COMPSYS 723 Embedded Systems Design 2017

FreeRTOS Notes

Zoran Salcic

1. Introduction

FreeRTOS is a lightweight, embeddable, multi-tasking, Real-Time Operating System (RTOS). It makes the key assumption that the target system has a single processing unit. It is really a library of types and functions that can be used to build microkernels using a combination of C and assembly language, and has been ported to most embedded systems architectures. It allows a very small kernel to be produced for target microcontrollers, somewhere between 4–9kB. It provides services for embedded programming tasks, communication and synchronisation, memory management, real-time events, and I/O-device control.

Fourteen different compilers are used with FreeRTOS, giving complex configuration options and extensive parametrisation. A version of the software, SafeRTOS, has been certified to Safety Integrity Level 3 by the Technical University of Vienna for the following safety standards: IEC 61508, FDA 510(k), and DO-178B.These certificates are for the process of development, rather than for the correctness of the software against stated requirements.

2. Overview of FreeRTOS

The key elements of FreeRTOS are:

- **Tasks**: user processes.
- **Queues**: communication mechanisms between tasks and interrupts.
- **Semaphores** and **Mutexes**: used for resource management, event counting, mutual exclusion locks, etc.

They are implemented as a set of functions written in C. It provides the following functionality to the application programmer.

1. Implement fixed-priority, preemptive scheduling.
2. Trap software interrupts:
   a) Find the highest priority ready task to run.
   b) Save the context of the
   c) Restore the context of the new task.
3. Trap timer interrupts:
   a) Update the tickcount.
   b) Check delayed tasks, and move to ready if required,
   c) Do context switching if required.
4. Provide API functions for:
   a) Creation and management of multiple tasks.
   b) Inter-task communication through queues, semaphores, and mutexes.
   c) Heap memory management through malloc and free.

3. Tasks 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())
- delay/resume ((vTaskDelay(), vTaskDelayUntil(), vTaskSuspend(), vTaskResume(), vTaskResumeFromISR()).

Additional options are available to a programmer, for instance to create a critical sequence or monitor the task for debugging purpose.

3.1. Task states

Tasks in FreeRTOS can be regarded as occupying one of two top-level states, running or notRunning. The running task is recorded by the task control block handler pxCurrentTCB, and simply indicates that the task is currently executing on the processor. The notRunning state can be decomposed into three substates: ready, suspended, and blocked; tasks in the ready state are available for scheduling to the running state. Tasks can also be blocked by an event for a certain period; these are the blocked tasks. Suspended tasks wait until they are resumed by another task. Tasks transit between these states as described in Figure 1. For instance, a task cannot transit from suspended to running, because only ready tasks can be scheduled as running.

