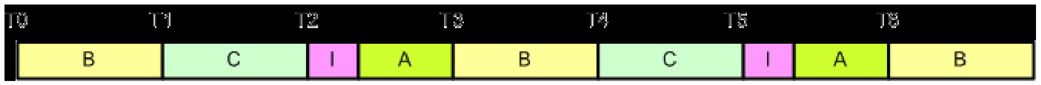
1.3.2.16 configIDLE*SHOULD*YIELD

This parameter controls the behaviour of tasks at the idle priority. It only has an effect if:

1. The preemptive scheduler is being used.
2. The users application creates tasks that run at the idle priority.

Tasks that share the same priority will time slice. Assuming none of the tasks get preempted, it might be assumed that each task of at a given priority will be allocated an equal amount of processing time - and if the shared priority is above the idle priority then this is indeed the case.

When tasks share the idle priority the behaviour can be slightly different. When configIDLE*SHOULD*YIELD is set to 1 the idle task will yield immediately should any other task at the idle priority be ready to run. This ensures the minimum amount of time is spent in the idle task when application tasks are available for scheduling. This behaviour can however have undesirable effects (depending on the needs of your application) as depicted below:



This diagram shows the execution pattern of four tasks at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. A context switch occurs with regular period at times T0, T1, ..., T6. When the idle task yields task A starts to execute - but the idle task has already taken up some of the current time slice. This results in task I and task A effectively sharing a time slice. The application tasks B and C therefore get more processing time than the application task A.

This situation can be avoided by:

- If appropriate, using an idle hook in place of separate tasks at the idle priority.
- Creating all application tasks at a priority greater than the idle priority.
- Setting configIDLE*SHOULD*YIELD to 0.

Setting configIDLE*SHOULD*YIELD prevents the idle task from yielding processing time until the end of its time slice. This ensure all tasks at the idle priority are allocated an equal amount of processing time - but at the cost of a greater proportion of the total processing time being allocated to the idle task.

```
这个参数控制那些优先级与idle 任务相同的任务的行为,
并且只有当内核被配置为抢占式任务调度时才有实际作用。
内核对具有同样优先级的任务会采用时间片轮转调度算法。
当任务的优先级高于idle任务时,各个任务分到的时间片是同样大小的。
但当任务的优先级与idle任务相同时情况就有些不同了。
当configIDLE_SHOULD_YIELD 被配置为1时,当任何优先级与idle 任务相同的任务处于就绪态时,
idle任务会立刻要求调度器进行任务切换。这会使idle任务占用最少的CPU时间,
但同时会使得优先级与idle 任务相同的任务获得的时间片不是同样大小的。
因为idle任务会占用某个任务的部分时间片。
```

1.3.2.17 configUSE*TASK*NOTIFICATIONS

Setting configUSE*TASK*NOTIFICATIONS to 1 (or leaving configUSE*TASK*NOTIFICATIONS undefined) will include direct to task notification functionality and its associated API in the build.

Setting configUSE*TASK*NOTIFICATIONS to 0 will exclude direct to task notification functionality and its associated API from the build.

Each task consumes 8 additional bytes of RAM when direct to task notifications are included in the build.

1.3.2.18 configUSE_MUTEXES

Set to 1 to include mutex functionality in the build, or 0 to omit mutex functionality from the build. Readers should familiarise themselves with the differences between mutexes and binary semaphores in relation to the FreeRTOS functionality.

```
设为 1 则程序中会包含mutex 相关的代码,设为 0 则忽略相关的代码。
```

1.3.2.19 configUSE*RECURSIVE*MUTEXES

Set to 1 to include recursive mutex functionality in the build, or 0 to omit recursive mutex functionality from the build.

```
设为 1 则程序中会包含recursive mutex 相关的代码，设为 0 则忽略相关的代码。
```

1.3.2.20 configUSE*COUNTING*SEMAPHORES

Set to 1 to include counting semaphore functionality in the build, or 0 to omit counting semaphore functionality from the build.

```
设为 1 则程序中会包含semaphore 相关的代码，设为 0 则忽略相关的代码。
```

1.3.2.21 configUSE*ALTERNATIVE*API

Set to 1 to include the 'alternative' queue functions in the build, or 0 to omit the 'alternative' queue functions from the build. The alternative API is described within the queue.h header file. **The alternative API is deprecated and should not be used in new designs.**

```
设为 1 则程序中会包含一些关于队列操作的额外API函数，设为 0 则忽略相关的代码。
这些额外提供的API运行速度更快，但是临界区（关中断）的长度也更长。有利也有弊，
是否要采用需要用户自己考虑了。
```

1.3.2.22 configCHECK_FOR*STACK*OVERFLOW

The stack overflow detection page describes the use of this parameter.

```
控制是否检测堆栈溢出。
```

1.3.2.23 configQUEUE*REGISTRY*SIZE

The queue registry has two purposes, both of which are associated with RTOS kernel aware debugging:

1. It allows a textual name to be associated with a queue for easy queue identification within a debugging GUI.
2. It contains the information required by a debugger to locate each registered queue and semaphore.

The queue registry has no purpose unless you are using a RTOS kernel aware debugger.

configQUEUE*REGISTRY*SIZE defines the maximum number of queues and semaphores that can be registered. Only those queues and semaphores that you want to view using a RTOS kernel aware debugger need be registered. See the API reference documentation for vQueueAddToRegistry() and vQueueUnregisterQueue() for more information.

```
队列注册表有两个作用，但是这两个作用都依赖于调试器的支持：
1.给队列一个名字，方便调试时辨认是哪个队列。
2.包含调试器需要的特定信息用来定位队列和信号量。
如果你的调试器没有上述功能，哪个这个注册表就毫无用处，还占用的宝贵的RAM空间。
```

1.3.2.24 configUSE*QUEUE*SETS

Set to 1 to include queue set functionality (the ability to block, or pend, on multiple queues and semaphores), or 0 to omit queue set functionality.

