Study of an operating system: FreeRTOS

Nicolas Melot

Operating systems for embedded devices

Sommaire

Introduction

FreeRTOS is an free and open-source Real-Time Operating system developed by Real Time Engineers Ltd. Its design has been developed to fit on very small embedded systems and implements only a very minimalist set of functions: very basic handle of tasks and memory management, just sufficient API concerning synchronization, and absolutely nothing is provided for network communication, drivers for external hardware, or access to a filesystem. However, among its features are the following characteristics: preemptive tasks, a support for 23 micro-controller architectures[1] by its developers, a small footprint[2] (4.3Kbytes on an ARM7 after compilation[3]), written in C and compiled with various C compiler (some ports are compiled with gcc, others with openwatcom or borland c++). It also allows an unlimited number of tasks to run at the same time and no limitation about their priorities as long as used hardware can afford it. Finally, it implements queues, binary and counting semaphores and mutexes.

1 http://www.freertos.org/a00090.html
2 http://www.freertos.org/FAQMem.html#QSize
3 http://www.freertos.org/FreeRTOS_Features.html

1 Tasks

1.1 A task in FreeRTOS

FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks. In RTOS, a task is defined by a simple C function, taking a void* parameter and returning nothing (void).

Several functions are available to manage tasks: task creation (vTaskCreate()), destruction (vTaskDelete()), priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()). More options are available to user, for instance to create a critical sequence or monitor the task for debugging purpose.

1.1.1 Life cycle of a task

This section will describe more precisely how can a task evolve from the moment it is created to when it is destroyed. In this context, we will consider to be available only one micro-controller core, which means only one calculation, or only one task, can be run at a given time. Any given task can be in one of two simple states : “running” or “not running”. As we suppose there is only one core, only one task can be running at a given time; all other tasks are in the “not running task. Figure 1 gives a simplified representation of this life cycle. When a task changes its state from “Not running” to running, it is said “swapped in” or “switched in” whereas it is called “swapped out” or “switched out” when changing to “Not running” state.

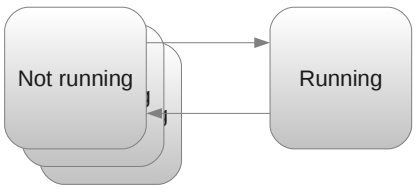

Figure 1: Simplified life cycle of a task : Only one task can be "running" at a given time, whereas the “not running state can be expanded“.

As there are several reasons for a task not to be running, the “Not running” state can be expanded as shows Figure 2. A task can be preempted because of a more priority task (scheduling is described in section 2), because it has been delayed or because it waits for a event. When a task can runs but is waiting for the processor to be available, its state is said “Ready”. This can happen when a task has it needs everything to run but there is a more priority task running at this time. When a task is delayed or is waiting for another task (synchronisation through semaphores or mutextes) a task is said to be “Blocked”. Finally, a call to vTaskSuspend() and vTaskResume() or xTaskResumeFromISR() makes the task going in and out the state “Suspend”.

It is important to underline that a if a task can leave by itself the “Running” state (delay, suspend or wait for an event), only the scheduler can “switch in” again this task. When a task wants to run again, its state turns to “Ready” an only the scheduler can choose which “Ready” task is run at a given time.

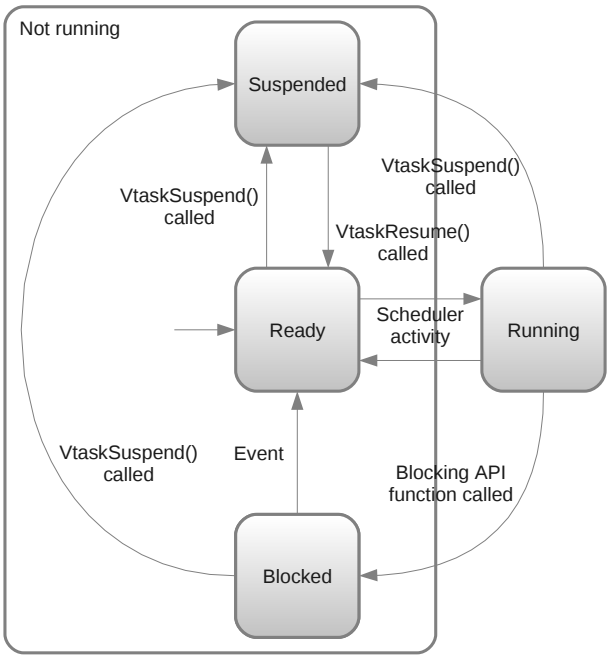

Figure 2: Life cycle of a task

1.2 Creating and deleting a task

A task defined by a simple C function, taking one void* argument and returning nothing (see Text 1)

```
void ATaskFunction( void *pvParameters );
```

Text 1: A typical task signature

Any created task should never end before it is destroyed. It is common for task's code to be wrapped in an infinite loop, or to invoke vTaskDestroy(NULL) before it reaches its final brace. As any code in infinite loop can fail and exit this loop, it is safer even for a repetitive task, to invoke vTaskDelete() before its final brace. An example of a typical task implementation is available on Text 3.

A task can be created using vTaskCreate() (Text 2). This function takes as argument the following list:

- pvTaskCode: a pointer to the function where the task is implemented.
- pcName: given name to the task. This is useless to FreeRTOS but is intented to debugging purpose only.
- usStackDepth: length of the stack for this task in **words**. The actual size of the stack depends on the micro controller. If stack with is 32 bits (4 bytes) and usStackDepth is 100, then 400 bytes (4 times 100) will be allocated for the task.
- pvParameters: a pointer to arguments given to the task. A good practice consists in creating a dedicated

structure, instantiate and fill it then give its pointer to the task.

- uxPriority: priority given to the task, a number between 0 and MAX_PRIORITIES – 1. This is discussed in section 2.
- pxCreatedTask: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

Text 2: Task creation routine

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static – in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Text 3: A typical task (from "Using the FreeRTOS Real Time Kernel").

A task is destroyed using xTaskDestroy() routine. It takes as argument pxCreatedTask which is given when the task was created. Signature of this routine is given in Text 4 and an example can be found in Text 3.

```
void vTaskDelete( xTaskHandle pxTask );
```

Text 4: Deleting a task

When a task is deleted, it is responsibility of idle task to free all allocated memory to this task by kernel. Notice that all memory dynamically allocated must be manually freed.

2 Scheduling

Task scheduling aims to decide which task in “Ready” state has to be run at a given time. FreeRTOS achieves this purpose with priorities given to tasks while they are created (see 1.2). Priority of a task is the only element the scheduler takes into account to decide which task has to be switched in.

Every clock tick makes the scheduler to decide which task has to be waken up, as shown in Figure 3.

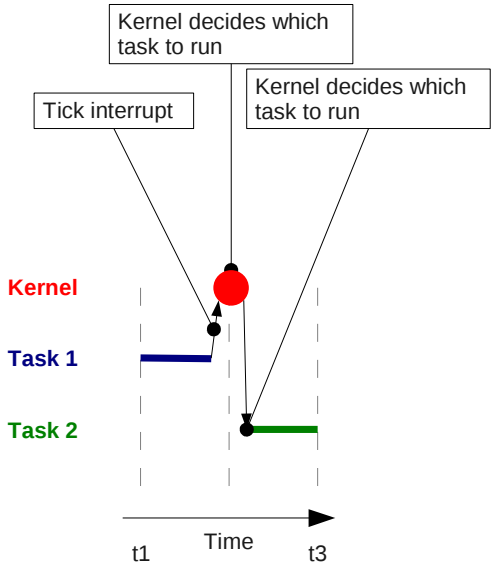

Figure 3: Every clock tick makes the scheduler to run a "Ready" state task and to switch out the running task.

2.1 Priorities

FreeRTOS implements tasks priorities to handle multi tasks scheduling. A priority is a number given to a task while it is created or changed manually using vTaskPriorityGet() and vTaskPrioritySet() (See FreeRTOS manual). There is no automatic management of priorities which mean a task always keeps the same priority unless the programmer change it explicitly. A low value means a low priority: A priority of 0 is the minimal priority a task could have and this level should be strictly reserved for the idle task. The last available priority in the application (the higher value) is the highest priority available for task. FreeRTOS has no limitation concerning the number of priorities it handles. Maximum number of priorities is defined in MAX_PRIORITIES constant in FreeRTOSConfig.h (see section 8.1), and hardware limitation (width of the MAX_PRIORITIES type). If an higher value is given to a task, then FreeRTOS cuts it to MAX_PRIORITIES – 1. Figure 4 gives an example of a application run in FreeRTOS. Task 1 and task 3 are event-based tasks (they start when a event occurs, run then wait for the event to occur again), Task 2 is periodic and idle task makes sure there is always a task running.