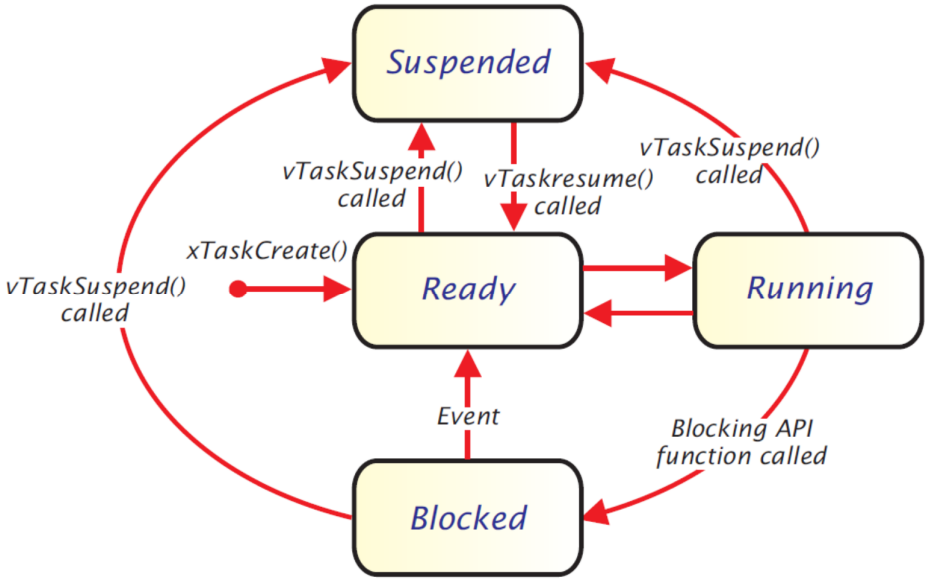

Figure 1 Task states and state transitions

There are several reasons for a task to be in nonRunning state. A task can be preempted because of a higher priority task, because it has been delayed or because it waits for an event. When a task can run but is waiting for the processor to be available, it is Ready state. This can happen when a task has everything it needs to run but there is a higher 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 Suspend state.

It is important to underline that a task can leave by itself the Running state (delay, suspend or wait for an event), but only the scheduler can switch in this task to the Running state and only from the Ready state.

In FreeRTOS, the scheduler takes responsibility for counting clock ticks, used to express time, and schedules tasks. The scheduling policy adopted is priority-based scheduling, which means that the task with the highest priority and in the ready state can be executed. As a result, it is impossible to use FreeRTOS in hard real-time environments. When a ready task has a higher priority than the running task, it will displace the running task from the CPU. The scheduler has two ways for switching tasks: preemptive and cooperative scheduling. In preemptive mode, the task with the highest priority will block the running task immediately and take the CPU. In cooperative mode, the running task can finish its CPU time before the task with the highest priority takes over. API functions are provided for task creation, deletion, and control. It is worth noting that the deletion API does not actually delete a task from the system: it only marks the task and removes its reference from related queues. The idle task, with permanent priority 0, the lowest priority, is used to actually do the deleting job and release the memory allocated by the kernel; however, it does not collect the memory allocated by user, so tasks have to release used memory by themselves before deleting.

3.2. Creating and Deleting Tasks

A task is defined by a simple C function, taking one void* argument and returning nothing (e.g. below).

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

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

An example of a typical task creation is shown below:

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

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 width 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.
- 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 left NULL.

4. Task Scheduling

Task scheduling aims to decide which task in Ready state has to be run at a given time. FreeRTOS achieves this by introducing priorities given to the tasks when they are created. Priority of a task is the only element the scheduler takes into account when deciding which task has to be run next. Every clock tick makes the scheduler to decide which task has to be waken up, as shown in Figure 2.

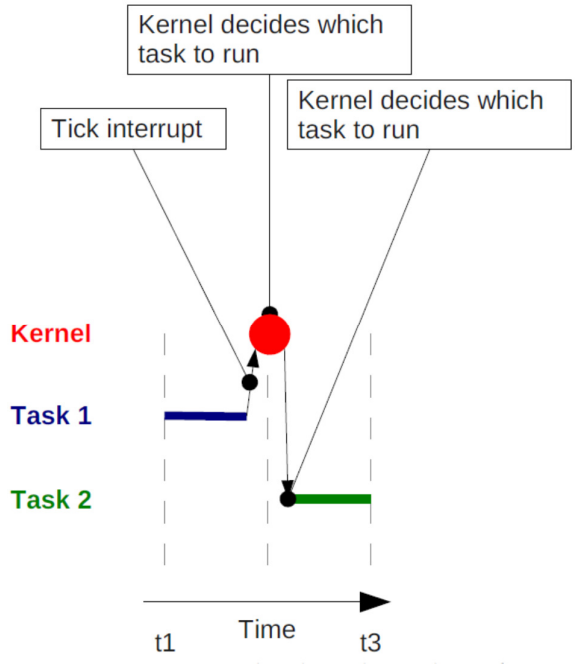

Figure 2 Scheduler (Kernel) deciding which task to run next

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(). 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, and hardware limitation (width of the MAX_PRIORITIES type). If a higher value is given to a task, then FreeRTOS cuts it to MAX_PRIORITIES

– 1. Figure 3 gives an example of an application run in FreeRTOS. Task 1 and Task 3 are event based tasks (they start when an event occurs, run then wait for the event to occur again), Task 2 is a periodic and idle task that 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 a higher priority whereas low frequencies tasks deserve a low priority. Event based or continuous tasks are pre-empted by periodic tasks.