1.3.2.25 configUSE*TIME*SLICING

By default (if configUSE*TIME*SLICING is not defined, or if configUSE*TIMESLICING is defined as 1) FreeRTOS uses prioritised preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSETIME*SLICING is set to 0 then the RTOS scheduler will still run the highest priority task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt has occurred.

1.3.2.26 configUSE*NEWLIB*REENTRANT

If configUSE*NEWLIB*REENTRANT is set to 1 then a newlib reent structure will be allocated for each created task.

Note Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for resulting newlib operation. User must be familiar with newlib and must provide system-wide implementations of the necessary stubs. Be warned that (at the time of writing) the current newlib design implements a system-wide malloc() that must be provided with locks.

1.3.2.27 configENABLE*BACKWARD*COMPATIBILITY

The FreeRTOS.h header file includes a set of #define macros that map the names of data types used in versions of FreeRTOS prior to version 8.0.0 to the names used in FreeRTOS version 8.0.0. The macros allow application code to update the version of FreeRTOS they are built against from a pre 8.0.0 version to a post 8.0.0 version without modification. Setting configENABLE*BACKWARD*COMPATIBILITY to 0 in FreeRTOSConfig.h excludes the macors from the build, and in so doing allowing validation that no pre version 8.0.0 names are being used.

1.3.2.28 configNUM*THREADLOCALSTORAGE*POINTERS

Sets the number of indexes in each task's thread local storage array.

1.3.2.29 configGENERATE_RUN*TIME*STATS

The Run Time Stats page describes the use of this parameter.

```
设置是否产生运行时的统计信息，这些信息只对调试有用，会保存在RAM 中，占用RAM空间。
因此，最终程序建议配置成不产生运行时统计信息。
```

1.3.2.30 configUSE*CO*ROUTINES

Set to 1 to include co-routine functionality in the build, or 0 to omit co-routine functionality from the build. To include co-routines croutine.c must be included in the project.

```
设置为1则包含co-routines 功能，如果包含了co-routines功能，则编译时需包含croutine.c 文件
```

1.3.2.31 configMAX_CO*ROUTINE*PRIORITIES

The number of priorities available to the application co-routines. Any number of co-routines can share the same priority. Tasks are prioritised separately - see configMAX_PRIORITIES.

```
co-routines 可以使用的优先级的数量。
```

1.3.2.32 configUSE_TIMERS

Set to 1 to include software timer functionality, or 0 to omit software timer functionality. See the FreeRTOS software timers page for a full description.

```
设置为1则包含软件定时器功能。
```

1.3.2.33 configTIMER*TASK*PRIORITY

Sets the priority of the software timer service/daemon task. See the FreeRTOS software timers page for a full description.

```
设置软件定时器任务的优先级。
```

1.3.2.34 configTIMER*QUEUE*LENGTH

Sets the length of the software timer command queue. See the FreeRTOS software timers page for a full description.

```
设置软件定时器任务中用到的命令队列的长度。
```

1.3.2.35 configTIMER_TASK*STACK*DEPTH

Sets the stack depth allocated to the software timer service/daemon task. See the FreeRTOS software timers page for a full description.

```
设置软件定时器任务需要的任务堆栈大小。
```

1.3.2.36 configKERNEL*INTERRUPT*PRIORITY

1.3.2.37 configMAX_SYSCALL*INTERRUPT*PRIORITY and

1.3.2.38 configMAX*APICALLINTERRUPT*PRIORITY

Ports that contain a configKERNEL*INTERRUPT*PRIORITY setting include ARM Cortex-M3, PIC24, dsPIC, PIC32, SuperH and RX600. Ports that contain a configMAX_SYSCALL*INTERRUPT*PRIORITY setting include PIC32, RX600, ARM Cortex-A and ARM Cortex-M ports.

ARM Cortex-M3 and ARM Cortex-M4 users please take heed of the special note at the end of this section!

configMAX*API*CALL*INTERRUPTPRIORITY is a new name for configMAXSYSCALL*INTERRUPT_PRIORITY that is used by newer ports only. The two are equivalent.

configKERNEL*INTERRUPT*PRIORITY should be set to the lowest priority.

Note in the following discussion that only API functions that end in "FromISR" can be called from within an interrupt service routine.

For ports that only implement configKERNELINTERRUPTPRIORITY

configKERNEL*INTERRUPT*PRIORITY sets the interrupt priority used by the RTOS kernel itself. Interrupts that call API functions must also execute at this priority. Interrupts that do not call API functions can execute at higher priorities and therefore never have their execution delayed by the RTOS kernel activity (within the limits of the hardware itself).

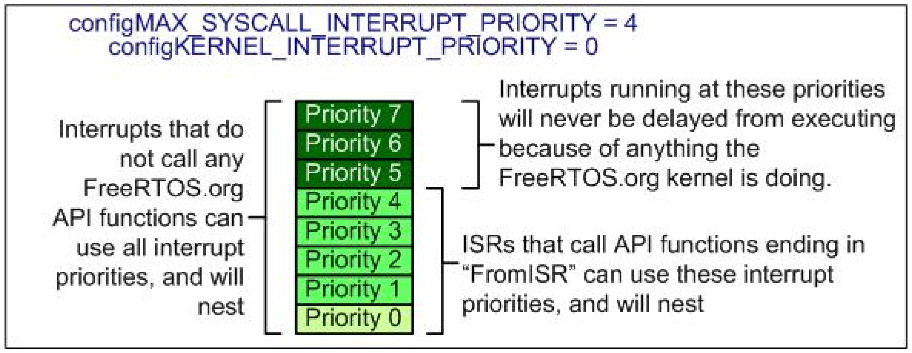
For ports that implement both configKERNELINTERRUPTPRIORITY and configMAXSYSCALLINTERRUPT_PRIORITY:

configKERNEL*INTERRUPT*PRIORITY sets the interrupt priority used by the RTOS kernel itself. configMAX_SYSCALL*INTERRUPT*PRIORITY sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.

