8

Interprocess Communication Primitives in FreeRTOS

CONTENTS

The previous chapter discussed the interprocess communication primitives available to the user of a "full-fledged" operating system that supports the POSIX standard at the Application Programming Interface (API) level. Since those operating systems were initially intended for general-purpose computers, one of their main goals is to put at the user's disposal a rich set of high-level primitives, meant to be powerful and convenient to use.

At the same time, they must also deal with multiple applications, maybe pertaining to different users, being executed at the same time. Protecting those applications from each other and preventing, for example, an error in one application from corrupting the memory space of another, is a daunting task that requires the adoption of complex process and memory models, and must be suitably supported by sophisticated hardware components, such as a Memory Management Unit (MMU).

Unfortunately, all these features come at a cost, in terms of operating system code size, memory requirements, execution overhead, and complexity of the underlying hardware. When designing a small-scale embedded system, it may be impossible to afford such a cost, and therefore, the developer is compelled to settle on a cheaper architecture in which both the operating system and its interface are much simpler. Besides the obvious disadvantages, making the operating system simpler and smaller has several advantages as well. For instance, it becomes easier to ensure that the operating system behavior is correct for what concerns real-time execution. A smaller code base usually leads to a more efficient and reliable system, too.

Up to a certain extent, it is possible to reach this goal within the POSIX framework: the IEEE Standard 1003.13 [41], also recognized as an ANSI standard, defines several POSIX subsets, or profiles, oriented to real-time and

TABLE 8.1
Summary of the task-related primitives of FreeRTOS

Function	Purpose	Optional
`vTaskStartScheduler`	Start the scheduler	-
`vTaskEndScheduler`	Stop the scheduler	-
`xTaskCreate`	Create a new task	-
`vTaskDelete`	Delete a task given its handle	*
`uxTaskPriorityGet`	Get the priority of a task	*
`vTaskPrioritySet`	Set the priority of a task	*
`vTaskSuspend`	Suspend a specific task	*
`vTaskResume`	Resume a specific task	*
`xTaskResumeFromISR`	Resume a specific task from an ISR	*
`xTaskIsTaskSuspended`	Check whether a task is suspended	*
`vTaskSuspendAll`	Suspend all tasks but the running one	-
`xTaskResumeAll`	Resume all tasks	-
`uxTaskGetNumberOfTasks`	Return current number of tasks	-

embedded application environments. Each profile represents a different trade-off between complexity and features.

However, in some cases, it may still be convenient to resort to an even simpler set of features with a streamlined, custom programming interface. This is the case with FreeRTOS [13], a small open-source operating system focusing on high portability, very limited footprint and, of course, real-time capabilities. It is licensed under a variant of the GNU General Public License (GPL). The most important difference is an exception to the GPL that allows application developers to distribute a combined work that includes FreeRTOS without being obliged to provide the source code for any proprietary components.

This chapter is meant as an overview of the FreeRTOS API, to highlight the differences that are usually found when comparing a small real-time operating system with a POSIX system and to introduce the reader to small-scale, embedded software development. The FreeRTOS manual [14] provides in-depth information on this topic, along with a number of real-world code examples. Chapter 17 will instead give more information about the internal structure of FreeRTOS.

8.1 FreeRTOS Threads and Processes

FreeRTOS supports only a single process, and multiprogramming is achieved by means of multiple threads of execution, all sharing the same address space. They are called *tasks* by FreeRTOS, as it happens in most other real-time operating systems. The main FreeRTOS primitives related to task management

and scheduling are summarized in Table 8.1. To invoke them, it is necessary to include the main FreeRTOS header file, `FreeRTOS.h`, followed by `task.h`.

With the FreeRTOS approach, quite common also in many other small real-time operating systems, the address space used by tasks is also shared with the operating system itself. The operating system is in fact a library of object modules, and the application program is linked against it when the application's executable image is built, exactly as any other library. The application and the operating system modules are therefore bundled together in the resulting executable image.

This approach keeps the operating system as simple as possible and makes any shared memory-based interprocess communication mechanism extremely easy and efficient to use because all tasks can share memory with no effort. On the other hand, tasks cannot be protected from each other with respect to illegal memory accesses, but it should be noted that many microcontrollers intended for embedded application lack any hardware support for this protection anyway. For some processor architectures, FreeRTOS is able to use a Memory Protection Unit (MPU), when available, to implement a limited form of data access protection among tasks.

Usually, the executable image is stored in a nonvolatile memory within the target system and is invoked either directly or through a minimal boot loader when the system is turned on. Therefore, unlike for Linux, the image must also include an appropriate startup code, which takes care of initializing the target hardware and is invoked before calling the `main()` entry point of the application. Another important difference with respect to Linux is that, when `main()` gets executed, the operating system scheduler is *not yet* active and must be explicitly started by means of the following function call:

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

It should be noted that this function reports errors back in an unusual way. When successful, it does not return to the caller. Instead, the execution proceeds with the FreeRTOS tasks that have been created before starting the scheduler, according to their priorities. On the other hand, `vTaskStartScheduler` may return to the caller for two distinct reasons:

1. An error occurred during scheduler initialization, and so it was impossible to start it successfully.
2. The scheduler was successfully started, but one of the tasks executed afterwards stopped the operating system by invoking

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

The two scenarios are clearly very different because, in the first case, the return is immediate and the application tasks are never actually executed, whereas in the second the return is delayed and usually occurs when the application is shut down in an orderly manner, for instance, at the user's request. However,

since `vTaskStartScheduler` has no return value, there is no immediate way to distinguish between them.

If the distinction is important for the application being developed, then the programmer must make the necessary information available on his or her own, for example, by setting a shared flag after a full and successful application startup so that it can be checked by the code that follows `vTaskStartScheduler`.