This task management allows an implementation of Rate Monotonic for task scheduling: tasks with higher frequencies are given an higher priority whereas low frequencies tasks deserve a low priority. Event-based or continuous tasks are preempted by periodic tasks.

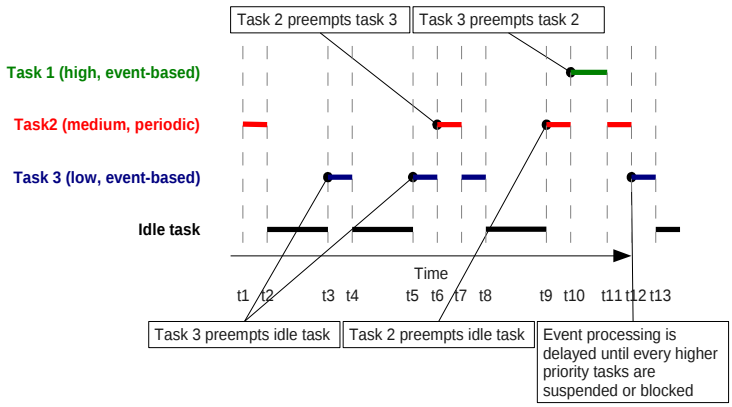

Figure 4: An hypothetic FreeRTOS application schedule

2.2 Priority-equally tasks

Tasks created with an equal priority are treated equally by the scheduler: If two of them are ready to run, the scheduler shares running time among all of them: at each clock tick, the scheduler chooses a different task among the ready tasks with highest priority. This implements a Round Robin implementation where quantum is the time between each clock tick. This value is available in TICK_RATE_HZ constant, in FreeRTOSConfig.h (section 8.1).

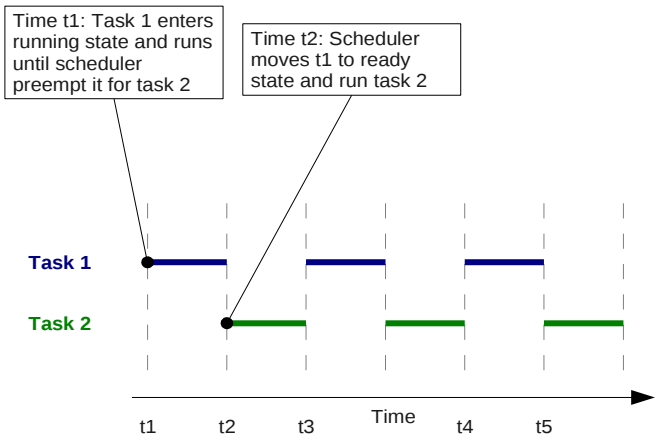

Figure 5: Two tasks with a equivalent priority are run after each other in turn.

2.3 Starvation

There is no mechanism implemented in FreeRTOS that prevents task starvation: the programmer has to make sure there is no higher priority task taking all running time for itself. It is also a good idea to let the idle task to run, since it can handle some important work such as free memory from deleted tasks, or switching the device into a sleeping mode.

3 Queue management

Queues are an underlying mechanism beyond all tasks communication or synchronization in a FreeRTOS environment. They are an important subject to understand as it is unavoidable to be able to build a complex application with tasks cooperating with each other. They are a mean to store a and finite number (named “length”) of fixed-size

data. They are able to be read and written by several different tasks, and don't belong to any task in particular. A queue is normally a FIFO which means elements are read in the order they have been written. This behavior depends on the writing method: two writing functions can be used to write either at the beginning or at the end of this queue.

3.1 Reading in a queue

When a single task reads in a queue, it is moved to “Blocked” state and moved back to “Ready” as soon as data has been written in the queue by another task or an interrupt. If several tasks are trying to read a queue, the highest priority task reads it first. Finally, if several tasks with the same priority are trying to read, the first task who asked for a read operation is chosen. A task can also specify a maximum waiting time for the queue to allow it to be read. After this time, the task switches back automatically to “Ready” state.

```
portBASE_TYPE xQueueReceive(
                                xQueueHandle xQueue,
                                const void * pvBuffer,
                                portTickType xTicksToWait
                            );
```

Text 5: normal method to read in a queue: it reads an element then removes it.

xqueue is the identifier of the queue to be read

pvBuffer is a pointer to the buffer where the read value will be copied to. This memory must be allocated and must be large enough to handle the element read from the queue.

xTicksToWait defines the maximum time to wait. 0 prevents the task from waiting even if a value is not available, whereas if INCLUDE_vTaskSuspend is set and xTicksToWait equals MAX_DELAY, the task waits indefinitely.

pdPASS is returned if a value was sucessfully read before xTicksToWait is reached. If not, errQUEUE_EMPTY is returned from xQueueReceive().

After reading an element in a queue, this element is normally removed from it; however, an other read function given in allows to read an element without having it to be deleted from the queue.

```
portBASE_TYPE xQueuePeek(
                                xQueueHandle xQueue,
                                const void * pvBuffer,
                                portTickType xTicksToWait
                            );
```

Text 6: It also possible to read in a queue without without removing the element from it.

3.2 Writing to a queue

Writing on a queue obeys to the same rules as reading it. When a task tries to write on a queue, it has to wait for it to have some free space: the task is blocked until another task reads the queue and free some space. If several tasks

attempt to write on the same queue, the higher priority task is chosen first. If several tasks with the same priority are trying to write on a queue, then the first one to wait is chosen. Figure 6 gives a good illustration on how queues work.

A prototype is available on Text 7. It describes the normal method to write on a queue. Text 8 gives the underlying function behind xQueueSend and the function to be used if the user wants the last written element to be read first (Last In, First Out or LIFO).

```
portBASE_TYPE xQueueSend(  xQueueHandle xQueue,
                                 const void * pvItemToQueue,
                                 portTickType xTicksToWait
                               );
```

Text 7: function to write on a queue in FIFO mode

xQueue is the queue to write on. This value is returned by the queue creation method.

pvItemToQueue is a pointer to an element which is wanted to be copied (by value) to the queue.

xticksToWait is the number of ticks to wait before the task gives up to write on this queue. If xTicksToWait is 0, the task won't wait at all if the queue is full. If INCLUDE_vTaskSuspend if defined to 1 inf FreeRTOSConfig.h (section 8.1) and xTicksToWait equals MAX_DELAY, then the task has no time limit to wait.

XqueueSend returns pdPASS if the element was successfully written to the queue before the maximum waiting time was reached, or errQUEUE_FULL if the maximum time was elapsed before the task could write on the queue.

```
portBASE_TYPE xQueueSendToBack(  xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait
                                 );
portBASE_TYPE xQueueSendToFront(   xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait
                                 );
```

Text 8: xQueueSendToBack: a synonym for xQueSend; xQueueSendToFront write on a queue in LIFO mode.

3.3 Creating a queue

Length of a queue and its width (the size of its elements) are given when the queue is created. Text 9 gives the function signature available to create a queue.