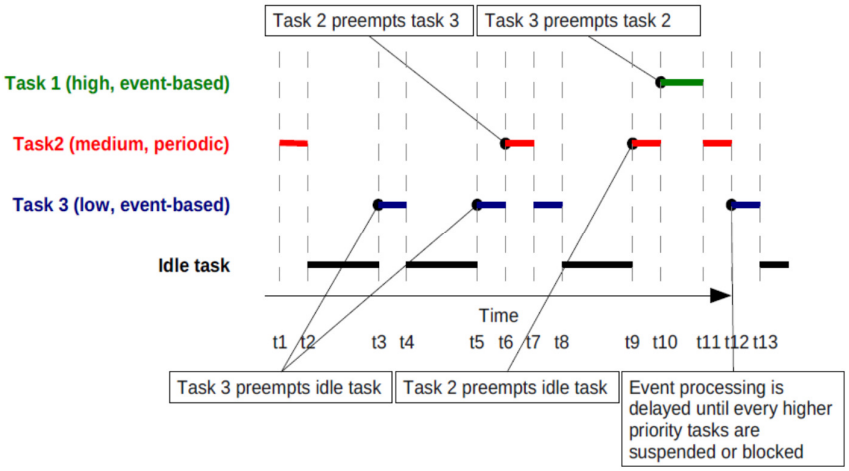

Figure 3 Execution and schedule for a FreeRTOS application

Tasks created with an equal priority are treated fairly 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, as illustrated by an example in Figure 4. Effectively, this implements a Round Robin schedule where quantum is the time between two consecutive clock ticks. This value is available in TICK_RATE_HZ constant, in FreeRTOSConfig.h.

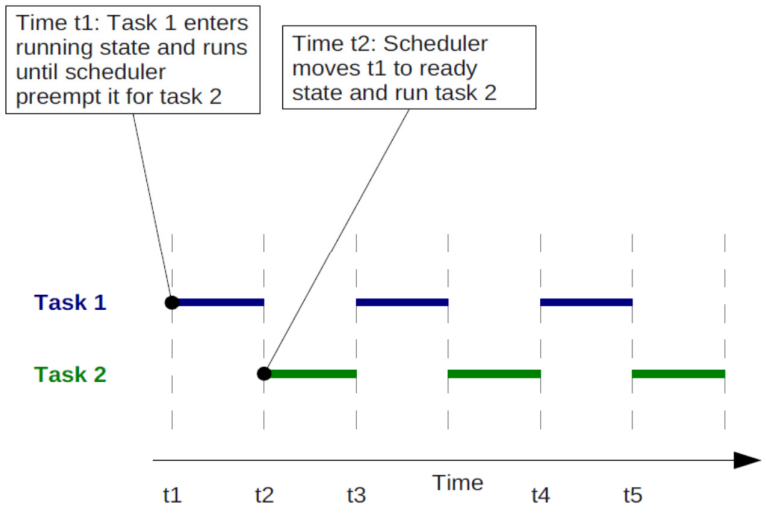

Figure 4 Running two tasks with the equal priority

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.

5. Queues

Queues are an underlying mechanism used for all tasks communication or synchronization in a FreeRTOS environment. They are an important concept to understand as it is unavoidable to be able to build a complex application with tasks cooperating with each other. Queues are a means to store a finite number (named 'length') of fixedsize data. They can be read and written by different tasks, and don't belong to any task in particular. A queue is normally a FIFO which means elements are read from a queue in the order they have been written to the queue. The queue behaviour depends on the writing method: two writing functions can be used to write either at the beginning or at the end of the queue.

Before use a queue has to be created. Length of a queue and its width (the size of its elements) are given when the queue is created. The function prototype available to create a queue is given below:

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

where uxQueueLength represents the number of elements this queue will be able to handle at any given time and uxItemSize is the size in bytes 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.

When a single task reads from 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.

Queues can be read using two methods:

1. With removing element that was read from the queue. An example is given below:

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

   where 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 successfully read before xTicksToWait is reached, and if not, errQUEUE_EMPTY is returned from xQueueReceive().