It is possible to create a new FreeRTOS task either before or after starting the scheduler, by calling the `xTaskCreate` function:

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE pvTaskCode,
    const char * const pcName,
    unsigned short usStackDepth,
    void *pvParameters,
    unsigned portBASE_TYPE uxPriority,
    xTaskHandle *pvCreatedTask);
```

where:

- `pvTaskCode` is a pointer to a function returning `void` and with one `void *` argument. It represents the entry point of the new task, that is, the function that the task will start executing from. This function must be designed to *never* return to the caller because this operation has undefined results in FreeRTOS. It is very important to remember this because, in a POSIX-compliant operating system, a thread is indeed allowed to return from its starting function; when it does so, it is implicitly terminated with no adverse consequences. On the contrary, returning from the start function of a FreeRTOS task may be quite unfortunate because, on most platforms, it leads to the execution of code residing at an unpredictable address.

- `pcName`, a constant string of characters, represents the human-readable name of the task being created. The operating system simply stores this name along with the other task information it keeps track of, without interpretation, but it is useful when inspecting the operating system data structures, for example, during debugging. The maximum length of the name actually stored by the operating system is limited by a configuration parameter; longer names are silently truncated.

- `usStackDepth` indicates how many stack *words* must be reserved for the task stack. The stack word size depends on the underlying hardware architecture and is configured when the operating system is being ported onto it. If necessary, the actual size of a stack word can be calculated by looking at the `portSTACK_TYPE` data type, defined in an architecture-dependent header file that is automatically included by the main FreeRTOS header file.

- `pvParameters` is a `void *` pointer that will be passed to the task entry point upon execution without any interpretation by the operating system. It is most commonly used to point at a shared memory structure that holds the task parameters and, possibly, return values.

- `uxPriority` represents the initial, or baseline, priority of the new task, expressed as a positive integer. The symbolic constant `tskIDLE_PRIORITY`, defined in the operating system's header files, gives the priority of the idle task, that is, the lowest priority in the system, and higher priority values correspond to higher priorities. The total number of priority levels available is set in the operating system configuration depending on the application requirements because the size of several operating system data structures depend on it. The currently configured value is available in the symbolic constant `configMAX_PRIORITIES`. Hence, the legal range of priorities in the system goes from `tskIDLE_PRIORITY` to `tskIDLE_PRIORITY`+`configMAX_PRIORITIES`−`1`, extremes included.

- `pvCreatedTask` points to the task *handle*, which will be filled upon successful completion of this function. The handle must be used to refer to the new task in the future and is taken as a parameter by all operating system functions that operate on, or refer to, a task.

The return value of `xTaskCreate` is a status code. If its value is `pdPASS`, the function was successful in creating the new task, whereas any other value means that an error occurred. For example, the value `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY` denotes that it was impossible to create the new task because not enough memory was available. The FreeRTOS header `projdefs.h`, automatically included by the main FreeRTOS header file, contains the full list of error codes that may be returned by the operating system functions.

After creation, a task can be deleted by means of the function

```
void vTaskDelete(xTaskHandle pxTaskToDelete);
```

Its only argument, `pxTaskToDelete`, is the handle of the task to be deleted. It is possible to delete the currently running task, and the effect in that case is that `vTaskDelete` will never return to the caller.

For technical reasons, the memory dynamically allocated to the task by the operating system (for instance, to store its stack) cannot be freed immediately during the execution of `vTaskDelete` itself; this duty is instead delegated to the idle task. If the application makes use of `vTaskDelete`, it is important to ensure that a portion of the processor time is available to the idle task, as otherwise the system may run out of memory not because there is not enough but because the idle task was unable to free it fast enough before reuse.

As many other operating system functions, the availability of `vTaskDelete` depends on the operating system configuration so that it can be excluded from systems in which it is not used in order to save code and data memory. In

Table 8.1, as well as in all the ensuing ones, those functions are marked as optional.

Another important difference with respect to a POSIX-compliant operating system is that FreeRTOS—like most other small, real-time operating systems—does not provide anything comparable to the POSIX *thread cancellation* mechanism. This mechanism is rather complex and allows POSIX threads to decline or postpone deletion requests, or *cancellation* requests as they are called in POSIX, directed to them. This is useful in ensuring that these requests are honored only when it is safe to do so.

In addition, POSIX threads can also register a set of functions, called *cleanup handlers*, which will be invoked automatically by the system while a cancellation request is being honored, before the target thread is actually deleted. Cleanup handlers, as their name says, provide therefore a good opportunity for POSIX threads to execute any last-second cleanup action they may need to make sure that they leave the application in a safe and consistent state upon termination.

On the contrary, task deletion is immediate in FreeRTOS, that is, it can neither be refused nor delayed by the target task. As a consequence, the target task may be deleted and cease execution at any time and location in the code, and it will not have the possibility of executing any cleanup handler before terminating. From the point of view of concurrent programming, it means that, for example, if a task is deleted when it is within a critical region controlled by a mutual exclusion semaphore, the semaphore will never be unlocked.

The high-level effect of the deletion is therefore the same as the terminated task never having exited from the critical region: no other tasks will ever be allowed to enter a critical region controlled by the same semaphore in the future. Since this usually corresponds to a complete breakdown of any concurrent program, the direct invocation of `vTaskDelete` should usually be avoided, and it should be replaced by a more sophisticated deletion mechanism.

One simple solution, mimicking the POSIX approach, is to send a deletion request to the target task by some other means—for instance, one of the interprocess communication mechanisms described in Sections 8.2 and 8.3, and design the target task so that it responds to the request by terminating itself at a well-known location in the target task's code and after any required cleanup operation has been carried out.

After creation, it is possible to retrieve the priority of a task and change it by means of the functions

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);

void vTaskPrioritySet(xTaskHandle pxTask,
    unsigned portBASE_TYPE uxNewPriority);
```

Both functions are optional, that is, they can be excluded from the operating system to reduce its code and data space requirements. They both take

a task handle, `pxTask`, as their first argument. The special value `NULL` can be used as a shortcut to refer to the calling task.

The function `vTaskPrioritySet` modifies the priority of a task after it has been created, and `uxTaskPriorityGet` returns the current priority of the task. It should, however, be noted that both the priority given at task creation and the priority set by `vTaskPrioritySet` represent the *baseline* priority of the task.

Instead, `uxTaskPriorityGet` returns its *active* priority, which may differ from the baseline priority when one of the mechanisms to prevent unbounded priority inversion, to be discussed in Chapter 15, is in effect. More specifically, FreeRTOS implements the priority inheritance protocol for mutual exclusion semaphores. See also Section 8.3 for more information.

The pair of optional functions `vTaskSuspend` and `vTaskResume` take an argument of type `xTaskHandle` according to the following prototypes:

```
void vTaskSuspend(xTaskHandle pxTaskToSuspend);

void vTaskResume(xTaskHandle pxTaskToResume);
```

They are used to suspend and resume the execution of the task identified by the argument. For `vTaskSuspend`, the special value `NULL` can be used to suspend the invoking task, whereas, obviously, it is impossible for a task to resume executing of its own initiative.

Like `vTaskDelete`, `vTaskSuspend` also may suspend the execution of a task at an arbitrary point. Therefore, it must be used with care when the task to be suspended contains critical sections—or, more generally, can get mutually exclusive access to one or more shared resources—because those resources are not implicitly released while the task is suspended.