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                         );
```

Text 9: Queue creation function

uxQueueLenght gives the number of elements this queue will be able to handle at any given time. uxItemSize is the size in byte of any element stored in the queue. xQueueCreate returns NULL if the queue was not created due to lack

of memory available; if not, the returned value should be kept to handle the newly created queue.

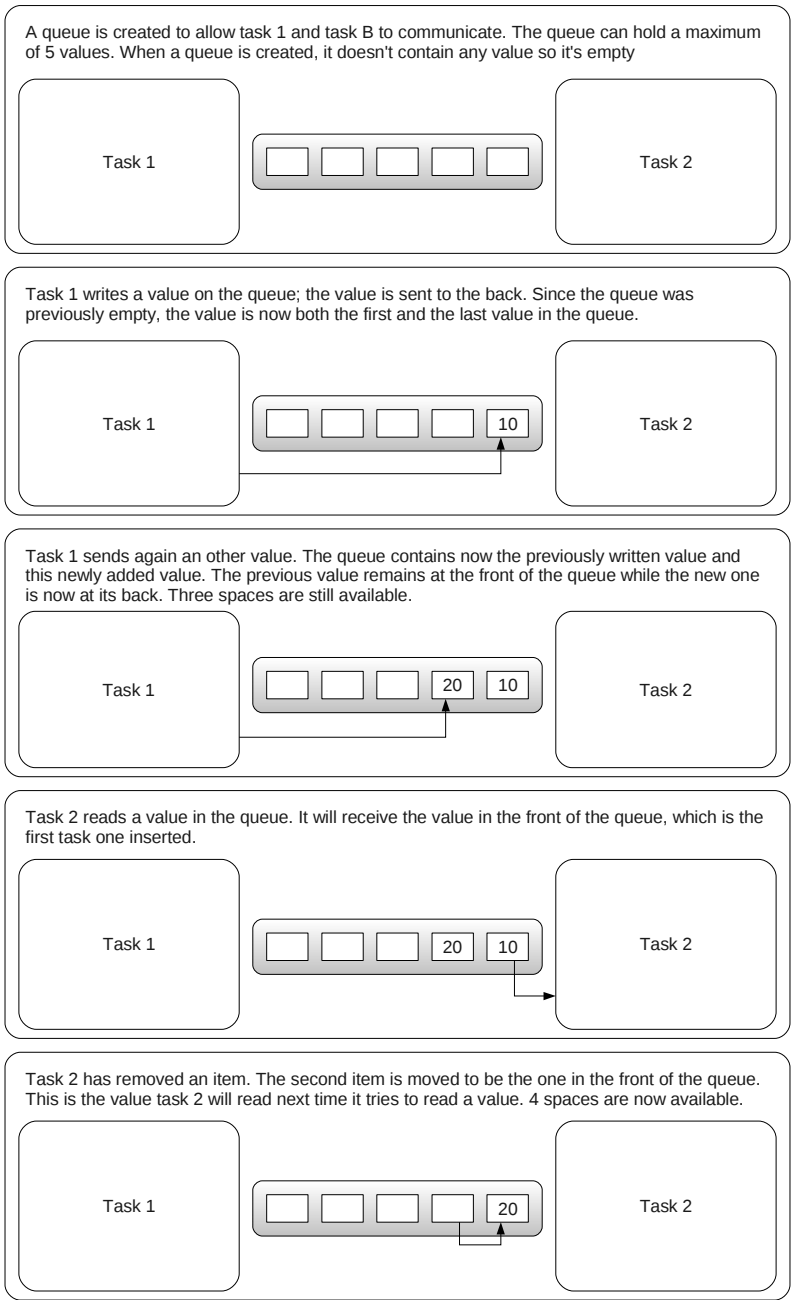

Figure 6: Possible scenario with a queue and two tasks

4 Resources management

4.1 Binary semaphores

Binary semaphores are the simplest effective way to synchronize tasks, an other even more simple, but not as effective, consists in polling an input or a resource. A binary semaphore can be seen as a queue which contains only one element. Figure 7 gives an idea on its mechanism.

4.1.1 Handle binary semaphores

4.1.1.1 Creation of a semaphore

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

Text 10: creating a semaphore

xSemaphore: semaphore to be created.

4.1.1.2 Taking a semaphore

This operation is equivalent to a P() operation, or if compared to queues, to a Receive() operation. A task taking the semaphore must wait it to be available and is blocked until it is or until a delay is elapsed (if applicable).

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType
xTicksToWait );
```

Text 11: taking a semaphore

xSsemaphore is the semaphore to take.

xTicksToWait is the time, in clock ticks, for the task to wait before it gives up with taking the semaphore. If xTicksToWait equals MAX_DELAY and INCLUDE_vTaskSuspend is 1, then the task won't stop waiting.

If the take operation succeed in time, the function returns pdPASS. If not, pdFALSE is returned.

4.1.1.3 Giving a semaphore

Giving a semaphore can be compared to a V() operation or to writing on a queue.

```
portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

Text 12: giving a semaphore

xSemaphore is the semaphore to be given.

The function returns pdPASS if the give operation was successful, or pdFAIL if the semaphore was already available, or if the task did not hold it.

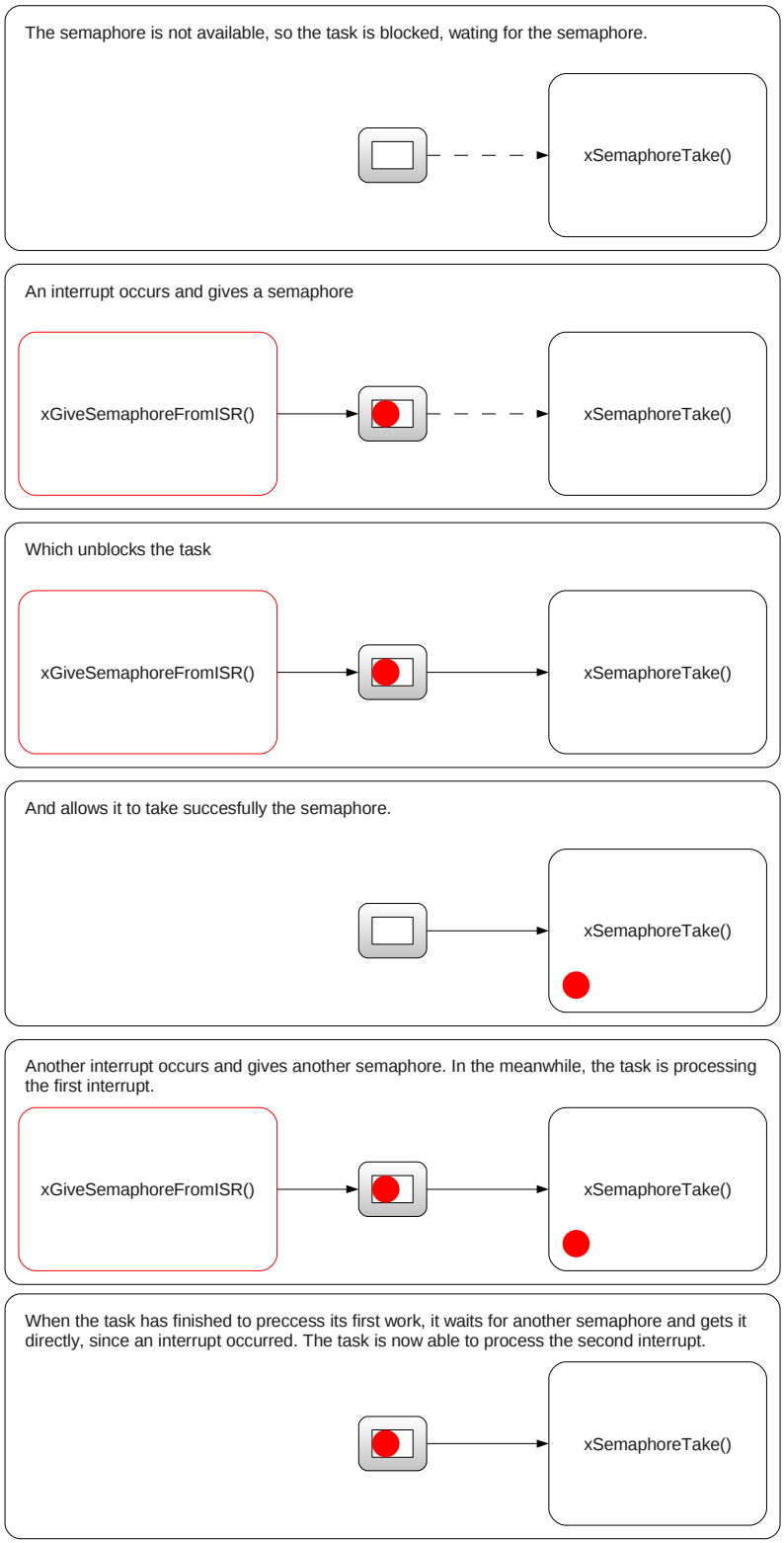

Figure 7: A binary semaphore is equivalent to a queue which can contain one element

4.2 Mutexes

Mutexes are designed to prevent mutual exclusion or deadlocking. A mutex is used similarly to a binary semaphore, except the task which take the semaphore must give it back. This can be though with a token associated with the resource to access to. A task holds the token, works with the resource then gives back the token; in the meanwhile, no other token can be given to the mutex. A good illustration is shown in Figure 8.

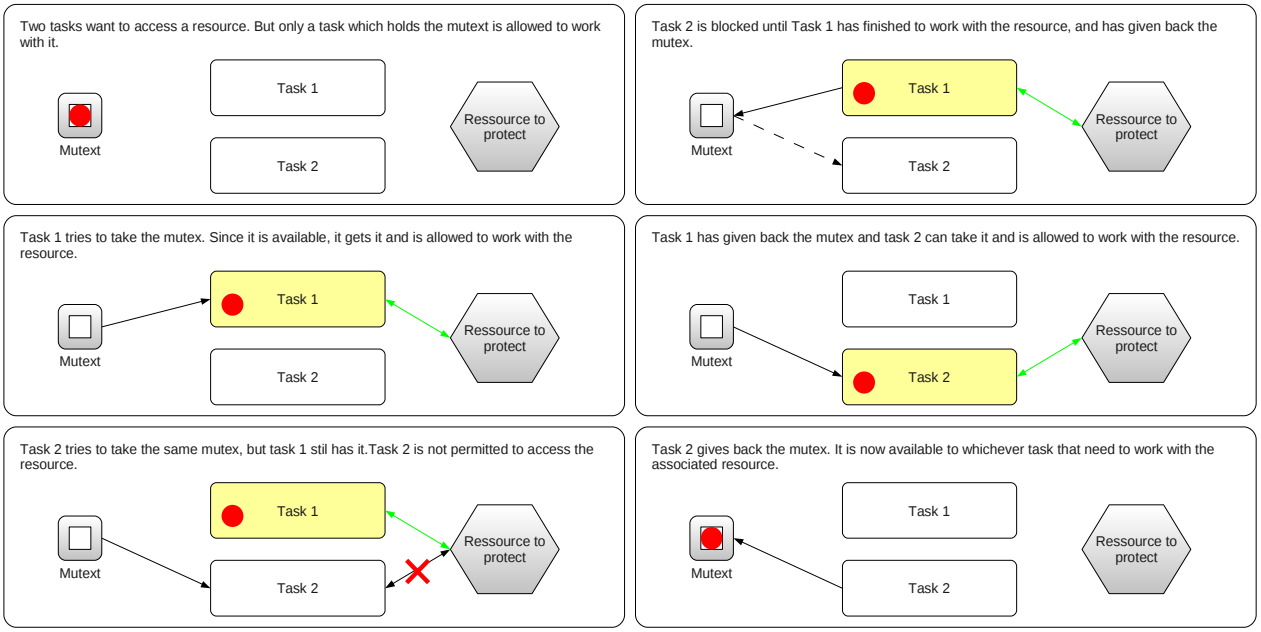

Figure 8: Usual use case of a mutex

4.2.1 Priority inheritance

Priority inheritance is actually the only difference between a binary semaphore and a mutex. When several tasks asks for a mutex, the mutex holder's priority is set to the highest waiting task priority. This mechanism helps against priority inversion phenomenon although it doesn't absolutely prevent it from happening. The use of a mutex raises the application global complexity and therefore should be avoided whenever it is possible.

4.3 Counting semaphores

A counting semaphore is a semaphore that can be taken several (but limited) times before is becomes unavailable. It maintains a value which is increased as the semaphore is given, and decreased when is is taken. Is is comparable to a queue with a certain amount of elements. When created, a counting semaphore can be initialized to be available an arbitrary number of times.

4.3.1 Counting semaphore routines

4.3.1.1 Creation

As described above, a counting semaphore can be taken a limited maximum times and is initialized to be available for an arbitrary number of take operations. These characteristics are given when the semaphore is created.