2. Without removing element from the queue (the element read remains in the queue to be used/read by other tasks)

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

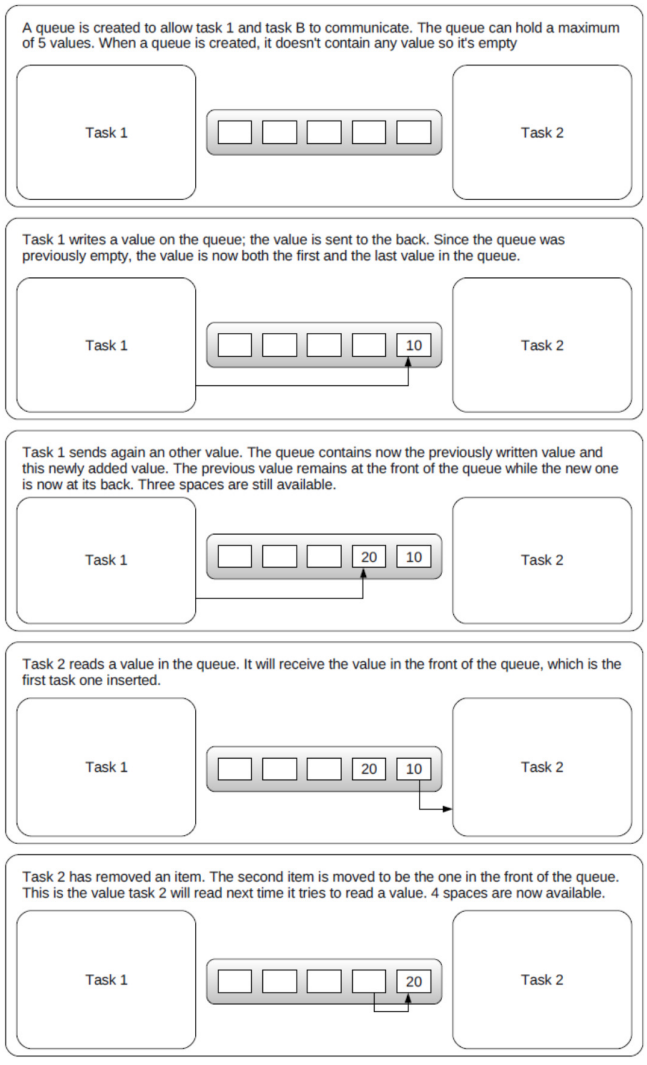

Figure 5 Illustration of using queue and its operations

Writing to a queue complies with the same rules as reading from it. When a task tries to write to a queue, it has to wait for it to have sufficient free space: the task is blocked until another task reads the queue and free some space. If several tasks attempt to write to the same queue, the higher priority task is chosen first. If several tasks with the same priority are trying to write to a queue, then the first one to wait is chosen. Figure 5 gives an illustration on how queues work.

A prototype below describes the normal method to write to a queue in FIFO mode:

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

where xQueue is the queue to write to. This value is returned by the queue creation method; pvItemToQueue is a pointer to an element which is to be copied (by value) to the queue; xticksToWait is the number of ticks to wait before the task gives up to write to this queue (If xTicksToWait is 0, the task won't wait at all if the queue is full, and if INCLUDE_vTaskSuspend if defined to 1 inf FreeRTOSConfig.h); 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.

Examples of a prototype function to be used if the user wants the last written element to be read first (Last In, First Out or LIFO) is given below:

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

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

While xQueueSendToBack is a synonym for xQueueSend (FIFO operation), xQueueSendToFront is used to write to the beginning of the queue, which then allows to read most recently stored item (Last In First Out, LIFO).

6. Resource Management

6.1. Binary semaphores

Binary semaphores are the simplest effective way to synchronize tasks. A binary semaphore can be seen as a queue which contains only one element. Figure 6 gives an idea on its mechanism. In order to be used, a semaphore has to be created first:

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

where xSemaphore is the semaphore name. Semaphore allows two operations, typically called P() and V(), which correspond to acquiring (taking) and releasing (giving back) semaphore, respectively.