FreeRTOS, like most other monolithic operating systems, does not hold a full task context for interrupt handlers, and hence, they are not full-fledged tasks. One of the consequences of this design choice is that interrupt handlers cannot block or suspend themselves (informally speaking, there is no dedicated space within the operating system to save their context into), and hence, calling `vTaskSuspend(NULL)` from an interrupt handler makes no sense. For related reasons, interrupt handlers are also not allowed to suspend regular tasks by invoking `vTaskSuspend` with a valid `xTaskHandle` as argument.

The function

```
portBASE_TYPE xTaskResumeFromISR(xTaskHandle pxTaskToResume);
```

is a variant of `vTaskResume` that must be used to resume a task from an interrupt handler, also known as Interrupt Service Routine (ISR) in the FreeRTOS jargon.

Since, as said above, interrupt handlers do not have a full-fledged, dedicated task context in FreeRTOS, `xTaskResumeFromISR` cannot perform a full context switch between tasks when needed as its regular counterpart would

do. A context switch would be necessary, for example, when a low-priority task is interrupted and the interrupt handler wakes up a different task, with a higher priority.

On the contrary, `xTaskResumeFromISR` merely returns a nonzero value in this case in order to make the invoking interrupt handler aware of the situation. In response to this indication, the interrupt handler will eventually invoke the FreeRTOS scheduling algorithm so that the higher-priority task just resumed will get executed upon its exit instead of the interrupted one. This is accomplished by invoking a primitive such as, for example, `vPortYieldFromISR()` for the ARM Cortex-M3 port of FreeRTOS. Although the implementation of the primitive is port-dependent, its name is the same across most recent ports of FreeRTOS.

This course of action has the additional advantage that the scheduling algorithm—a quite expensive algorithm to be run in an interrupt context—will be triggered only when strictly necessary to avoid priority inversion.

The optional function

```
portBASE_TYPE xTaskIsTaskSuspended(xTaskHandle xTask)
```

can be used to tell whether a certain task, identified by `xTask`, is currently suspended or not. Its return value will be nonzero if the task is suspended, and zero otherwise.

The function

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

suspends all tasks but the calling one. Interrupt handling is not suspended and is still performed as usual.

Symmetrically, the function

```
portBASE_TYPE xTaskResumeAll(void);
```

resumes all tasks suspended by `vTaskSuspendAll`. In turn, for example, when the priority of one of the resumed tasks is higher than the priority of the invoking task, this may require a context switch. In this case, the context switch is performed immediately within `xTaskResumeAll` itself, and therefore, this function cannot be called from an interrupt handler. The invoking task is later notified that it lost the processor for this reason because it will get a nonzero return value from `xTaskResumeAll`.

Contrary to what could be expected, both `vTaskSuspendAll` and `xTaskResumeAll` are extremely efficient on single-processor systems, such those targeted by FreeRTOS. In fact, these functions are not implemented by suspending and resuming all tasks one by one but by temporarily disabling the operating system scheduler, and the latter operation requires little more work than updating a shared counter.

Hence, they can be used to implement critical regions without using any semaphore and without fear of unbounded priority inversion simply by using

them as a pair of brackets around the critical code. In fact, in a single-processor system, `vTaskSuspendAll` opens a mutual exclusion region because the first task to successfully execute it will effectively prevent all other tasks from being executed until it invokes `xTaskResumeAll`.

Moreover, they also realize an extremely aggressive form of immediate priority ceiling, as will be discussed in Chapter 15, because any task executing between `vTaskSuspendAll` and `xTaskResumeAll` implicitly gets the highest possible priority in the system, too, except interrupt handlers. That said, the method just described has two main shortcomings:

1. Any FreeRTOS primitive that might block the caller for any reason and even temporarily, or might require a context switch, must not be used within this kind of critical region. This is because blocking the only task allowed to run would completely lock up the system, and it is impossible to perform a context switch with the scheduler disabled.
2. Protecting critical regions with a sizable execution time in this way would probably be unacceptable in many applications because it leads to a large amount of unnecessary blocking. This is especially true for high-priority tasks, because if one of them becomes ready for execution while a low-priority task is engaged in a critical region of this kind, it will not run immediately, but only at the end of the critical region itself. See Chapter 15 for additional information on how to compute the worst-case blocking time a task will suffer, depending on the method used to address the unbounded priority inversion problem.

The last function related to task management simply returns the number of tasks currently present in the system, regardless of their state:

```
unsigned portBASE_TYPE uxTaskGetNumberOfTasks(void);
```

Therefore, the count also includes the calling task and blocked tasks. Moreover, it may also include some tasks that have been deleted by `vTaskDelete`. This is a side effect of the delayed dismissal of the operating system's data structures associated with a task upon deletion previously mentioned.

8.2 Message Queues

Message queues are the main Interprocess Communication (IPC) mechanism provided by FreeRTOS. They are also the basic block on which the additional IPC mechanisms discussed in the next sections are built. For this reason, none of the primitives that operate on message queues, summarized in Table 8.2,

TABLE 8.2
Summary of the main message-queue related primitives of FreeRTOS

Function	Purpose	Optional
`xQueueCreate`	Create a message queue	-
`vQueueDelete`	Delete a message queue	-
`xQueueSendToBack`	Send a message	-
`xQueueSendToFront`	Send a high-priority message	-
`xQueueSendToBackFromISR`	... from an interrupt handler	-
`xQueueSendToFrontFromISR`	... from an interrupt handler	-
`xQueueReceive`	Receive a message	-
`xQueueReceiveFromISR`	... from an interrupt handler	-
`xQueuePeek`	Nondestructive receive	-
`uxQueueMessagesWaiting`	Query current queue length	-
`uxQueueMessagesWaitingFromISR`	... from an interrupt handler	-
`xQueueIsQueueEmptyFromISR`	Check if a queue is empty	-
`xQueueIsQueueFullFromISR`	Check if a queue is full	-

can be excluded from the operating system configuration. To use them, it is necessary to include the main FreeRTOS header `FreeRTOS.h`, followed by `queue.h`. There are also some additional primitives, intended either for internal operating system's use or to facilitate debugging, that will not be discussed here.

With respect to the nomenclature presented in Chapter 6, FreeRTOS adopts a symmetric, indirect naming scheme for message queues because the sender task does not name the intended receiver task directly. Rather, both the sender and the receiver make reference to an intermediate entity, that is, the message queue itself.

The synchronization model is asynchronous with finite buffering because the sender always proceeds as soon as the message has been stored into the message queue without waiting for the message to be received at destination. The amount of buffering is fixed and known in advance for each message queue. It is set when the queue is created and cannot be modified afterwards.