```
 xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount,
                                            unsigned portBASE_TYPE
uxInitialCount );
```

Text 13: Creation of a counting semaphore.

uxMaxCount is the capacity of the counting semaphore, its maximum ability to be taken.

uxInitialCount is the new semaphore's availability after it is created.

Returned value is NULL if the semaphore was not created, because of a lack of memory, or a pointer to the new semaphore and can be used to handle it.

4.3.1.2 Take & give operations

P() and V() operation to counting semaphores are realized using the same function as the one described in sections 4.1.1.2 and 4.1.1.3.

5 Handling interrupts

An interrupt is a mechanism fully implemented and handled by hardware. Software and more particularly FreeRTOS tasks or kernel can only give methods to handle a given interrupt, or it can raise some by calling an hardware instruction. We will suppose we are using a micro controller that handles 7 different levels of interrupts. The more an interrupt number is important, the more it will be priority over other interrupts. Depending on hardware, this is not always the case. interrupts priorities are not, in any case, related to tasks priorities, and will always preempt them.

A function defined as an interrupt handler cannot use freely FreeRTOS API: access to queues or semaphores is forbidden through the normal functions described in previous section, but FreeRTOS provides some specialized functions to be used in that context: for instance, in an interrupt handler, a V() operation to a semaphore must be realized using xSemaphoreGiveFromISR() instead of xSemaphoreGive(). The prototypes for these method can be different as they can involve some particular problems (this is the case of xSemaphoreGiveFromISR() which implements a mechanism to make the user to be aware that this give operation makes the interrupt to be preempted by a higher priority interrupt unlocked by this give operation).

Interrupt management can be configured in FreeRTOS using constants available in FreeRTOSConfig.h.

- configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority level for the tick interrupt.
- configMAX_SYSCALL_INTERRUPT_PRIORITY defines the highest interrupt level available to interrupts that use interrupt-safe FreeRTOS API functions. If this constant is not defined, then any interrupt handler function that makes a use of FreeRTOS API must execute at configKERNEL_INTERRUPT_PRIORITY.

Any interrupt whose priority level is greater than configMAX_SYSCALL_INTERRUPT_PRIORITY or configKERNEL_INTERRUPT_PRIORITY if configMAX_SYSCALL_INTERRUPT_PRIORITY is not defined, will

never be preempted by the kernel, but are forbidden to use FreeRTOS API functions.

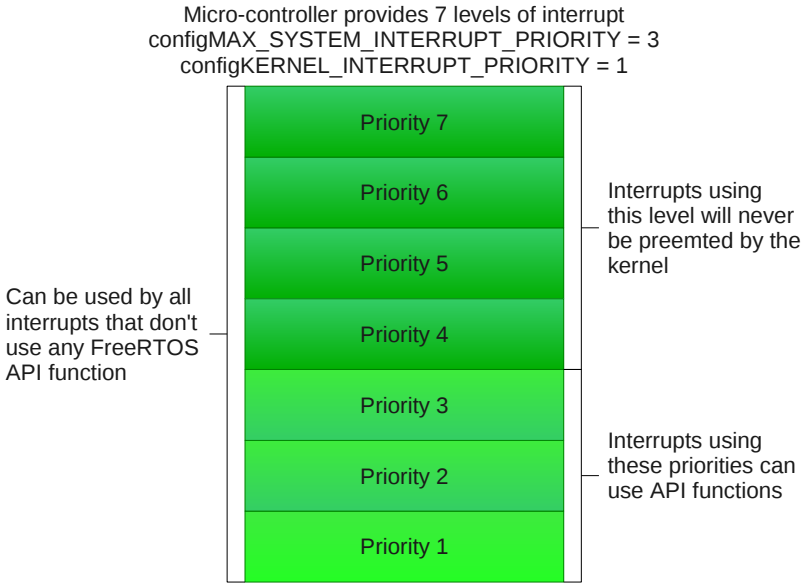

Figure 9: Interrupt organization in FreeRTOS

5.1 Manage interrupts using a binary semaphore

Interrupt handlers are pieces of code run by the micro-controller and therefore are not handled by FreeRTOS. This can potentially create problems with memory access since the operating system cannot handle these context changes. This is a reason why several functions exists in two versions: one for regular tasks and another is intended to interrupts handler. This is the case of queue management functions like xQueueReceive() and wQueueReceiveFromISR(). For this reason, it is necessary to make interrupts handlers' execution as short as possible. On way to achieve this goal consists in the creation of tasks waiting for an interrupt to occur with a semaphore, and let this safer portion of code actually handle the interrupt.

Figure 10 proposes a solution to reduce significantly the time an ISR can run. An ISR “gives” a semaphore and unblock a 'Handler” task that is able to handler the ISR, making the ISR execution much shorter.

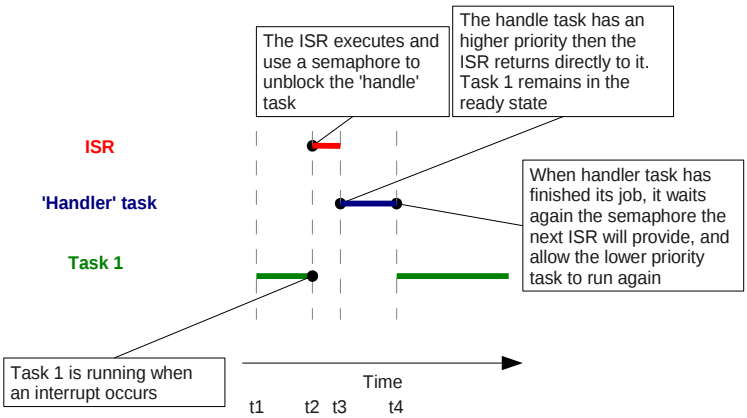

Figure 10: Deferred interrupt processing: a regular tasks waits for an interrupt to occur with a semaphore, and handle it.

5.2 Critical sections

Sometimes a portion of code needs to be protected from any context change so as to prevent a calculation from being corrupted or an I/O operation being cut or mixed with another. FreeRTOS provides two mechanisms to protect some as small portions as possible; some protects from any context change, either from a scheduler operation, or an interrupt event, others only prevents scheduler from preempting the task.

Handling this can be very important as many instructions, affectations for instance, may look atomic but require several hardware instructions (load variable address to a registry, load a value to another registry and move the value to the matching memory address using the two registries).

5.2.1 Suspend interrupts

This form or critical section is very efficient but must be kept as short as possible since it makes the whole system in such a state that any other portion of code cannot be executed. This can be a problem for a task to meet its time constraint, or an external event to be treated by an interruption.