A task taking the semaphore must wait until the semaphore is available. The task is blocked until semaphore is available or until a delay has elapsed (if applicable):

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

where xSemaphore is the semaphore to take (acquire). 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, and if not, pdFALSE is returned.

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

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

where xSemaphore is the semaphore to be given (released). 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.

6.2. Mutexes

Mutexes are designed to prevent mutual exclusion or deadlocking. A mutex is used similarly to a binary semaphore, except the task which takes the semaphore must give it back. This can be achieved 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. An illustration of using mutexes is shown in Figure 7.

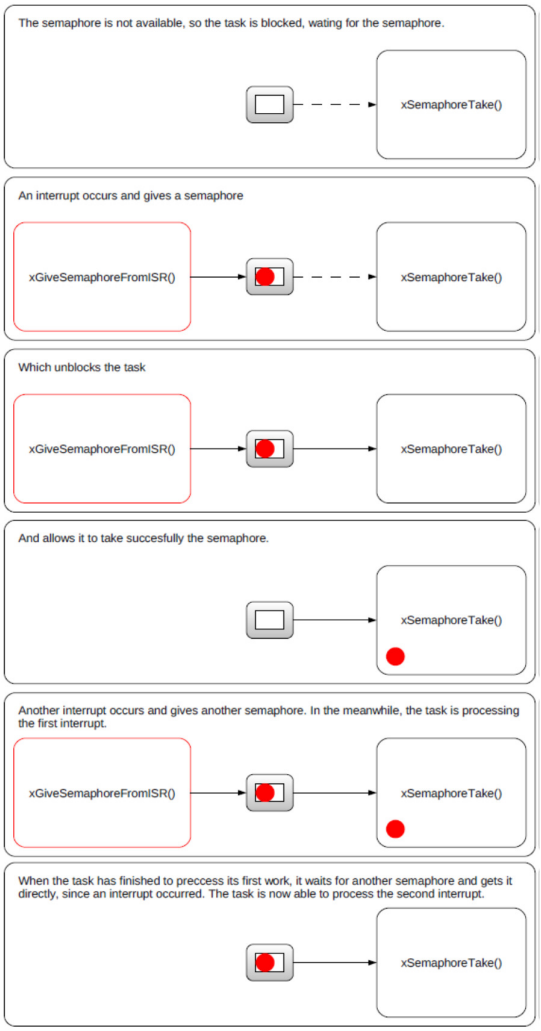

Figure 6 Using semaphores

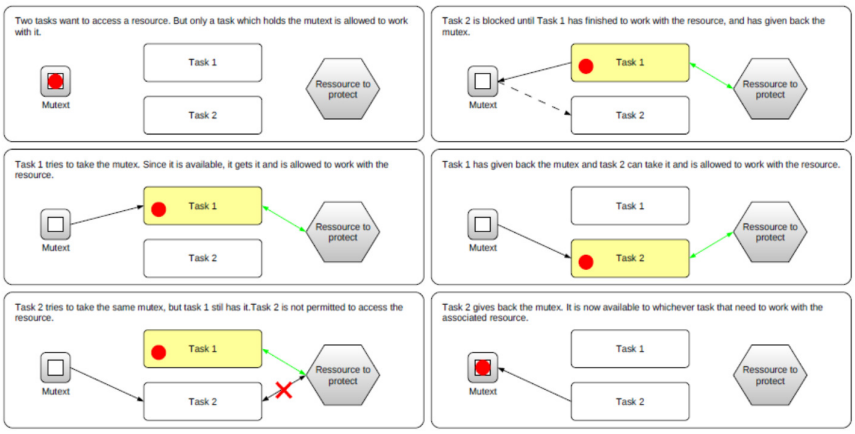

Figure 7 Typical use of mutexes

A problem that can emerge when using mutexes is known as priority inversion, when a task of lower priority prevents scheduling and execution of a task of higher priority. Priority inheritance is actually the only difference between a binary semaphore and a mutex. When several tasks try to take 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.

6.3. Counting semaphores

Counting semaphores are similar to simple semaphores except that they are initialised to a number of instances that can be taken by different tasks. A task will be able to take semaphore without waiting (blocking) as long as there are free instances of the semaphore.