No functions are provided for data serialization, but this does not have serious consequences because FreeRTOS message passing is restricted anyway to take place between tasks belonging to a single process. All these tasks are therefore necessarily executed on the same machine and share the same address space.

The function

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

creates a new message queue, given the maximum number of elements it can contain, `uxQueueLength`, and the size of each element, `uxItemSize`, expressed in bytes. Upon successful completion, the function returns a valid message

queue handle to the caller, which must be used for any subsequent operation on the queue just created. When an error occurs, the function returns a `NULL` pointer instead.

When a message queue is no longer needed, it is advisable to delete it, in order to reclaim its memory for future use, by means of the function

```
void vQueueDelete(xQueueHandle xQueue);
```

It should be noted that the deletion of a FreeRTOS message queue takes place immediately and is never delayed even if some tasks are waiting on it. The fate of the waiting tasks then depends on whether they specified a time limit for the wait or not:

- if they did specify a time limit for the message queue operation, they will receive an error indication when the operation times out;
- otherwise, they will be blocked forever.

After a message queue has been successfully created and its `xQueue` handle is available for use, it is possible to send a message to it by means of the functions

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

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

portBASE_TYPE xQueueSendToBackFromISR(
    xQueueHandle xQueue,
    const void *pvItemToQueue,
    portBASE_TYPE *pxHigherPriorityTaskWoken);

portBASE_TYPE xQueueSendToFrontFromISR(
    xQueueHandle xQueue,
    const void *pvItemToQueue,
    portBASE_TYPE *pxHigherPriorityTaskWoken);
```

The first function, `xQueueSendToBack`, sends a message to the back of a message queue. The message to be sent is pointed by the `pvItemToQueue` argument, whereas its size is implicitly assumed to be equal to the size of a message queue item, as declared when the queue was created.

The last argument, `xTicksToWait`, specifies the maximum amount of time allotted to the operation. In particular,

- If the value is `0` (zero), the function returns an error indication to the caller when the operation cannot be performed immediately because the message queue is completely full at the moment.

- If the value is `portMAX_DELAY` (a symbolic constant defined in a port-dependent header file that is automatically included by the main FreeRTOS header file), when the message queue is completely full, the function blocks indefinitely until the space it needs becomes available. For this option to be available, the operating system must be configured to support task suspend and resume, as described in Section 8.1.

- Any other value is interpreted as the maximum amount of time the function will wait, expressed as an integral number of clock *ticks*. See Section 8.4 for more information about ticks.

The return value of `xQueueSendToBack` will be `pdPASS` if the function was successful; any other value means than an error occurred. In particular, the error code `errQUEUE_FULL` means that the function was unable to send the message within the maximum amount of time specified by `xTicksToWait` because the queue was full.

Unlike in POSIX, FreeRTOS messages do not have a full-fledged priority associated with them, and hence, they are normally sent and received in First-In, First-Out (FIFO) order. However, a high-priority message can be sent using the `xQueueSendToFront` function instead of `xQueueSendToBack`. The only difference between those two functions is that `xQueueSendToFront` sends the message to the *front* of the message queue so that it passes over the other messages stored in the queue and will be received before them.

Neither `xQueueSendToBack` nor `xQueueSendToFront` can be called from an interrupt handler. Instead, either `xQueueSendToBackFromISR` or `xQueueSendToFrontFromISR` must be used. The only differences with respect to their regular counterparts are

- They cannot block the caller, and hence, they do not have a `xTicksToWait` argument and always behave as if the timeout were `0`, that is, they return an error indication to the caller if the operation cannot be concluded immediately.

- The argument `pxHigherPriorityTaskWoken` points to a `portBASE_TYPE` variable. The function will set the referenced variable to either `pdTRUE` or `pdFALSE`, depending on whether or not it awakened a task with a priority higher than the task which was running when the interrupt handler started.

  The interrupt handler should use this information, as discussed in Section 8.1, to determine if it should invoke the FreeRTOS scheduling algorithm before exiting.

The functions `xQueueSend` and `xQueueSendFromISR` are just synonyms of `xQueueSendToBack` and `xQueueSendToBackFromISR`, respectively, They have

been retained for backward compatibility with previous versions of FreeRTOS, which did not have the ability to send a message to the front of a message queue.

Messages are always received from the front of a message queue by means of the following functions:

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

portBASE_TYPE xQueueReceiveFromISR(
    xQueueHandle xQueue,
    void *pvBuffer,
    portBASE_TYPE *pxHigherPriorityTaskWoken);

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

All these functions take a message queue handle, `xQueue`, as their first argument; this is the message queue they will work upon. The second argument, `pvBuffer`, is a pointer to a memory buffer into which the function will store the message just received. It must be large enough to hold the message, that is, at least as large as a message queue item as declared when the queue was created.

In the case of `xQueueReceive`, the last argument, `xTicksToWait`, specifies how much time the function should wait for a message to become available if the message queue was completely empty when it was invoked. The valid values of `xTicksToWait` are the same already mentioned when discussing `xQueueSendToBack`.

The return value of `xQueueReceive` will be `pdPASS` if the function was successful; any other value means than an error occurred. In particular, the error code `errQUEUE_EMPTY` means that the function was unable to receive a message within the maximum amount of time specified by `xTicksToWait` because the queue was empty. In this case, the buffer pointed by `pvBuffer` will not contain any valid message after `xQueueReceive` returns.

The function `xQueueReceive`, when successful, *removes* the message it just received from the message queue so that each message sent to the queue is received exactly once. On the contrary, the function `xQueuePeek` simply *copies* the message into the memory buffer indicated by the caller without removing it for the queue. It takes the same arguments as `xQueueReceive`.

The function `xQueueReceiveFromISR` is the variant of `xQueueReceive` that must be used within an interrupt handler. It never blocks, but it re-

turns to the caller in the variable pointed by `pxHigherPriorityTaskWoken`, an indication on whether it awakened a higher priority task or not.

The last group of functions,