A full interrupt nesting model is achieved by setting configMAX*SYSCALL*INTERRUPT_PRIORITY above (that is, at a higher priority level) than configKERNEL*INTERRUPT*PRIORITY. **This means the FreeRTOS kernel does not completely disable interrupts, even inside critical sections.**Further, this is achieved without the disadvantages of a segmented kernel architecture. Note however, certain microcontroller architectures will (in hardware) disable interrupts when a new interrupt is accepted - meaning interrupts are unavoidably disabled for the short period between the hardware accepting the interrupt, and the FreeRTOS code re-enabling interrupts.

Interrupts that do not call API functions can execute at priorities above configMAX*SYSCALL*INTERRUPT_PRIORITY and therefore never be delayed by the RTOS kernel execution.

For example, imagine a hypothetical microcontroller that has 8 interrupt priority levels - 0 being the lowest and 7 being the highest (see the special note for ARM Cortex-M3 users at the end of this section). The picture below describes what can and cannot be done at each priority level should the two configuration constants be set to 4 and 0 as shown:


configMAX_SYSCALL_INTERRUPT_PRIORITY = 4
configKERNEL_INTERRUPT_PRIORITY = 0
Interrupts that do not call any FreeRTOS.org API functions can use all interrupt priorities, and will nest
Priority 7
Priority 6
Priority 5
Priority 4
Priority 3
Priority 2
Priority 1
Priority 0
Interrupts running at these priorities will never be delayed from executing because of anything the FreeRTOS.org kernel is doing.
ISRs that call API functions ending in "FromISR" can use these interrupt priorities, and will nest
Example interrupt priority configuration

These configuration parameters allow very flexible interrupt handling:

- Interrupt handling 'tasks' can be written and prioritised as per any other task in the system. These are tasks that are woken by an interrupt. The interrupt service routine (ISR) itself should be written to be as short as it possibly can be - it just grabs the data then wakes the high priority handler task. The ISR then returns directly into the woken handler task - so interrupt processing is contiguous in time just as if it were all done in the ISR itself. The benefit of this is that all interrupts remain enabled while the handler task executes.
- Ports that implement configMAX*SYSCALL*INTERRUPT*PRIORITY take this further - permitting a fully nested model where interrupts between the RTOS kernel interrupt priority and configMAXSYSCALLINTERRUPT*PRIORITY can nest and make applicable API calls. Interrupts with priority above configMAX*SYSCALL*INTERRUPT_PRIORITY are never delayed by the RTOS kernel activity.
- ISR's running above the maximum syscall priority are never masked out by the RTOS kernel itself, so their responsiveness is not effected by the RTOS kernel functionality. This is ideal for interrupts that require very high temporal accuracy - for example interrupts that perform motor commutation. However, such ISR's cannot use the FreeRTOS API functions.

To utilize this scheme your application design must adhere to the following rule: Any interrupt that uses the FreeRTOS API must be set to the same priority as the RTOS kernel (as configured by the configKERNEL*INTERRUPT*PRIORITY macro), or at or below configMAX_SYSCALL*INTERRUPT*PRIORITY for ports that include this functionality.

A special note for ARM Cortex-M3 and ARM Cortex-M4 users: Please read the page dedicated to interrupt priority settings on ARM Cortex-M devices. As a minimum, remember that ARM Cortex-M3 cores use numerically low priority numbers to represent HIGH priority interrupts, which can seem counter-intuitive and is easy to forget! If you wish to assign an interrupt a low priority do NOT assign it a priority of 0 (or other low numeric value) as this can result in the interrupt actually having the highest priority in the system - and therefore potentially make your system crash if this priority is above configMAX*SYSCALL*INTERRUPT_PRIORITY.

The lowest priority on a ARM Cortex-M3 core is in fact 255 - however different ARM Cortex-M3 vendors implement a different number of priority bits and supply library functions that expect priorities to be specified in different ways. For example, on the STM32 the lowest priority you can specify in an ST driver library call is in fact 15 - and the highest priority you can specify is 0.

```
Cortex-M3, PIC24, dsPIC, PIC32, SuperH 和 RX600 的移植代码中会使用到
configKERNEL_INTERRUPT_PRIORITY.
PIC32, RX600 和 Cortex-M系列 会使用到 configMAX_SYSCALL_INTERRUPT_PRIORITY
configKERNEL_INTERRUPT_PRIORITY应该被设为最低优先级。

对那些只定义了 configKERNEL_INTERRUPT_PRIORITY 的系统:
configKERNEL_INTERRUPT_PRIORITY决定了FreeRTOS内核使用的优先级。
所有调用API函数的中断的优先级都应设为这个值，不调用API函数的中断可以设为更高的优先级。

对那些定义了configKERNEL_INTERRUPT_PRIORITY 和configMAX_SYSCALL_INTERRUPT_PRIORITY的系统:
configKERNEL_INTERRUPT_PRIORITY决定了FreeRTOS内核使用的优先级。
configMAX_SYSCALL_INTERRUPT_PRIORITY决定了可以调用API函数的中断的最高优先级。
高于这个值的中断处理函数不能调用任何API 函数。
```

1.3.2.39 configASSERT

The semantics of the configASSERT() macro are the same as the standard C assert() macro. An assertion is triggered if the parameter passed into configASSERT() is zero.

configASSERT() is called throughout the FreeRTOS source files to check how the application is using FreeRTOS. It is highly recommended to develop FreeRTOS applications with configASSERT() defined.