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

where uxMaxCount is the capacity of the counting semaphore, its maximum number of instances to be taken; and 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. Take and give operations on counting semaphores are realized using the same function as for binary semaphores.

7. 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 a hardware instruction. We will suppose we are using a micro controller that handles limited number of different levels of interrupts, which are given different priority. It should be noticed that the interrupt priorities are considered more important than task priorities, i.e. the tasks will always pre-empted by interrupts (see Figure 8).

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 use 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 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.

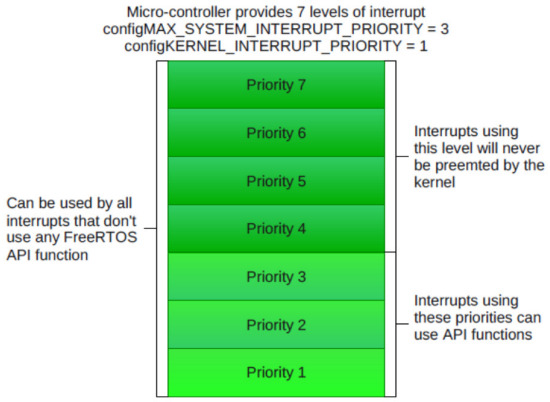

Figure 8 Interrupt organisation in FreeRTOS for a microcomputer with 7 interrupt levels

7.1. Managing interrupts using binary semaphores

Interrupt handlers are pieces of code run by the microcontroller 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 for interrupt 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. One 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 9 proposes a solution to reduce significantly the time an ISR can run. An ISR ‘gives’ a semaphore and unblocks a ‘Handler’ task that is able to handle the ISR, making the ISR execution much shorter.

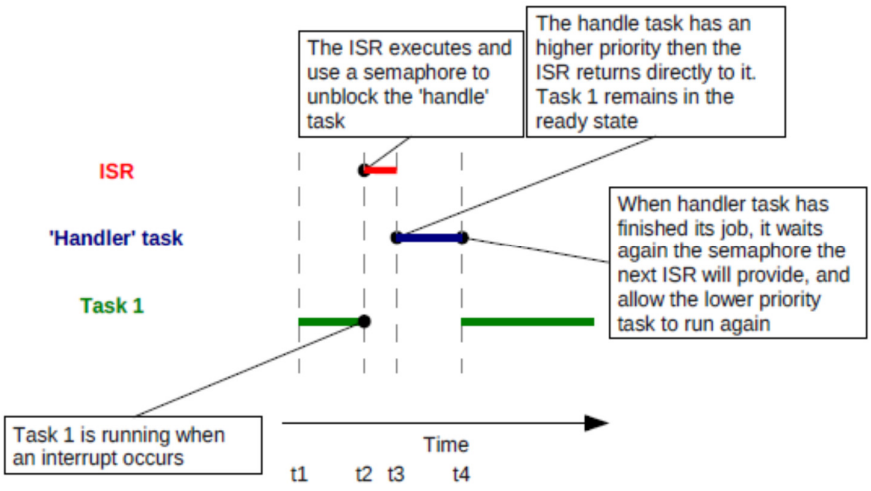

Figure 9 Deferred interrupt processing by using 'Handler' tasks

7.2. 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 protect from any context change, either from a scheduler operation, or an interrupt event, others only prevent scheduler from preempting the task.

Handling this can be very important as many instructions may look atomic but require several hardware instructions (e.g., load variable address to a register file, load a value to another register and move the value to the matching memory address using the two registers).

Disabling interrupts

This form or critical section is very efficient but must be kept as short as possible since it brings the whole system in a state when 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 interrupt mechanism. FreeRTOS provides two mechanisms for entering a critical section of code and disabling interrupts and exiting the critical section and enabling interrupts:

```
taskENTER_CRITICAL();

/* critical section code

taskEXIT_CRITICAL();
```

A task can start a critical section with taskENTER_CRITICAL() and stop it using taskEXIT_CRITICAL(). The system allows a critical section to be started while another 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 many times as taskSTART_CRITICAL was called.