```
unsigned portBASE_TYPE
    uxQueueMessagesWaiting(const xQueueHandle xQueue);

unsigned portBASE_TYPE
    uxQueueMessagesWaitingFromISR(const xQueueHandle xQueue);

portBASE_TYPE
    xQueueIsQueueEmptyFromISR(const xQueueHandle xQueue);

portBASE_TYPE
    xQueueIsQueueFullFromISR(const xQueueHandle xQueue);
```

queries various aspects of a message queue status. In particular,

- `uxQueueMessagesWaiting` and `uxQueueMessagesWaitingFromISR` return the number of items currently stored in the message queue `xQueue`. The latter variant must be used when the invoker is an interrupt handler.
- `xQueueIsQueueEmptyFromISR` and `xQueueIsQueueFullFromISR` return a Boolean value that will be `pdTRUE` if the message queue `xQueue` is empty (or full, respectively) and `pdFALSE` otherwise. Both can be invoked safely from an interrupt handler.

These functions should be used with caution because, although the information they return is certainly correct and valid at the time of the call, the scope of its validity is somewhat limited. It is worth mentioning, for example, that the information may *no longer* be valid and should not be relied upon when any subsequent message queue operation is attempted because other tasks may have changed the queue status in the meantime.

For example, the preventive execution of `uxQueueMessageWaiting` by a task, with a result less than the total length of the message queue, is not enough to guarantee that the same task will be able to immediately conclude a `xQueueSendToBack` in the near future: other tasks, or interrupt handlers, may have sent additional items into the queue and filled it completely in the meantime.

The following program shows how the producers/consumers problem can be solved using a FreeRTOS message queue.

```
/* Producers/Consumers problem solved with a FreeRTOS message queue */

#include <stdio.h>     /* For printf() */
#include <stdlib.h>
#include <FreeRTOS.h> /* Main RTOS header */
#include <task.h>      /* Task and time functions */
#include <queue.h>     /* Message queue functions */
```

```
#define  N         10  /* Buffer size (# of items) */

#define  NP         3  /* Number of producer tasks */
#define  NC         2  /* Number of consumer tasks */

/* The minimal task stack size specified in the FreeRTOS configuration
   does not support printf().  Make it larger.
*/
#define  STACK_SIZE     (configMINIMAL_STACK_SIZE+512)

#define  PRODUCER_DELAY 500  /* Delays in producer/consumer tasks */
#define  CONSUMER_DELAY 300

/* Data type for task arguments, for both producers and consumers */
struct task_args_s {
  int n;                /* Task number */
  xQueueHandle q;       /* Message queue to use */
};

void producer_code(void *argv)
{
  /* Cast the argument pointer, argv, to the right data type */
  struct task_args_s *args = (struct task_args_s *)argv;
  int c = 0;
  int item;

  while(1)
  {
    /* A real producer would put together an actual data item.
       Here, we block for a while and then make up a fake item.
    */
    vTaskDelay(PRODUCER_DELAY);
    item = args->n*1000 + c;
    c++;

    printf("Producer #%d - sending  item %6d\n", args->n, item);

    /* Send the data item to the back of the queue, waiting if the
       queue is full.  portMAX_DELAY means that there is no upper
       bound on the amount of wait.
    */
    if(xQueueSendToBack(args->q, &item, portMAX_DELAY) != pdPASS)
      printf("*  Producer %d unable to send\n", args->n);
  }
}

void consumer_code(void *argv)
{
  struct task_args_s *args = (struct task_args_s *)argv;
  int item;

  while(1)
  {
    /* Receive a data item from the front of the queue, waiting if
       the queue is empty.  portMAX_DELAY means that there is no
       upper bound on the amount of wait.
    */
    if(xQueueReceive(args->q, &item, portMAX_DELAY) != pdPASS)
      printf("* Consumer #%d unable to receive\n", args->n);
    else
      printf("Consumer #%d - received item %6d\n", args->n, item);

    /* A real consumer would do something meaningful with the data item.
       Here, we simply block for a while
    */
    vTaskDelay(CONSUMER_DELAY);
  }
```

```
}

int main(int argc, char *argv[])
{
  xQueueHandle q;                     /* Message queue handle */
  struct task_args_s prod_args[NP];   /* Task arguments for producers */
  struct task_args_s cons_args[NC];   /* Task arguments for consumers */
  xTaskHandle dummy;
  int i;

  /* Create the message queue */
  if((q = xQueueCreate(N, sizeof(int))) == NULL)
    printf("* Cannot create message queue of %d elements\n", N);

  else
  {
    /* Create NP producer tasks */
    for(i=0; i<NP; i++)
    {
      prod_args[i].n = i;  /* Prepare the arguments */
      prod_args[i].q = q;

      /* The task handles are not used in the following,
         so a dummy variable is used for them
      */
      if(xTaskCreate(producer_code, "PROD", STACK_SIZE,
                     &(prod_args[i]), tskIDLE_PRIORITY, &dummy) != pdPASS)
        printf("* Cannot create producer #%d\n", i);
    }

    /* Create NC consumer tasks */
    for(i=0; i<NC; i++)
    {
      cons_args[i].n = i;
      cons_args[i].q = q;

      if(xTaskCreate(consumer_code, "CONS", STACK_SIZE,
                     &(cons_args[i]), tskIDLE_PRIORITY, &dummy) != pdPASS)
        printf("* Cannot create consumer #%d\n", i);
    }

    vTaskStartScheduler();
    printf("* vTaskStartScheduler() failed\n");
  }

  /* Since this is just an example, always return a success
     indication, even this might not be true.
  */
  return EXIT_SUCCESS;
}
```

The main program first creates the message queue that will be used for interprocess communication, and then a few producer and consumer tasks. For the sake of the example, the number of tasks to be created is controlled by the macros `NP` and `NC`, respectively.

The producers will all execute the same code, that is, the function `producer_code`. Each of them receives as argument a pointer to a `struct task_args_s` that holds two fields:

1. a task number (`n` field) used to distinguish one task from another in the debugging printouts, and
2. the message queue (`q` field) that the task will use to send data.

In this way, all tasks can work together by only looking at their arguments and without sharing any variable, as foreseen by the message-passing paradigm. Symmetrically, the consumers all execute the function `consumer_code`, which has a very similar structure.

8.3 Counting, Binary, and Mutual Exclusion Semaphores

FreeRTOS provides four different kinds of semaphores, representing different trade-offs between features and efficiency:

1. **Counting semaphores** are the most general kind of semaphore provided by FreeRTOS. They are also the only kind of semaphore actually able to hold a count, as do the abstract semaphores discussed in Chapter 5. The main difference is that, in the FreeRTOS case, the maximum value the counter may assume must be declared when the semaphore is first created.
2. **Binary semaphores** have a maximum value as well as an initial value of one. As a consequence, their value can only be either one or zero, but they can still be used for either mutual exclusion or task synchronization.
3. **Mutex semaphores** are similar to binary semaphores, with the additional restriction that they must *only* be used as mutual exclusion semaphores, that is, `P()` and `V()` must always appear in pairs and must be placed as brackets around critical regions. Hence, mutex semaphores cannot be used for task synchronization. In exchange for this, mutex semaphores implement priority inheritance, which as discussed in Chapter 15, is especially useful to address the unbounded priority inversion problem.
4. **Recursive mutex semaphores** have all the features ordinary mutex semaphores have, and also optionally support the so-called "recursive" locks and unlocks in which a process is allowed to contain more than one nested critical region, all controlled by the same semaphore and delimited by their own `P()`/`V()` brackets, without deadlocking. In this case, the semaphore is automatically locked and unlocked only at the outermost region boundary, as it should.

The four different kinds of semaphores are created by means of distinct functions, all listed in Table 8.3. In order to call any semaphore-related function, it is necessary to include the main FreeRTOS header `FreeRTOS.h`, followed by `semphr.h`.

The function