The example definition (shown at the top of the file and replicated below) calls vAssertCalled(), passing in the file name and line number of the triggering configASSERT() call (**FILE** and **LINE** are standard macros provided by most compilers). This is just for demonstration as vAssertCalled() is not a FreeRTOS function, configASSERT() can be defined to take whatever action the application writer deems appropriate.

It is normal to define configASSERT() in such a way that it will prevent the application from executing any further. This if for two reasons; stopping the application at the point of the assertion allows the cause of the assertion to be debugged, and executing past a triggered assertion will probably result in a crash anyway.

Note defining configASSERT() will increase both the application code size and execution time. When the application is stable the additional overhead can be removed by simply commenting out the configASSERT() definition in FreeRTOSConfig.h.

```
/* Define configASSERT() to call vAssertCalled() if the assertion fails.  The assertion
has failed if the value of the parameter passed into configASSERT() equals zero. */
#define configASSERT( ( x ) )
if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )
```

If running FreeRTOS under the control of a debugger, then configASSERT() can be defined to just disable interrupts and sit in a loop, as demonstrated below. That will have the effect of stopping the code on the line that failed the assert test - pausing the debugger will then immediately take you to the offending line so you can see why it failed.

```
/* Define configASSERT() to disable interrupts and sit in a loop. */
#define configASSERT( ( x ) )
if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }


宏configASSERT()的作用类似C语言标准库中的宏assert()，
configASSERT() 可以帮助调试，但是定义了configASSERT()后会增加程序代码，也会使程序变慢。
```

1.3.2.40 configINCLUDE*APPLICATIONDEFINEDPRIVILEGED*FUNCTIONS

configINCLUDE*APPLICATION*DEFINED*PRIVILEGED*FUNCTIONS is only used by FreeRTOS MPU.

If configINCLUDE*APPLICATION*DEFINED*PRIVILEGEDFUNCTIONS is set to 1 then the application writer must provide a header file called "applicationdefined*privileged_functions.h", in which functions the application writer needs to execute in privileged mode can be implemented. Note that, despite having a .h extension, the header file should contain the implementation of the C functions, not just the functions' prototypes.

Functions implemented in "application*defined*privileged*functions.h" must save and restore the processor's privilege state using the prvRaisePrivilege() function and portRESET*PRIVILEGE() macro respectively. For example, if a library provided print function accesses RAM that is outside of the control of the application writer, and therefore cannot be allocated to a memory protected user mode task, then the print function can be encapsulated in a privileged function using the following code:

```
void MPU_debug_printf( const char *pcMessage )
{
    /* State the privilege level of the processor when the function was called. */
    BaseType_t xRunningPrivileged = prvRaisePrivilege();

    /* Call the library function, which now has access to all RAM. */
    debug_printf( pcMessage );

    /* Reset the processor privilege level to its original value. */
    portRESET_PRIVILEGE( xRunningPrivileged );
}
```

This technique should only be use during development, and not deployment, as it circumvents the memory protection.

```
以 'INCLUDE' 开头的宏允许我们将部分不需要的API 函数排除在编译生成的代码之外。
这可以使内核代码占用更少的ROM 和RAM。
比如，如果代码中需要用到 vTaskDelete 函数则这样写：
#defineINCLUDE_vTaskDelete    1
如果不需要，则这样写：
#defineINCLUDE_vTaskDelete    0
```

1.3.2.41 INCLUDE Parameters

The macros starting 'INCLUDE' allow those components of the real time kernel not utilized by your application to be excluded from your build. This ensures the RTOS does not use any more ROM or RAM than necessary for your particular embedded application.

Each macro takes the form ...

INCLUDE_FunctionName

... where FunctionName indicates the API function (or set of functions) that can optionally be excluded. To include the API function set the macro to 1, to exclude the function set the macro to 0. For example, to include the vTaskDelete() API function use:

```
#define INCLUDE_vTaskDelete    1
```

To exclude vTaskDelete() from your build use:

```
#define INCLUDE_vTaskDelete    0
```

1.4 Demo Projects

1.5 Supported Devices & Demos

1.6 API Reference//API参考手册

1.6.1 Task Creation//任务创建

1.6.1.1 TaskHandle_t (type)//任务句柄类型

task. h

Type by which tasks are referenced. For example, a call to xTaskCreate returns (via a pointer parameter) an TaskHandle_t variable that can then be used as a parameter to vTaskDelete to delete the task.

1.6.1.2 xTaskCreate()//创建一个任务

task. h

```
BaseType_t xTaskCreate(
        TaskFunction_t pvTaskCode,
        const char * const pcName,
        unsigned short usStackDepth,
        void *pvParameters,
        UBaseType_t uxPriority,
        TaskHandle_t *pvCreatedTask
);
```

Create a new task and add it to the list of tasks that are ready to run.

If you are using **FreeRTOS-MPU** then it is recommended to use **xTaskCreateRestricted()** in place of xTaskCreate(). Using xTaskCreate() with

FreeRTOS-MPU allows tasks to be created to run in either Privileged or User modes (see the description of uxPriority below). When Privileged mode it used the task will have access to the entire memory map, when User mode is used the task will have access to only its stack. In both cases the MPU will not automatically catch stack overflows, although the standard FreeRTOS stack overflow detection schemes can still be used. xTaskCreateRestricted() permits much greater flexibility.