Generally 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 with priority 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.

A less drastic method to create a critical section consists in preventing any task from being preempted, but let interrupts to do their job. This goal can be achieve by preventing any task to leave the Ready state to Running state, which essentially mean as stopping the scheduler, or stopping all the tasks. An example of code is given below:

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

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

8. 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 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 own memory management method, or to use one of the FreeRTOS standard methods which can be found in heap_1.c, heap_2.c, heap_3.c or heap_4.c).

8.1. Prototypes

All implementations respect the same allocation/deallocation memory function prototypes. These prototypes are available in two functions:

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

xWanted size is the size, in bytes, to be allocated, pv is a pointer to the memory to be freed. pvPortMalloc returns a pointer to the memory allocated, so it is straightforward to free that memory when not needed.

8.2. Fixed/permanent memory allocation

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.

Since this scheme is made under assumption that 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 10 gives an illustration on 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.

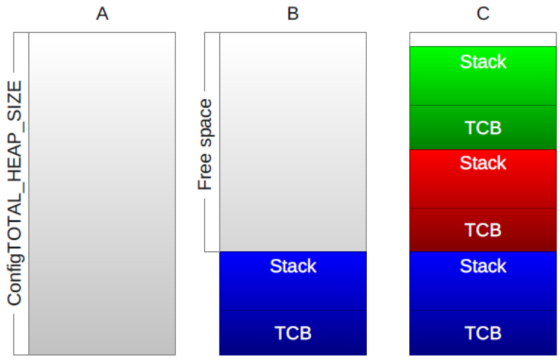

Figure 10 (a) No memory allocated yet, (b) memory for one task allocated and (c) memory for three tasks allocated

8.3. Constant sized and numbered memory allocations

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

As in the previous strategy, FreeRTOS uses a large initial array, whose size depends on configTOTAL_HEAP_SIZE and makes the application to appears to consume huge RAM. However, compared with the previous case, there is a different 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 (see Figure 11).

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.

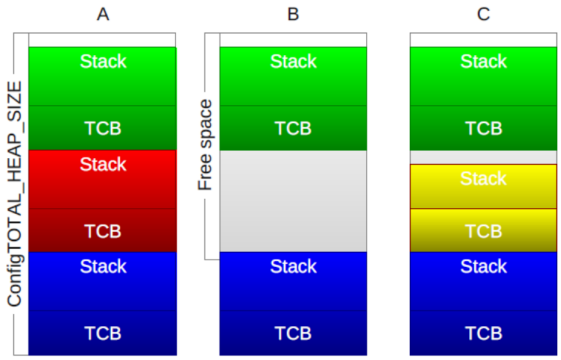

Figure 11 Algorithm that always allocates the smallest memory segment where the requested partition fits