```
xSemaphoreHandle xSemaphoreCreateCounting(
```

TABLE 8.3
Summary of the semaphore creation/deletion primitives of FreeRTOS

Function	Purpose	Optional
`xSemaphoreCreateCounting`	Create a counting semaphore	*
`vSemaphoreCreateBinary`	Create a binary semaphore	-
`xSemaphoreCreateMutex`	Create a mutex semaphore	*
`xSemaphoreCreateRecursiveMutex`	Create a recursive mutex	*
`vQueueDelete`	Delete a semaphore of any kind	-

```
    unsigned portBASE_TYPE uxMaxCount,
    unsigned portBASE_TYPE uxInitialCount);
```

creates a counting semaphore with a given maximum (`uxMaxCount`) and initial (`uxInitialCount`) value. When successful, it returns to the caller a valid semaphore handle; otherwise, it returns a `NULL` pointer. To create a binary semaphore, use the macro `xSemaphoreCreateBinary` instead:

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

It should be noted that

- Unlike `xSemaphoreCreateCounting`, `xSemaphoreCreateBinary` has no return value. Being a macro rather than a function, it directly manipulates `xSemaphore` instead. Upon success, `xSemaphore` will be set to the semaphore handle just created; otherwise, it will be set to `NULL`.
- Both the maximum and initial value of a binary semaphore are constrained to be `1`, and hence, they are not explicitly indicated.
- Binary semaphores are the only kind of semaphore that is always available for use in FreeRTOS, regardless of its configuration. All the others are optional.

Mutual exclusion semaphores are created by means of two different functions, depending on whether the recursive lock and unlock feature is desired or not:

```
xSemaphoreHandle xSemaphoreCreateMutex(void);
xSemaphoreHandle xSemaphoreCreateRecursiveMutex(void);
```

In both cases, the creation function returns either a semaphore handle upon successful completion, or `NULL`. All mutual exclusion semaphores are unlocked when they are first created, and priority inheritance is always enabled for them.

Since FreeRTOS semaphores of all kinds are built on top of a message queue, they can be deleted by means of the function `vQueueDelete`, already discussed in Section 8.2. Also in this case, the semaphore is destroyed immediately even if there are some tasks waiting on it.

After being created, all kinds of semaphores except recursive, mutual exclusion semaphores are acted upon by means of the functions `xSemaphoreTake` and `xSemaphoreGive`, the FreeRTOS counterpart of `P()` and `V()`, respectively. Both take a semaphore handle `xSemaphore` as their first argument:

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

portBASE_TYPE xSemaphoreGive(xSemaphoreHandle xSemaphore);
```

In addition, `xSemaphoreTake` also takes a second argument, `xBlockTime`, that specifies the maximum amount of time allotted to the operation. The interpretation and valid values that this argument can assume are the same as for message queues.

The function `xSemaphoreTake` returns `pdTRUE` if it was successful, that is, it was able to conclude the semaphore operation before the specified amount of time elapsed. Otherwise, it returns `pdFALSE`. Similarly, `xSemaphoreGive` returns `pdTRUE` when successful, and `pdFALSE` if an error occurred.

The function `xSemaphoreGiveFromISR` is the variant of `xSemaphoreGive` that must be used within an interrupt handler:

```
portBASE_TYPE xSemaphoreGiveFromISR(xSemaphoreHandle xSemaphore,
    portBASE_TYPE *pxHigherPriorityTaskWoken);
```

Like many other FreeRTOS primitives that can be invoked from an interrupt handler, this function returns to the caller, in the variable pointed by `pxHigherPriorityTaskWoken`, an indication on whether or not it awakened a task with a priority higher than the task which was running when the interrupt handler started.

The interrupt handler should use this information, as discussed in Section 8.1, to determine if it should invoke the FreeRTOS scheduling algorithm before exiting. The function also returns either `pdTRUE` or `pdFALSE`, depending on whether it was successful or not.

The last pair of functions to be discussed here are the counterpart of `xSemaphoreTake` and `xSemaphoreGive`, to be used with recursive mutual exclusion semaphores:

```
portBASE_TYPE xSemaphoreTakeRecursive(xSemaphoreHandle xMutex,
    portTickType xBlockTime);

portBASE_TYPE xSemaphoreGiveRecursive(xSemaphoreHandle xMutex);
```

Both their arguments and return values are the same as `xSemaphoreTake` and `xSemaphoreGive`, respectively. Table 8.4 summarizes the FreeRTOS functions that work on semaphores.

The following program shows how the producers–consumers problem can be solved using a shared buffer and FreeRTOS semaphores.