Parameters:

> **pvTaskCode:** Pointer to the task entry function. Tasks must be implemented to **never return** (i.e. continuous loop).
>
> **pcName:** A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by **configMAX*TASK*NAME_LEN**.
>
> **usStackDepth:** The size of the task stack specified as the number of variables the stack can hold - **not the number of bytes**. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.
>
> **pvParameters:** Pointer that will be used as the parameter for the task being created.
>
> **uxPriority:** The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE*BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE*BIT).
>
> **pvCreatedTask:** Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs. h

Example usage:

```
/* Task to be created. */
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        /* Task code goes here. */
    }
}


/* Function that creates a task. */
void vOtherFunction( void )
{
    static unsigned char ucParameterToPass;
    TaskHandle_t xHandle = NULL;

    /* Create the task, storing the handle.  Note that the passed parameter
    ucParameterToPass must exist for the lifetime of the task, so in this
    case is declared static.  If it was just an an automatic stack variable
    it might no longer exist, or at least have been corrupted, by the time
    the new task attempts to access it. */
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
    &xHandle );
    configASSERT( xHandle );

    /* Use the handle to delete the task. */
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
 }
```

1.6.1.3 vTaskDelete()//删除一个任务

task. h

```
void vTaskDelete( TaskHandle_t xTask );
```

INCLUDE_vTaskDelete must be defined as 1 for this function to be available. See the configuration section for more information.

Remove a task from the RTOS kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

NOTE: The idle task is responsible for freeing the RTOS kernel allocated memory from tasks that have been deleted. It is therefore important that the idle task is not starved of microcontroller processing time if your application makes any calls to vTaskDelete (). Memory allocated by the task code is not automatically freed, and should be freed before the task is deleted.

See the demo application file death. c for sample code that utilises vTaskDelete ().

Parameters:

> **xTask:** The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

```
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;
    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
    // Use the handle to delete the task.
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

1.6.2 Task Control

1.6.2.1 vTaskDelay()

1.6.2.2 vTaskDelayUntil()

1.6.2.3 uxTaskPriorityGet()

1.6.2.4 vTaskPrioritySet()

1.6.2.5 vTaskSuspend()

1.6.2.6 vTaskResume()

1.6.2.7 xTaskResumeFromISR()

1.6.3 Task Utilities

1.6.3.1 uxTaskGetSystemState()

1.6.3.2 xTaskGetApplicationTaskTag()

1.6.3.3 xTaskGetCurrentTaskHandle()

1.6.3.4 xTaskGetIdleTaskHandle()

1.6.3.5 uxTaskGetStackHighWaterMark()

1.6.4 RTOS Kernel Control//内核控制函数

1.6.4.1 taskYIELD()

task. h

Macro for forcing a context switch.//一个宏，主动进行上下文切换

来源于网络解释：比如我创建了8个优先级一样的task,并且没有创建其他优先级的进程, 而且8个task每个task都不会调用任何引起本task从就绪运行队列链表中被摘掉的系统函数,就像示例中 vStartIntegerMathTasks()创建vCompeteingIntMathTask1(),vCompeteingIntMathTask2()...vCompeteingIntMathTask8()一样, 每个task都是不会睡眠的不停的执行自己,当每个task觉得自己占用cpu的时间已经差不多的时候, 就会**调用taskYIELD(),主动让出cpu,让同优先级的其他task获得cpu**,因为没有其他优先级的task,所以调度器不会切换优先级, 而是采用轮转调度策略,运行同优先级的就绪运行队列链表中调用taskYIELD()函数的当前task的下一个task. 就这样8个task轮流让出cpu给同优先级的下一个兄弟task,8个task都采用主动协作的方式,彼此安全顺利的跑了起来.

下面是跟踪ARM_CM3的切换形式。

```
#define taskYIELD()                  portYIELD()   //task. h
#define portYIELD()                  vPortYield()  //portmacro.h
void vPortYield( void )//port.c
{
    /* Set a PendSV to request a context switch. */
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
    /* Barriers are normally not required but do ensure the code is completely
    within the specified behaviour for the architecture. */
    __asm volatile( "dsb" );
    __asm volatile( "isb" );
}
```

1.6.4.2 taskENTER_CRITICAL()

task. h

Macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

```
功能：进入代码的临界段的宏。在临界段，抢先的现场切换无法发生。
注意，这可能改变堆栈（依赖于移植是如何实现的），故必须小心使用。
其实现原理跟所移植的芯片有关，具体参见port.c里的函数void vPortEnterCritical( void )的实现
此函数和taskEXIT_CRITICAL必须成对出现使用。
```

1.6.4.3 taskEXIT_CRITICAL()

task. h

Macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

NOTE: This may alter the stack (depending on the portable implementation) so must be used with care!

```
功能：退出代码的临界段的宏。在临界段，抢先的现场切换无法发生。
注意，这可能改变堆栈（依赖于移植是如何实现的），故必须小心使用。
此函数和taskENTER_CRITICAL必须成对出现使用。
```

1.6.4.4 taskDISABLE_INTERRUPTS

task. h

Macro to disable all maskable interrupts.

```
功能：关处理器的总中断
```

1.6.4.5 taskENABLE_INTERRUPTS

task. h

Macro to enable microcontroller interrupts.