8.4. Free memory allocation and deallocation

This 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 makes them thread safe by suspending the scheduler while allocating or deallocating. The example below demonstrates an example of 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();
        }
}
```

9. A Simple Example of FreeRTOS Application

We use a simple example application to illustrate the functionality provided by FreeRTOS. Figure 12 shows the C code of an application that uses the FreeRTOS. Initially, the application creates two tasks: Task1 and Task2, with priority 1 and 2 respectively (a higher number indicates higher priority), and then starts the FreeRTOS scheduler. The scheduler then runs Task2, which immediately increases the priority of Task1 to 3. Task2 is now preempted by Task1, which gets to execute and creates a new task—Task3 with priority 4, which is the highest at the moment. Therefore, it preempts Task1 and gets

to execute. Once Task3 is executing, it deletes itself, which triggers the scheduler to reschedule the system. As Task1 has the highest priority at this moment, it gets to execute again and forever.

We now describe in more detail what happens in the FreeRTOS implementation code. The application code for main, tx1, tx2 and tx3 is compiled along with the FreeRTOS code (for the scheduler, and the API calls including xTaskCreate), and loaded into memory. The scheduler code is loaded into the Interrupt Service Routine (ISR) code area so that it services software interrupts.

By analysing the source code of FreeRTOS, we see that execution begins with the first instruction in main, which is the call to the xTaskCreate API function. This code is provided by FreeRTOS, allocates 1 kilobyte of memory on the heap for the task stack, as well as space to store its Task Control Block (TCB) [Bar12a, Bar12b]. From the source code, we can find that the TCB contains all vital information about the task: where its code (tx1 in this case) is located, where its stack begins, where its current top-of-stack pointer is, what its priority is, and so on. The API call initialises the TCB entries for Task1. It then creates and initialises the various lists that the OS maintains, such as pxReadyTasksLists for recording tasks in ready state, xSuspendedTaskList for representing tasks in suspended state and so on. It finally adds Task1 to the ready list and returns. Next, main calls xTaskCreate for Task2 and the API call sets up the stack and TCB for Task2 and adds it to the ready list, in a similar way. The next instruction in main is a call to the vTaskStartSchedular API, which is also provided by FreeRTOS. This call creates the idle task with priority 0, and adds it to the ready list. It also sets the timer tick interrupt to occur at the required frequency.

Finally, it does a context-switch to the highest priority ready task (i.e., it restores its execution state, namely the contents of its registers, from the task's stack where they were stored). The processor will next execute the instruction in the task that is resumed. In our example, this means that Task2 will now begin execution.
When Task2 begins execution, it makes an API call to vTaskPrioritySet. The code for this API call compares the new priority and the current priority to decide whether scheduling is needed. If the API increases the priority of a task or decreases the priority of the current running task, a reschedule will be requested. It then assigns the new priority to the target task, and moves the task to the proper position in the ready list, if it is a ready task. In our case, the priority of Task1 is changed to 3, it is moved to its proper position in pxReadyTasksLists. The API code then does a yield (a software interrupt) that is trapped by the scheduler. The scheduler picks the longest waiting, highest priority ready task, which in this case is Task1, and makes it the running task. Before this, the scheduler saves the registers of Task2 on its stack, and restores the register context of Task1 from its stack.

Task1 now creates the new task Task3. The process is similar to the xTaskCreate call to create Task1 and Task2. The difference is that this time xTaskCreate triggers scheduling to make Task3 running. When Task3 begins execution it makes a call to the vTaskDelete API call. The code for this API is simple. It removes the target task from state list and related events list, in this case, Task3 is removed from pxReadyTasksLists. As it is the current running task, the API code triggers scheduling again to makes

the highest priority ready task running, which is Task1. Task1 then executes its trivial for-loop, ad infinitum.

```
xTaskHandle txh1;

void tx1(void * xPara){
   xTaskCreate(tx3, (signed char *) "Task 3", 1000, NULL, 4, NULL);
   for( ;; );
}

void tx2(void * xPara){
   for( ;; ){
      vTaskPrioritySet(txh1, 3);
   }
}

void tx3(void * xPara){
   for( ;; ){
      vTaskDelete(NULL);
   }
}

int main(void){
   xTaskCreate(tx1, (signed char *) "Task 1", 1000, NULL, 1, & txh1);
   xTaskCreate(tx2, (signed char *) "Task 2", 1000, NULL, 2, NULL);
   vTaskStartScheduler();
   return 0;
}
```

Figure 12. An example application that uses FreeRTOS

10. Data Types and Coding Conventions

FreeRTOS has two special data types associated with each port:

portTickType used to store tick count value and to specify block times (depending on the processor can unsigned 16-bit or 32-bit type)

portBASE_TYPE selected as the most efficient data type for the underlying processor.

Variable names are prefixed with their type: ‘c’ for char, ‘s’ for short, ‘l’ for long, and ‘x’ for portBASE_TYPE and any other type (structures, handles, que handles, etc.).

If a variable is unsigned it is prefixed with a ‘u’, if it is a pointer it is also prefixed with ‘p’. For example, a variable of type pointer to char is prefixed with ‘pc’.

Functions are prefixed with both type they return and the file they are defined within:

vTaskPrioritySet() returns void and is defined within task.c

xQueueReceive() returns a variable of portBASE_TYPE and is defined within queue.c.

File scope (private) functions are prefixed with ‘prv’.

11. Best source of information:

You can find free software, very detailed explanations and user references on:

http://www.freertos.org/