TABLE 8.4
Summary of the semaphore manipulation primitives of FreeRTOS

Function	Purpose	Optional
`xSemaphoreTake`	Perform a `P()` on a semaphore	-
`xSemaphoreGive`	Perform a `V()` on a semaphore	-
`xSemaphoreGiveFromISR`	... from an interrupt handler	-
`xSemaphoreTakeRecursive`	`P()` on a recursive mutex	*
`xSemaphoreGiveRecursive`	`V()` on a recursive mutex	*


```
/* Producers/Consumers problem solved with FreeRTOS semaphores */

#include <stdio.h>     /* For printf() */
#include <stdlib.h>
#include <FreeRTOS.h> /* Main RTOS header */
#include <task.h>      /* Task and time functions */
#include <semphr.h>    /* Semaphore functions */

#define  N         10  /* Buffer size (# of items) */

#define  NP         3  /* Number of producer tasks */
#define  NC         2  /* Number of consumer tasks */

/* The minimal task stack size specified in the FreeRTOS configuration
   does not support printf(). Make it larger.
*/
#define  STACK_SIZE     (configMINIMAL_STACK_SIZE+512)

#define  PRODUCER_DELAY 500  /* Delays in producer/consumer tasks */
#define  CONSUMER_DELAY 300

/* Data type for task arguments, for both producers and consumers */
struct task_args_s {
  int n;               /* Task number */
};

/* Shared variables and semaphores. They implement the shared
   buffer, as well as mutual exclusion and task synchronization
*/

int buf[N];
int in = 0, out = 0;
xSemaphoreHandle empty, full;
xSemaphoreHandle mutex;


void producer_code(void *argv)
{
  /* Cast the argument pointer, argv, to the right data type */
  struct task_args_s *args = (struct task_args_s *)argv;
  int c = 0;
  int item;

  while(1)
  {
    /* A real producer would put together an actual data item.
       Here, we block for a while and then make up a fake item.
    */
    vTaskDelay(PRODUCER_DELAY);
    item = args->n*1000 + c;
    c++;
```

```
    printf("Producer #%d - sending  item %6d\n", args->n, item);

    /* Synchronize with consumers */
    if(xSemaphoreTake(empty, portMAX_DELAY) != pdTRUE)
      printf("*  Producer %d unable to take 'empty'\n", args->n);

    /* Mutual exclusion for buffer access */
    else if(xSemaphoreTake(mutex, portMAX_DELAY) != pdTRUE)
      printf("*  Producer %d unable to take 'mutex'\n", args->n);

    else
    {
      /* Store data item into 'buf', update 'in' index */
      buf[in] = item;
      in = (in + 1) % N;

      /* Release mutex */
      if(xSemaphoreGive(mutex) != pdTRUE)
        printf("*  Producer %d unable to give 'mutex'\n", args->n);

      /* Synchronize with consumers */
      if(xSemaphoreGive(full) != pdTRUE)
        printf("*  Producer %d unable to give 'full'\n", args->n);
    }
  }
}

void consumer_code(void *argv)
{
  struct task_args_s *args = (struct task_args_s *)argv;
  int item;

  while(1)
  {
    /* Synchronize with producers */
    if(xSemaphoreTake(full, portMAX_DELAY) != pdTRUE)
      printf("*  Consumer %d unable to take 'full'\n", args->n);

    /* Mutual exclusion for buffer access */
    else if(xSemaphoreTake(mutex, portMAX_DELAY) != pdTRUE)
      printf("*  Consumer %d unable to take 'mutex'\n", args->n);

    else
    {
      /* Get data item from 'buf', update 'out' index */
      item = buf[out];
      out = (out + 1) % N;

      /* Release mutex */
      if(xSemaphoreGive(mutex) != pdTRUE)
        printf("*  Consumer %d unable to give 'mutex'\n", args->n);

      /* Synchronize with producers */
      if(xSemaphoreGive(empty) != pdTRUE)
        printf("*  Consumer %d unable to give 'full'\n", args->n);

      /* A real consumer would do something meaningful with the data item.
         Here, we simply print it out and block for a while
      */
      printf("Consumer #%d - received item %6d\n", args->n, item);
      vTaskDelay(CONSUMER_DELAY);
    }
  }
}

int main(int argc, char *argv[])
{
```

```
  struct task_args_s prod_args[NP];  /* Task arguments for producers */
  struct task_args_s cons_args[NC];  /* Task arguments for consumers */
  xTaskHandle dummy;
  int i;

  /* Create the two synchronization semaphores, empty and full.
     They both have a maximum value of N (first argument), and
     an initial value of N and 0, respectively
  */
  if((empty = xSemaphoreCreateCounting(N, N)) == NULL
     || (full = xSemaphoreCreateCounting(N, 0)) == NULL)
    printf("* Cannot create counting semaphores\n");

  /* Create the mutual exclusion semaphore */
  else if((mutex = xSemaphoreCreateMutex()) == NULL)
    printf("* Cannot create mutex\n");

  else
  {
    /* Create NP producer tasks */
    for(i=0; i<NP; i++)
    {
      prod_args[i].n = i;  /* Prepare the argument */

      /* The task handles are not used in the following,
         so a dummy variable is used for them
      */
      if(xTaskCreate(producer_code, "PROD", STACK_SIZE,
                     &(prod_args[i]), tskIDLE_PRIORITY, &dummy) != pdPASS)
        printf("* Cannot create producer #%d\n", i);
    }

    /* Create NC consumer tasks */
    for(i=0; i<NC; i++)
    {
      cons_args[i].n = i;

      if(xTaskCreate(consumer_code, "CONS", STACK_SIZE,
                     &(cons_args[i]), tskIDLE_PRIORITY, &dummy) != pdPASS)
        printf("* Cannot create consumer #%d\n", i);
    }

    vTaskStartScheduler();
    printf("* vTaskStartScheduler() failed\n");
  }

  /* Since this is just an example, always return a success
     indication, even this might not be true.
  */
  return EXIT_SUCCESS;
}
```

As before, the main program takes care of initializing the shared synchronization and mutual exclusion semaphores needed by the application, creates several producers and consumers, and then starts the scheduler. Even if the general parameter passing strategy adopted in the previous example has been maintained, the only argument passed to the tasks is their identification number because the semaphores, as well as the data buffer itself, are shared and globally accessible.

With respect to the solution based on message queues, the most important difference to be remarked is that, in this case, the data buffer shared between the producers and consumers must be allocated and handled explicitly by

TABLE 8.5
Summary of the time-related primitives of FreeRTOS

Function	Purpose	Optional
`xTaskGetTickCount`	Get current time, in ticks	-
`vTaskDelay`	Relative time delay	*
`vTaskDelayUntil`	Absolute time delay	*

the application code, instead of being hidden behind the operating system's implementation of message queues. In the example, it has been implemented by means of the (circular) buffer `buf[]`, assisted by the input and output indexes `in` and `out`.

8.4 Clocks and Timers

Many of the activities performed in a real-time system ought to be correlated, quite intuitively, with time. Accordingly, all FreeRTOS primitives that may potentially block the caller, such as those discussed in Sections 8.2 and 8.3, allow the caller to specify an upper bound to the blocking time.

Moreover, FreeRTOS provides a small set of primitives, listed in Table 8.5, to *keep track* of the elapsed time and *synchronize* a task with it by delaying its execution. Since they do not have their own dedicated header file, it is necessary to include the main FreeRTOS header file, `FreeRTOS.h`, followed by `task.h`, in order to use them.

In FreeRTOS, the same data type, `portTickType`, is used to represent both the current time and a time interval. The current time is simply represented by the number of clock *ticks* elapsed from when the operating system scheduler was first started. The length of a tick depends on the operating system configuration and, to some extent, on hardware capabilities.