```
功能：开处理器的总中断
```

1.6.4.6 vTaskStartScheduler()

task. h

```
void vTaskStartScheduler( void );
```

Starts the RTOS scheduler. After calling the RTOS kernel has control over which tasks are executed and when.

The idle task and optionally the timer daemon task are created automatically when the RTOS scheduler is started.

vTaskStartScheduler() will only return if there is insufficient RTOS heap available to create the idle or timer daemon tasks.

All the RTOS demo application projects contain examples of using vTaskStartScheduler(), normally in the main() function within main.c.

相关解释翻译：

```
函数功能：启动实时内核时间封处理。调用后，内核控制何时何地哪个任务运行。
当此函数被调用时，空闲任务自动创建。
此函数成功时，永远不会返回，直到vTaskEndScheduler()被调用。
如果没有足够的RAM创建空闲函数，则该函数可能失败并立即返回。
```

Example usage:

```
void vAFunction( void )
{
     // Tasks can be created before or after starting the RTOS scheduler
     xTaskCreate( vTaskCode,
                  "NAME",
                  STACK_SIZE,
                  NULL,
                  tskIDLE_PRIORITY,
                  NULL );
     // Start the real time scheduler.
     vTaskStartScheduler();
     // Will not get here unless there is insufficient RAM.
 }
```

1.6.4.7 vTaskEndScheduler()

task. h

```
void vTaskEndScheduler( void );
```

NOTE: This has only been implemented for the x86 Real Mode PC port.

Stops the RTOS kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler() was called, as if vTaskStartScheduler() had just returned.

See the demo application file main. c in the demo/PC directory for an example that uses vTaskEndScheduler ().

vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the RTOS kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the RTOS kernel to be freed - but will not free resources allocated by application tasks.

```
   函数功能：停止实时内核的时间封。所有已经创建的任务会被自动删除，
多任务处理（无论是可占先还是非可占先任务）停止。
   执行从调用vTaskStartScheduler处恢复执行，当vTaskStartScheduler返回时。
（笔者注释，翻译得比较拗口，后续修正。）
   见demo/PC 文件夹的DEMO文件main. c，演示了vTaskEndScheduler ()的使用。
vTaskEndScheduler需要在移植层定义一个退出函数，这表现为硬件指定操作，
比如停止时间封（停止定时器？待后续追加注释）。
  vTaskEndScheduler导致所有由内核分配的资源被释放，但由任务分配的资源不会被释放。
```

Example usage:

```
void vTaskCode( void * pvParameters )
{
     for( ;; )
     {
         // Task code goes here.

         // At some point we want to end the real time kernel processing
         // so call ...
         vTaskEndScheduler ();
     }
}

void vAFunction( void )
{
     // Create at least one task before starting the RTOS kernel.
     xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL );

     // Start the real time kernel with preemption.
     vTaskStartScheduler();

     // Will only get here when the vTaskCode () task has called
     // vTaskEndScheduler ().  When we get here we are back to single task
     // execution.
}
```

1.6.4.8 vTaskSuspendAll()

task. h

```
void vTaskSuspendAll( void );
```

Suspends the scheduler without disabling interrupts. Context switches will not occur while the scheduler is suspended. RTOS ticks that occur while the scheduler is suspended will be held pending until the scheduler has been unsuspended using a call to xTaskResumeAll().

API functions that have the potential to cause a context switch (for example, vTaskDelayUntil(), xQueueSend(), etc.) must not be called while the scheduler is suspended.

```
函数功能：在保持中断使能（包括内核时间封中断）的情况下，挂起所有实时内核的活动。
当调用 vTaskSuspendAll()后，调用该函数的任务依然会执行而不会被置换出运行态，
直到xTaskResumeAll () 被调用（即可以重新开始任务调度，则当前的任务当然有可能被置换出运行态）。
有潜在可能导致现场切换的API函数不得被调用，当调度器被挂起时。
```

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
         // Task code goes here.

         // ...

         // At some point the task wants to perform a long operation during
         // which it does not want to get swapped out.  It cannot use
         // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
         // operation may cause interrupts to be missed - including the
         // ticks.

         // Prevent the RTOS kernel swapping out the task.
         vTaskSuspendAll ();

         // Perform the operation here.  There is no need to use critical
         // sections as we have all the microcontroller processing time.
         // During this time interrupts will still operate and the RTOS kernel
         // tick count will be maintained.

         // ...

         // The operation is complete.  Restart the RTOS kernel.
         xTaskResumeAll ();
    }
}
```

1.6.4.9 xTaskResumeAll()

task. h

```
BaseType_t xTaskResumeAll( void );
```

Resumes the scheduler after it was suspended using a call to vTaskSuspendAll().

xTaskResumeAll() only resumes the scheduler. It does not unsuspend tasks that were previously suspended by a call to vTaskSuspend().

Returns:

If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

```
函数功能：恢复实时内核的活动。调用后，内核决定何时何地何任务运行。返回值
如果再召唤调度器导致现场切换，返回pdTRUE，否则为pdFALSE。
```

Example usage:

```
void vTask1( void * pvParameters )
{
     for( ;; )
     {
         /* Task code goes here. */

         /* ... */

         /* At some point the task wants to perform a long operation
         during which it does not want to get swapped out.  It cannot
         use taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length
         of the operation may cause interrupts to be missed -
         including the ticks.

         Prevent the RTOS kernel swapping out the task. */
         vTaskSuspendAll();

         /* Perform the operation here.  There is no need to use critical
         sections as we have all the microcontroller processing time.
         During this time interrupts will still operate and the real
         time RTOS kernel tick count will be maintained. */

         /* ... */

         /* The operation is complete.  Restart the RTOS kernel.  We want to force
         a context switch - but there is no point if resuming the scheduler
         caused a context switch already. */
         if( !xTaskResumeAll () )
         {
              taskYIELD ();
         }
     }
}
```

1.6.4.10 vTaskStepTick()

task.h

```
void vTaskStepTick( TickType_t xTicksToJump );
```

If the RTOS is configured to use tickless idle functionality then the tick interrupt will be stopped, and the microcontroller placed into a low power state, whenever the Idle task is the only task able to execute. Upon exiting the low power state the tick count value must be corrected to account for the time that passed while it was stopped.