```
/* Ensure access to the PORTA register cannot be interrupted by
   placing it within a critical section.  Enter the critical section. */
   taskENTER_CRITICAL();

   /* A switch to another task cannot occur between the call to
   taskENTER_CRITICAL() and the call to taskEXIT_CRITICAL().  Interrupts
   may still execute on FreeRTOS ports that allow interrupt nesting, but
   only interrupts whose priority is above the value assigned to the
   configMAX_SYSCALL_INTERRUPT_PRIORITY constant — and those interrupts are
   not permitted to call FreeRTOS API functions. */
   PORTA |= 0x01;

   /* We have finished accessing PORTA so can safely leave the critical
   section. */
   taskEXIT_CRITICAL();
```

Text 14: A critical section protected against both scheduler "switch out" operations, and hardware interrupts.

A task can start a critical section with taskENTER_CRITICAL() and stop it using taskEXIT_CRITICAL(). The system allow a critical section to be started while an other one is already opened: this makes much easier to call external functions that can need such a section whereas the calling function also need it. However, it is important to notice that in order to end a critical section, taskEXIT_CRITICAL() must be called exactly as much as taskSTART_CRITICAL was. Generaly speaking, these two functions must be called as close as possible in the code to make this section very short.

Such a critical section is not protected from interrupts which priority is greater than configMAX_SYSCALL_INTERRUPT_PRIORITY (if defined in FreeRTOSConfig.h; if not, prefer to consider the value configKERNEL_INTERRUPT_PRIORITY instead) to create a context change.

5.2.2 Stop the scheduler

A less drastic method to create a critical section consists in preventing any task from preempting it, but let interrupts to do their job. This goal can be achieve by preventing any task to leave the “Ready” state to “Running”, it can be understood as stopping the scheduler, or stopping all the tasks.

Notice it is important that FreeRTOS API functions must not be called when the scheduler is stopped.

```
/* Write the string to stdout, suspending the scheduler as a method
 of mutual exclusion. */
 vTaskSuspendAll();
 {
     printf( "%s", pcString );
     fflush( stdout );
 }
 xTaskResumeAll();
```

Text 15: Creation of a counting semaphore.

When Calling xTaskResumeAll() is called, it returns pdTRUE if no task requested a context change while scheduler was suspended and returns pdFALSE if there was.

6 Memory management

In a small embedded system, using malloc() and free() to allocate memory for tasks, queues or semaphores can cause various problems: preemption while allocating some memory, memory allocation and free can be an nondeterministic operations, once compiled, they consume a lot of space or suffer from memory fragmentation.

Instead, FreeRTOS provides three different ways to allocate memory, each adapted to a different situation but all try to provide a solution adapted to small embedded systems. Once the proper situation identified, the programmer can choose the right memory management method once for all, for kernel activity included. It is possible to implement its own method, or use one of the three FreeRTOS proposes and which can be found in heap_1.c, heap_2.c or heap_3.c (or respecctively in sections 8.2, 8.3 and 8.4).

6.1 Prototypes

All implementations respect the same allocation/free memory function prototypes. These prototypes stands in two functions.

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

Text 16: Prototypes for memory allocation/deallocation

xWanted size is the size, in byte, to be allocated, pv is a pointer to the memory to be freed. pvPortMalloc returns a pointer to the memory allocated.

6.2 Memory allocated once for all

It is possible in small embedded systems, to allocate all tasks, queues and semaphores, then start the scheduler and run the entire application, which will never have to reallocate free any of structures already allocated, or allocate some new. This extremely simplified case makes useless the use of a function to free memory: only pvPortMalloc is implemented. This implementation can be found in Source/portable/MemMang/heap_1.c or appendix 8.2

Since the use of this scheme suppose all memory is allocated before the application actually starts, and there will have no need to reallocate or free memory, FreeRTOS simply adds a task TCB (Task Control Block, the structure FreeRTOS uses to handle tasks) then all memory it needs, and repeat this job for all implemented tasks. Figure 11 gives a illustration about how the memory is managed.

This memory management allocates a simple array sized after the constant configTOTAL_HEAP_SIZE in FreeRTOSConfig.h, and divides it in smaller parts which are allocated for memory all tasks require. This makes the application to appear to consume a lot of memory, even before any memory allocation.

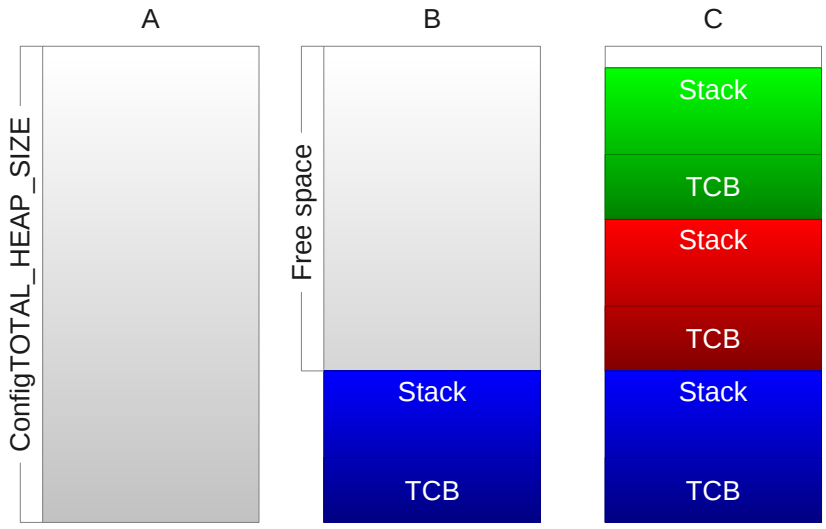

Figure 11: In A: no memory is allocated yet; in B, memory has been allocated for blue task; in C, all required memory is allocated

6.3 Constant sized and numbered memory

An application can require to allocate and deallocation dynamically memory. If in every tasks' life cycle, number of variables and it's size remains constant, then this second mechanism can be set up. Its implementation can be found in Source/portable/MemMang/heap_2.c or appendix 8.3.

As the previous strategy, FreeRTOS uses a large initial array, which size depends on configTOTAL_HEAP_SIZE and makes the application to appears to consume huge RAM. A difference with the previous solution consists in an implementation of vPortFree(). As memory can be freed, the memory allocation is also adapted. Let's consider the big initial array to be allocated and freed in such a way that there are three consecutive free spaces available. First is 5 bytes, second is 25 and the last one is 100 bytes large. A call to pvPortMalloc(20) requires 20 bytes to be free so has to reserve

it and return back its reference. This algorithm will return the second free space, 25 bytes large and will keep the remaining 5 bytes for a later call to pvPortMalloc(). It will always choose the smallest free space where can fit the requested size.

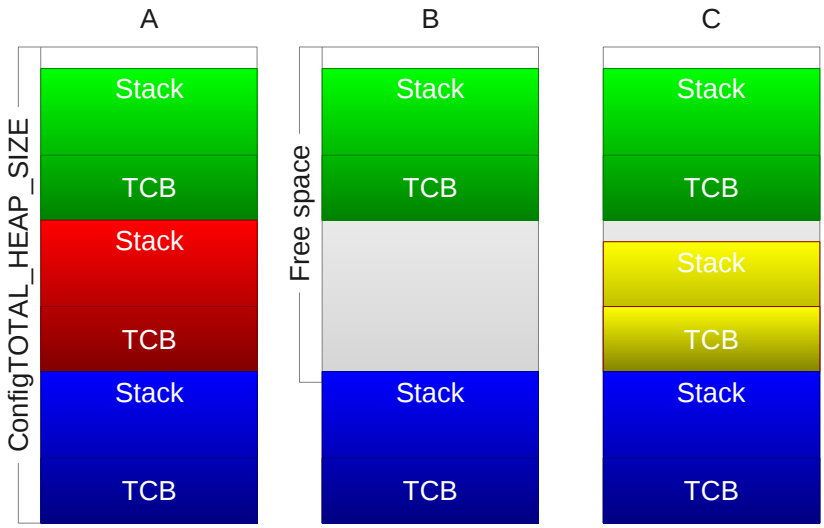

Figure 12: The algorithm will always use the smallest free space where the requested portion can fit.

Such an algorithm can generate a lot of fragmentation in memory if allocations are not regular, but it fits if allocations remains constant in size and number.

6.4 Free memory allocation and deallocation

This last strategy makes possible every manipulation, but suffers from the same drawbacks as using malloc() and free(): large compiled code or nondeterministic execution. This implementation wraps the two functions, but make them thread safe by suspending the scheduler while allocating or deallocating. Text 17 And section 8.4 give the implementation for this memory management strategy.

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;
    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
    }
    xTaskResumeAll();
    return pvReturn;
}

void vPortFree( void *pv )
{
    if( pv != NULL )
    {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

Text 17: Thread safe wrappers for malloc() and free() are another solution to manage memory.

Conclusion

FreeRTOS is an operating system designed for small embedded system: but if its memory footprint can be very small, its functionalities are also very limited: no support for thread, minimalist memory management, no driver is available to handle resources on usual bus such as USB or PCI, no support for any communication protocols such as an IP stack and nothing is available to handle any file system; even input-output primitives are not available. However basic functions of an operating system are implemented; this is enough for the design of very small and rather complex applications.

References

This works makes references to FreeRTOS documentation books “Using the FreeRTOS Real Time kernel” available to download on http://www.freertos.org/a00104.html. Illustrations used in this report can be found in this book. This report also makes reference to FreeRTOS API published on http://www.freertos.org/a00106.html and on the book FreeRTOS Reference manual http://www.freertos.org/a00104.html.

7 Illustrations

8 Appendix

8.1 An example of FreeRTOSConfig.h