If a FreeRTOS port includes a default portSUPPRESS*TICKS*AND*SLEEP() implementation, then vTaskStepTick() is used internally to ensure the correct tick count value is maintained. vTaskStepTick() is a public API function to allow the default portSUPPRESSTICKSAND*SLEEP() implementation to be overridden, and for a portSUPPRESS*TICKS*AND_SLEEP() to be provided if the port being used does not provide a default.

The configUSE*TICKLESS*IDLE configuration constant must be set to 1 for vTaskStepTick() to be available.

Parameters:

> **xTicksToJump:** The number of RTOS ticks that have passed since the tick interrupt was stopped. For correct operation the parameter must be less than or equal to the portSUPPRESS_TICKS*AND*SLEEP() parameter.

Returns:

None.

Example usage:

The example shows calls being made to several functions. Only vTaskStepTick() is part of the FreeRTOS API. The other functions are specific to the clocks and power saving modes available on the hardware in use, and as such, must be provided by the application writer.

```
/* First define the portSUPPRESS_TICKS_AND_SLEEP().  The parameter is the time,
in ticks, until the kernel next needs to execute. */
#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )

/* Define the function that is called by portSUPPRESS_TICKS_AND_SLEEP(). */
void vApplicationSleep( TickType_t xExpectedIdleTime )
{
    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

    /* Read the current time from a time source that will remain operational
    while the microcontroller is in a low power state. */
    ulLowPowerTimeBeforeSleep = ulGetExternalTime();

    /* Stop the timer that is generating the tick interrupt. */
    prvStopTickInterruptTimer();

    /* Configure an interrupt to bring the microcontroller out of its low power
    state at the time the kernel next needs to execute.  The interrupt must be
    generated from a source that is remains operational when the microcontroller
    is in a low power state. */
    vSetWakeTimeInterrupt( xExpectedIdleTime );

    /* Enter the low power state. */
    prvSleep();

    /* Determine how long the microcontroller was actually in a low power state
    for, which will be less than xExpectedIdleTime if the microcontroller was
    brought out of low power mode by an interrupt other than that configured by
    the vSetWakeTimeInterrupt() call.  Note that the scheduler is suspended
    before portSUPPRESS_TICKS_AND_SLEEP() is called, and resumed when
    portSUPPRESS_TICKS_AND_SLEEP() returns.  Therefore no other tasks will
    execute until this function completes. */
    ulLowPowerTimeAfterSleep = ulGetExternalTime();

    /* Correct the kernels tick count to account for the time the microcontroller
    spent in its low power state. */
    vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );

    /* Restart the timer that is generating the tick interrupt. */
    prvStartTickInterruptTimer();
}
```

1.6.5 Direct To Task Notifications

1.6.6 FreeRTOS-MPU Specific

1.6.7 Queues

1.6.8 Queue Sets

1.6.9 Semaphore / Mutexes

1.6.10 Software Timers

1.6.11 Event Groups (or 'flags')

1.6.12 Co-routines

第二章：FreeRTOS Interactive!//互动交流FreeRTOS

2.1 Contributed Vs Official Code//开源贡献和官方代码

2.2 Contributing Source Code//开源代码

2.3 Upload / Download Contributions//上传和下载开源代码

第三章：Contact, Support, Advertising//联系，支持和广告

3.1 Contact & Advertising//联系与广告

3.2 Free Support Forum//自由支持形式

3.3 Commercial Support//商业支持与合作

3.4 Contract Services//签约合同服务

3.5 FAQ//常问问题

第四章：The FreeRTOS Ecosystem Showcase . . . //FreeRTOS生态扩展组建

4.1 FreeRTOS+CLI//命令交互组建

4.1.1 About//关于FreeRTOS+CLI

4.1.2 Configuration and Use//配置和使用

4.1.3 Licensing//授权

4.1.4 Download//代码下载

4.1.5 实际工程移植步骤

见附录二

4.2 FreeRTOS+IO//通用IO组建

4.3 FreeRTOS+FAT SL//文件系统组建

4.4 FreeRTOS+UDP//网络组建

4.5 FreeRTOS+Trace//调试组建

附录一：修改记录

```
1. 时间：20150821    修改人：姚伟民    版本：V0.00.001    修改内容：初始版本
2. 时间：20150828    修改人：姚伟民    版本：V0.00.002    修改内容：增加 1.2.3的“关于任务认识”部分
                                                             增加 第二章第三章第四章目录
                                                             增加 附录二 附录三
```

附录二：将FreeRTOS-Plus-CLI项目加入工程使用步骤

http://blog.csdn.net/loveywm/article/details/47339357

FreeRTOS+CLI (Command Line Interface) provides a simple, small, extensible and RAM efficient method of enabling your FreeRTOS application to process command line input.