```
/*
    FreeRTOS V6.0.0 - Copyright (C) 2009 Real Time Engineers Ltd.

    ***************************************************************************
    *                                                                         *
    * If you are:                                                             *
    *                                                                         *
    *    + New to FreeRTOS,                                                   *
    *    + Wanting to learn FreeRTOS or multitasking in general quickly       *
    *    + Looking for basic training,                                        *
    *    + Wanting to improve your FreeRTOS skills and productivity           *
    *                                                                         *
    * then take a look at the FreeRTOS eBook                                  *
    *                                                                         *
    *        "Using the FreeRTOS Real Time Kernel - a Practical Guide"        *
    *                  http://www.FreeRTOS.org/Documentation                  *
    *                                                                         *
    * A pdf reference manual is also available.  Both are usually delivered   *
    * to your inbox within 20 minutes to two hours when purchased between 8am *
    * and 8pm GMT (although please allow up to 24 hours in case of            *
    * exceptional circumstances).  Thank you for your support!                *
    *                                                                         *
    ***************************************************************************

    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    ***NOTE*** The exception to the GPL is included to allow you to distribute
    a combined work that includes FreeRTOS without being obliged to provide the
    source code for proprietary components outside of the FreeRTOS kernel.
    FreeRTOS is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    http://www.FreeRTOS.org - Documentation, latest information, license and
    contact details.

    http://www.SafeRTOS.com - A version that is certified for use in safety
    critical systems.

    http://www.OpenRTOS.com - Commercial support, development, porting,
    licensing and training services.
*/

#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
```

```
#include <i86.h>
#include <conio.h>

/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK                   1
#define configUSE_TICK_HOOK                   1
#define configTICK_RATE_HZ                    ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE        ( ( unsigned short ) 256 ) /* This can
be made smaller if required. */
#define configTOTAL_HEAP_SIZE           ( ( size_t ) ( 32 * 1024 ) )
#define configMAX_TASK_NAME_LEN               ( 16 )
#define configUSE_TRACE_FACILITY        1
#define configUSE_16_BIT_TICKS          1
#define configIDLE_SHOULD_YIELD               1
#define configUSE_CO_ROUTINES                 1
#define configUSE_MUTEXES                     1
#define configUSE_COUNTING_SEMAPHORES   1
#define configUSE_ALTERNATIVE_API       1
#define configUSE_RECURSIVE_MUTEXES     1
#define configCHECK_FOR_STACK_OVERFLOW  0 /* Do not use this option on the PC
port. */
#define configUSE_APPLICATION_TASK_TAG  1
#define configQUEUE_REGISTRY_SIZE       0

#define configMAX_PRIORITIES       ( ( unsigned portBASE_TYPE ) 10 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet        1
#define INCLUDE_uxTaskPriorityGet       1
#define INCLUDE_vTaskDelete             1
#define INCLUDE_vTaskCleanUpResources   1
#define INCLUDE_vTaskSuspend            1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_uxTaskGetStackHighWaterMark 0 /* Do not use this option on the
PC port. */


/* An example "task switched in" hook macro definition. */
#define traceTASK_SWITCHED_IN() xTaskCallApplicationTaskHook( NULL, ( void * )
0xabcd )
```

```
extern void vMainQueueSendPassed( void );
#define traceQUEUE_SEND( pxQueue ) vMainQueueSendPassed()

#endif /* FREERTOS_CONFIG_H */
```

8.2 heap_1.c

```
/*
    FreeRTOS V6.0.0 - Copyright (C) 2009 Real Time Engineers Ltd.

    ***************************************************************************
    *                                                                         *
    * If you are:                                                             *
    *                                                                         *
    *    + New to FreeRTOS,                                                   *
    *    + Wanting to learn FreeRTOS or multitasking in general quickly       *
    *    + Looking for basic training,                                        *
    *    + Wanting to improve your FreeRTOS skills and productivity           *
    *                                                                         *
    * then take a look at the FreeRTOS eBook                                  *
    *                                                                         *
    *        "Using the FreeRTOS Real Time Kernel - a Practical Guide"        *
    *                  http://www.FreeRTOS.org/Documentation                  *
    *                                                                         *
    * A pdf reference manual is also available.  Both are usually delivered   *
    * to your inbox within 20 minutes to two hours when purchased between 8am *
    * and 8pm GMT (although please allow up to 24 hours in case of            *
    * exceptional circumstances).  Thank you for your support!                *
    *                                                                         *
    ***************************************************************************

    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    ***NOTE*** The exception to the GPL is included to allow you to distribute
    a combined work that includes FreeRTOS without being obliged to provide the
    source code for proprietary components outside of the FreeRTOS kernel.
    FreeRTOS is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    http://www.FreeRTOS.org - Documentation, latest information, license and
    contact details.

    http://www.SafeRTOS.com - A version that is certified for use in safety
    critical systems.

    http://www.OpenRTOS.com - Commercial support, development, porting,
    licensing and training services.
```