上面是官网关于FreeRTOS+CLI的简单介绍，在网上搜索了好久没看到相关人员的使用，于是凭着感觉开始将源码加入工程使用，下面是我的步骤经历。

开发环境：

1. 系统：win7
2. 开发IDE：CoIDE_V2Beta
3. 编译器：GNU Tools ARM Embedded\4.7 2014q2（arm-none-eabi-gcc）
4. 嵌入式环境：stm32f103vet6

第一步：首先建立一个简单的FreeRTOS工程，这个很简单，请自行参考网上资料。

第二步：下载FreeRTOSV8.2.1源代码并解压。

第三步：将“\FreeRTOSV8.2.1\FreeRTOS-Plus\Source\FreeRTOS-Plus-CLI”下的两个文件加入工程。其中在头文件“FreeRTOS_CLI.h”中加入一个没定义的数据（这个是我在编译的时候报错发现的）：

```
    #define configCOMMAND_INT_MAX_OUTPUT_SIZE 1000
```

第四步：在“\FreeRTOSV8.2.1\FreeRTOS-Plus\Demo\Common\FreeRTOS_Plus*CLI*Demos”是几个使用的demo，由于我使用的是串口，所以就使用“UARTCommandConsole.c”和“Sample-CLI-commands.c”，所以就将这两个文件也加入工程。

在““UARTCommandConsole.c””中修改参数：

```
#define cmdQUEUE_LENGTH 1000  这个数据是串口一下子输出的数据大小，尽量大点，
                              这样输出可以完全，太小的话可能串口输出不完全，这个是我实际使用发现的。
```

第五步：由于我使用的串口交互，那么需要底层的硬件支持，所以就是硬件驱动了。在“UARTCommandConsole.c”中引入了个头文件“#include "serial.h"”。此文件在FreeRTOS中的关于stm32f10X的demo中有这个头文件和.c文件的实现，直接拷贝过来然后加入工程。

第六步：按自己硬件修改“serial.c”文件。

1：修改头文件

```
    /* Library includes. */
    //#include "stm32f10x_lib.h"
    #include "stm32f10x_conf.h"
```

2：xSerialPortInitMinimal（）函数中对使用的串口初始化。

3：中断函数的修改

```
    //void vUARTInterruptHandler( void )
    void USART3_IRQHandler( void )
```

4：反正在“serial.c”中每个函数要根据自己硬件去修改，最好从头至尾检查一遍，防止一个函数参数和使用的硬件接口不同都可能照成使用不成功。

第七步：在main函数中加入相关初始化和CLI任务。

```
vRegisterSampleCLICommands();//此函数是官方提供的例子，可以不加入，如果加入注意里面的一些函数使用需要开启一些依靠的宏。
vUARTCommandConsoleStart( 1000, 1 );
```

第八步：编译无问题后下载和PC通信，设置好波特率，然后使用官方的几个例子测试。测试成功，以后就可以自己添加一下交互命令了。

我提供我的工程文件在github中。下面是工程目录局部视图：

附录三：如何在FreeRTOS-Plus-CLI中添加一个自己的命令行

http://blog.csdn.net/loveywm/article/details/47336257

> 第一作者：姚伟民

根据**附录**二添加完成后，下面就是添加一个自己的命令行。其实添加一个命令行格式很简单，就是按照demo中的数据格式添加一个结构体，然后在注册函数中注册就可以使用了。

第一步：添加自己的结构体

```
/* The structure that defines command line commands.    A command line command should be defined by declaring a const structure of this
typedef struct xCOMMAND_LINE_INPUT
{
     const char * const pcCommand; /* The command that causes pxCommandInterpreter to be executed.  For example "help".              Mus
     const char * const pcHelpString; /* String that describes how to use the command.  Should start with the command itself, and end wi
     const pdCOMMAND_LINE_CALLBACK pxCommandInterpreter;
/* A pointer to the callback function that will return the output generated by the command. */
     int8_t cExpectedNumberOfParameters; /* Commands expect a fixed number of parameters, which may be zero. */
} CLI_Command_Definition_t;
```

参考实际格式：

```
/* Structure that defines the "task-stats" command line command.  This generates
a table that gives information on each task in the system. */
static const CLI_Command_Definition_t xTaskStats =
{
     "task-stats", /* The command string to type. */
     "\r\ntask-stats:\r\n Displays a table showing the state of each FreeRTOS task\r\n",
      prvTaskStatsCommand, /* The function to run. */
     0 /* No parameters are expected. */
};
```

于是按葫芦画瓢自己写一个自己的命令“whatbook”吧！如下：

```
static const CLI_Command_Definition_t xWhatbook =
{
     "whatbook",
     "\r\nwhatbook:\r\n Displays a string "I Love ZML!"\r\n",
      prvWhatbookCommand, /* The function to run. */
     0 /* No parameters are expected. */
};
```

第二步：实现上面数据结构后，然后写数据结构中的回调函数。

```
static BaseType_t prvWhatbookCommand( char *pcWriteBuffer, size_t xWriteBufferLen, const char *pcCommandString )
{
     BaseType_t xReturn;
     /* Remove compile time warnings about unused parameters, and check the
     write buffer is not NULL.  NOTE - for simplicity, this example assumes the
     write buffer length is adequate, so does not check for buffer overflows. */
     ( void ) pcCommandString;
     ( void ) xWriteBufferLen;
     configASSERT( pcWriteBuffer );
     /* The first time the function is called after the command has been
     entered just a header string is returned. */
     sprintf( pcWriteBuffer, "I Love ZML!!!\r\n" );
     xReturn = pdFALSE;
     return xReturn;
}
```

第三步：注册数据结构。

```
FreeRTOS_CLIRegisterCommand( &xWhatbook );
```

至此，就完成了一个命令whatbook的实现。在串口工具输入whatbook，就返回如下打印信息：

```
>
I Love ZML!!!
[Press ENTER to execute the previous command again]
```

附录四：FreeRTOS