```
*/


/*
 * The simplest possible implementation of pvPortMalloc().  Note that this
 * implementation does NOT allow allocated memory to be freed again.
 *
 * See heap_2.c and heap_3.c for alternative implementations, and the memory
 * management pages of http://www.FreeRTOS.org for more information.
 */
#include <stdlib.h>

/* Defining MPU_WRAPPERS_INCLUDED_FROM_API_FILE prevents task.h from redefining
all the API functions to use the MPU wrappers.  That should only be done when
task.h is included from an application file. */
#define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

#include "FreeRTOS.h"
#include "task.h"

#undef MPU_WRAPPERS_INCLUDED_FROM_API_FILE

/* Allocate the memory for the heap.  The struct is used to force byte
alignment without using any non-portable code. */
static union xRTOS_HEAP
{
	#if portBYTE_ALIGNMENT == 8
		volatile portDOUBLE dDummy;
	#else
		volatile unsigned long ulDummy;
	#endif
	unsigned char ucHeap[ configTOTAL_HEAP_SIZE ];
} xHeap;

static size_t xNextFreeByte = ( size_t ) 0;
/*-----------------------------------------------------------*/

void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn = NULL;

	/* Ensure that blocks are always aligned to the required number of bytes.
*/
	#if portBYTE_ALIGNMENT != 1
		if( xWantedSize & portBYTE_ALIGNMENT_MASK )
		{
			/* Byte alignment required. */
			xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
		}
	#endif

	vTaskSuspendAll();
	{
		/* Check there is enough room left for the allocation. */
		if( ( ( xNextFreeByte + xWantedSize ) < configTOTAL_HEAP_SIZE ) &&
			( ( xNextFreeByte + xWantedSize ) > xNextFreeByte )	)/* Check
for overflow. */
```

```
            {
                  /* Return the next free byte then increment the index past
this
                  block. */
                  pvReturn = &( xHeap.ucHeap[ xNextFreeByte ] );
                  xNextFreeByte += xWantedSize;
            }
      }
      xTaskResumeAll();

      #if( configUSE_MALLOC_FAILED_HOOK == 1 )
      {
            if( pvReturn == NULL )
            {
                  extern void vApplicationMallocFailedHook( void );
                  vApplicationMallocFailedHook();
            }
      }
      #endif

      return pvReturn;
}
/*-----------------------------------------------------------*/

void vPortFree( void *pv )
{
      /* Memory cannot be freed using this scheme.  See heap_2.c and heap_3.c
      for alternative implementations, and the memory management pages of
      http://www.FreeRTOS.org for more information. */
      ( void ) pv;
}
/*-----------------------------------------------------------*/

void vPortInitialiseBlocks( void )
{
      /* Only required when static memory is not cleared. */
      xNextFreeByte = ( size_t ) 0;
}
/*-----------------------------------------------------------*/

size_t xPortGetFreeHeapSize( void )
{
      return ( configTOTAL_HEAP_SIZE - xNextFreeByte );
}
```

8.3 heap_2.c

```
/*
    FreeRTOS V6.0.0 - Copyright (C) 2009 Real Time Engineers Ltd.

    ***************************************************************************
    *                                                                         *
    * If you are:                                                             *
    *                                                                         *
    *    + New to FreeRTOS,                                                   *
    *    + Wanting to learn FreeRTOS or multitasking in general quickly       *
    *    + Looking for basic training,                                        *
    *    + Wanting to improve your FreeRTOS skills and productivity           *
```

```
    *                                                                       *
    * then take a look at the FreeRTOS eBook                                *
    *                                                                       *
    *        "Using the FreeRTOS Real Time Kernel - a Practical Guide"      *
    *                  http://www.FreeRTOS.org/Documentation                *
    *                                                                       *
    * A pdf reference manual is also available.  Both are usually delivered *
    * to your inbox within 20 minutes to two hours when purchased between 8am *
    * and 8pm GMT (although please allow up to 24 hours in case of          *
    * exceptional circumstances).  Thank you for your support!              *
    *                                                                       *
    *************************************************************************

    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    ***NOTE*** The exception to the GPL is included to allow you to distribute
    a combined work that includes FreeRTOS without being obliged to provide the
    source code for proprietary components outside of the FreeRTOS kernel.
    FreeRTOS is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    http://www.FreeRTOS.org - Documentation, latest information, license and
    contact details.

    http://www.SafeRTOS.com - A version that is certified for use in safety
    critical systems.

    http://www.OpenRTOS.com - Commercial support, development, porting,
    licensing and training services.
*/

/*
 * A sample implementation of pvPortMalloc() and vPortFree() that permits
 * allocated blocks to be freed, but does not combine adjacent free blocks
 * into a single larger block.
 *
 * See heap_1.c and heap_3.c for alternative implementations, and the memory
 * management pages of http://www.FreeRTOS.org for more information.
 */
#include <stdlib.h>

/* Defining MPU_WRAPPERS_INCLUDED_FROM_API_FILE prevents task.h from redefining
all the API functions to use the MPU wrappers.  That should only be done when
task.h is included from an application file. */
#define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

#include "FreeRTOS.h"
```

```
#include "task.h"

#undef MPU_WRAPPERS_INCLUDED_FROM_API_FILE

/* Allocate the memory for the heap.  The struct is used to force byte
alignment without using any non-portable code. */
static union xRTOS_HEAP
{
      #if portBYTE_ALIGNMENT == 8
            volatile portDOUBLE dDummy;
      #else
            volatile unsigned long ulDummy;
      #endif
      unsigned char ucHeap[ configTOTAL_HEAP_SIZE ];
} xHeap;

/* Define the linked list structure.  This is used to link free blocks in order
of their size. */
typedef struct A_BLOCK_LINK
{
      struct A_BLOCK_LINK *pxNextFreeBlock;    /*<< The next free block in the
list. */
      size_t xBlockSize;                                    /*<< The size of the
free block. */
} xBlockLink;


static const unsigned short  heapSTRUCT_SIZE   = ( sizeof( xBlockLink ) +
portBYTE_ALIGNMENT - ( sizeof( xBlockLink ) % portBYTE_ALIGNMENT ) );
#define heapMINIMUM_BLOCK_SIZE     ( ( size_t ) ( heapSTRUCT_SIZE * 2 ) )

/* Create a couple of list links to mark the start and end of the list. */
static xBlockLink xStart, xEnd;

/* Keeps track of the number of free bytes remaining, but says nothing about
fragmentation. */
static size_t xFreeBytesRemaining;

/* STATIC FUNCTIONS ARE DEFINED AS MACROS TO MINIMIZE THE FUNCTION CALL DEPTH.
*/

/*
 * Insert a block into the list of free blocks - which is ordered by size of
 * the block.  Small blocks at the start of the list and large blocks at the end
 * of the list.
 */
#define prvInsertBlockIntoFreeList( pxBlockToInsert )
                  \
{
                                                \
xBlockLink *pxIterator;
                                    \
size_t xBlockSize;
                                          \

                                                \
      xBlockSize = pxBlockToInsert->xBlockSize;
                        \
```

```
                                                    \
      /* Iterate through the list until a block is found that has a larger size
*/    \
      /* than the block we are inserting. */
                                \
      for( pxIterator = &xStart; pxIterator->pxNextFreeBlock->xBlockSize <
xBlockSize; pxIterator = pxIterator->pxNextFreeBlock )     \
      {
                                                    \
            /* There is nothing to do here - just iterate to the correct
position. */     \
      }
                                                    \

                                                    \
      /* Update the list to include the block being inserted in the correct */
          \
      /* position. */
                                              \
      pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;
                  \
      pxIterator->pxNextFreeBlock = pxBlockToInsert;
                        \
}
/*-----------------------------------------------------------*/

#define prvHeapInit()
                                      \
{
                                                    \
xBlockLink *pxFirstFreeBlock;
                                \

                                                    \
      /* xStart is used to hold a pointer to the first item in the list of free
*/    \
      /* blocks.  The void cast is used to prevent compiler warnings. */
                  \
      xStart.pxNextFreeBlock = ( void * ) xHeap.ucHeap;
                        \
      xStart.xBlockSize = ( size_t ) 0;
                                \

                                                    \
      /* xEnd is used to mark the end of the list of free blocks. */
                  \
      xEnd.xBlockSize = configTOTAL_HEAP_SIZE;
                        \
      xEnd.pxNextFreeBlock = NULL;
                                \

                                                    \
      /* To start with there is a single free block that is sized to take up the
          \
      entire heap space. */
                                      \
      pxFirstFreeBlock = ( void * ) xHeap.ucHeap;
```

```
                                \
      pxFirstFreeBlock->xBlockSize = configTOTAL_HEAP_SIZE;
                  \
      pxFirstFreeBlock->pxNextFreeBlock = &xEnd;
                                \

                                                \
      xFreeBytesRemaining = configTOTAL_HEAP_SIZE;
                           \
}
/*-----------------------------------------------------------*/

void *pvPortMalloc( size_t xWantedSize )
{
xBlockLink *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
static portBASE_TYPE xHeapHasBeenInitialised = pdFALSE;
void *pvReturn = NULL;

      vTaskSuspendAll();
      {
            /* If this is the first call to malloc then the heap will require
            initialisation to setup the list of free blocks. */
            if( xHeapHasBeenInitialised == pdFALSE )
            {
                  prvHeapInit();
                  xHeapHasBeenInitialised = pdTRUE;
            }

            /* The wanted size is increased so it can contain a xBlockLink
            structure in addition to the requested amount of bytes. */
            if( xWantedSize > 0 )
            {
                  xWantedSize += heapSTRUCT_SIZE;

                  /* Ensure that blocks are always aligned to the required
number of bytes. */
                  if( xWantedSize & portBYTE_ALIGNMENT_MASK )
                  {
                        /* Byte alignment required. */
                        xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
                  }
            }

            if( ( xWantedSize > 0 ) && ( xWantedSize < configTOTAL_HEAP_SIZE ) )
            {
                  /* Blocks are stored in byte order - traverse the list from
the start
                  (smallest) block until one of adequate size is found. */
                  pxPreviousBlock = &xStart;
                  pxBlock = xStart.pxNextFreeBlock;
                  while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock-
>pxNextFreeBlock ) )
                  {
                        pxPreviousBlock = pxBlock;
                        pxBlock = pxBlock->pxNextFreeBlock;
                  }
```

```
                  /* If we found the end marker then a block of adequate size
was not found. */
                  if( pxBlock != &xEnd )
                  {
                        /* Return the memory space - jumping over the xBlockLink
structure
                        at its start. */
                        pvReturn = ( void * ) ( ( ( unsigned char * )
pxPreviousBlock->pxNextFreeBlock ) + heapSTRUCT_SIZE );

                        /* This block is being returned for use so must be taken
our of the
                        list of free blocks. */
                        pxPreviousBlock->pxNextFreeBlock = pxBlock-
>pxNextFreeBlock;

                        /* If the block is larger than required it can be split
into two. */
                        if( ( pxBlock->xBlockSize - xWantedSize ) >
heapMINIMUM_BLOCK_SIZE )
                        {
                              /* This block is to be split into two.  Create a
new block
                              following the number of bytes requested. The void
cast is
                              used to prevent byte alignment warnings from the
compiler. */
                              pxNewBlockLink = ( void * ) ( ( ( unsigned char
* ) pxBlock ) + xWantedSize );

                              /* Calculate the sizes of two blocks split from
the single
                              block. */
                              pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
xWantedSize;
                              pxBlock->xBlockSize = xWantedSize;

                              /* Insert the new block into the list of free
blocks. */
                              prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
                        }

                        xFreeBytesRemaining -= xWantedSize;
                  }
            }
      }
      xTaskResumeAll();

      #if( configUSE_MALLOC_FAILED_HOOK == 1 )
      {
            if( pvReturn == NULL )
            {
                  extern void vApplicationMallocFailedHook( void );
                  vApplicationMallocFailedHook();
            }
      }
      #endif
```

```
	return pvReturn;
}
/*-----------------------------------------------------------*/

void vPortFree( void *pv )
{
unsigned char *puc = ( unsigned char * ) pv;
xBlockLink *pxLink;

	if( pv )
	{
		/* The memory being freed will have an xBlockLink structure
immediately
		before it. */
		puc -= heapSTRUCT_SIZE;

		/* This casting is to keep the compiler from issuing warnings. */
		pxLink = ( void * ) puc;

		vTaskSuspendAll();
		{
			/* Add this block to the list of free blocks. */
			prvInsertBlockIntoFreeList( ( ( xBlockLink * ) pxLink ) );
			xFreeBytesRemaining += pxLink->xBlockSize;
		}
		xTaskResumeAll();
	}
}
/*-----------------------------------------------------------*/

size_t xPortGetFreeHeapSize( void )
{
	return xFreeBytesRemaining;
}
/*-----------------------------------------------------------*/

void vPortInitialiseBlocks( void )
{
	/* This just exists to keep the linker quiet. */
}
```

8.4 heap_3.c

```
/*
    FreeRTOS V6.0.0 - Copyright (C) 2009 Real Time Engineers Ltd.

    ***************************************************************************
    *                                                                         *
    * If you are:                                                             *
    *                                                                         *
    *    + New to FreeRTOS,                                                   *
    *    + Wanting to learn FreeRTOS or multitasking in general quickly       *
    *    + Looking for basic training,                                        *
    *    + Wanting to improve your FreeRTOS skills and productivity           *
    *                                                                         *
    * then take a look at the FreeRTOS eBook                                  *
    *                                                                         *
    *        "Using the FreeRTOS Real Time Kernel - a Practical Guide"        *
```

```
    *                  http://www.FreeRTOS.org/Documentation                *
    *                                                                       *
    * A pdf reference manual is also available.  Both are usually delivered *
    * to your inbox within 20 minutes to two hours when purchased between 8am *
    * and 8pm GMT (although please allow up to 24 hours in case of          *
    * exceptional circumstances).  Thank you for your support!              *
    *                                                                       *
    *************************************************************************

    This file is part of the FreeRTOS distribution.

    FreeRTOS is free software; you can redistribute it and/or modify it under
    the terms of the GNU General Public License (version 2) as published by the
    Free Software Foundation AND MODIFIED BY the FreeRTOS exception.
    ***NOTE*** The exception to the GPL is included to allow you to distribute
    a combined work that includes FreeRTOS without being obliged to provide the
    source code for proprietary components outside of the FreeRTOS kernel.
    FreeRTOS is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
    more details. You should have received a copy of the GNU General Public
    License and the FreeRTOS license exception along with FreeRTOS; if not it
    can be viewed here: http://www.freertos.org/a00114.html and also obtained
    by writing to Richard Barry, contact details for whom are available on the
    FreeRTOS WEB site.

    1 tab == 4 spaces!

    http://www.FreeRTOS.org - Documentation, latest information, license and
    contact details.

    http://www.SafeRTOS.com - A version that is certified for use in safety
    critical systems.

    http://www.OpenRTOS.com - Commercial support, development, porting,
    licensing and training services.
*/


/*
 * Implementation of pvPortMalloc() and vPortFree() that relies on the
 * compilers own malloc() and free() implementations.
 *
 * This file can only be used if the linker is configured to to generate
 * a heap memory area.
 *
 * See heap_2.c and heap_1.c for alternative implementations, and the memory
 * management pages of http://www.FreeRTOS.org for more information.
 */

#include <stdlib.h>

/* Defining MPU_WRAPPERS_INCLUDED_FROM_API_FILE prevents task.h from redefining
all the API functions to use the MPU wrappers.  That should only be done when
task.h is included from an application file. */
#define MPU_WRAPPERS_INCLUDED_FROM_API_FILE

#include "FreeRTOS.h"
```

```
#include "task.h"

#undef MPU_WRAPPERS_INCLUDED_FROM_API_FILE

/*-----------------------------------------------------------*/

void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn;

	vTaskSuspendAll();
	{
		pvReturn = malloc( xWantedSize );
	}
	xTaskResumeAll();

	#if( configUSE_MALLOC_FAILED_HOOK == 1 )
	{
		if( pvReturn == NULL )
		{
			extern void vApplicationMallocFailedHook( void );
			vApplicationMallocFailedHook();
		}
	}
	#endif

	return pvReturn;
}
/*-----------------------------------------------------------*/

void vPortFree( void *pv )
{
	if( pv )
	{
		vTaskSuspendAll();
		{
			free( pv );
		}
		xTaskResumeAll();
	}
}
```