The configuration macro `configTICK_RATE_HZ` represents the tick frequency in Hertz. In addition, most porting layers define the macro `portTICK_RATE_MS` as the fraction `1000/configTICK_RATE_HZ` so that is represents the tick period, expressed in milliseconds. Both of them can be useful to convert back and forth between the usual time measurement units and clock ticks.

The function

```
portTickType xTaskGetTickCount(void);
```

returns the current time, expressed in ticks. It is quite important to keep in mind that the returned value comes from a tick counter of type `portTickType` maintained by FreeRTOS. Barring some details, the operating system resets the counter to zero when the scheduler is first started, and increments

it with the help of a periodic interrupt source, running at a frequency of `configTICK_RATE_HZ`.

In most microcontrollers, the tick counter data type is either a 16- or 32-bit unsigned integer, depending on the configuration. Therefore, it will sooner or later wrap around and resume counting from zero. For instance, an unsigned, 32-bit counter incremented at 1000 Hz—a common configuration choice for FreeRTOS—will wrap around after about 1193 hours, that is, a bit more than 49 days.

It is therefore crucial that any application planning to "stay alive" for a longer time, as many real-time applications must do, is aware of the wraparound and handles it appropriately if it manipulates time values directly. If this is not the case, the application will be confronted with time values that suddenly "jump into the past" when a wraparound occurs, with imaginable consequences. The delay functions, to be discussed next, already handle time wraparound automatically, and hence, no special care is needed to use them.

Two distinct delay functions are available, depending on whether the delay should be *relative*, that is, measured with respect to the instant in which the delay function is invoked, or *absolute*, that is, until a certain instant in the future, measured as a number of ticks, from when the scheduler has been started:

```
void vTaskDelay(portTickType xTicksToDelay);
void vTaskDelayUntil(portTickType *pxPreviousWakeTime,
   portTickType xTimeIncrement);
```

The function `vTaskDelay` implements a relative time delay: it blocks the calling task for `xTicksToDelay` ticks, then returns. As shown in Figure 8.1, the time interval is relative to the time of the call and the amount of delay is fixed, that is, `xTicksToDelay`.

Instead, the function `vTaskDelayUntil` implements an absolute time delay: referring again to Figure 8.1, the next wake-up time is calculated as the previous one, `*pxPreviousWakeTime`, plus the time interval `xTimeIncrement`. Hence, the amount of delay varies from call to call, and the function might not block at all if the prescribed wake-up time is already in the past. Just before returning, the function also increments `*pxPreviousWakeTime` by `xTimeIncrement` so that it is ready for the next call.

The right kind of delay to be used depends on its purpose. A relative delay may be useful, for instance, if an I/O device must be allowed (at least) a certain amount of time to react to a command. In this case, the delay must be measured from when the command has actually been sent to the device, and a relative delay makes sense.

On the other hand, an absolute delay is better when a task has to carry out an operation periodically because it guarantees that the period will stay constant even if the response time of the task—the grey rectangles in Figure 8.1—varies from one instance to another.

The last example program, listed below, shows how absolute and relative

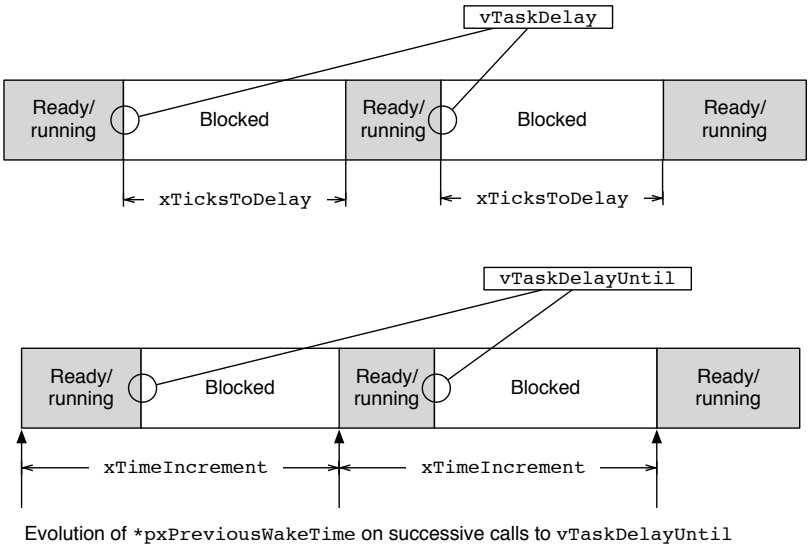

FIGURE 8.1
Comparison between relative and absolute time delays, as implemented by `vTaskDelay` and `vTaskDelayUntil`.

delays can be used in an actual piece of code. When run for a long time, the example is also useful in better highlighting the difference between those two kinds of delay. In fact, it can be seen that the wake-up time of task `rel_delay` (that uses a relative delay) not only drifts forward but is also irregular because the variations in its response time are not accounted for when determining the delay before its next activation. On the contrary, the wake-up time of task `abs_delay` (that uses an absolute delay) does not drift, and it strictly periodic.

```
/* FreeRTOS vTaskDelay versus vTaskDelayUntil */

#include <stdio.h>     /* For printf() */
#include <stdlib.h>
#include <FreeRTOS.h> /* Main RTOS header */
#include <task.h>      /* Task and time functions */

/* The minimal task stack size specified in the FreeRTOS configuration
   does not support printf(). Make it larger.

   Period is the nominal period of the tasks to be created
*/
#define  STACK_SIZE     (configMINIMAL_STACK_SIZE+512)
#define  PERIOD         ((portTickType)100)

/* Periodic task with relative delay (vTaskDelay) */
void rel_delay(void *dummy)
{
  while(1)
  {
    /* Block for PERIOD ticks, measured from 'now' */
```

```
    vTaskDelay(PERIOD);

    printf("rel_delay active at --------- %9u ticks\n",
           (unsigned int)xTaskGetTickCount());
  }
}

/* Periodi task with absolute delay (vTaskDelayUntil) */
void abs_delay(void *dummy)
{
  portTickType last_wakeup = xTaskGetTickCount();

  while(1)
  {
    /* Block until the instant last_wakeup + PERIOD,
       then update last_wakeup and return
    */
    vTaskDelayUntil(&last_wakeup, PERIOD);

    printf("abs_delay active at %9u --------- ticks\n",
           (unsigned int)xTaskGetTickCount());
  }
}

int main(int argc, char *argv[])
{
  xTaskHandle dummy;

  /* Create the two tasks to be compared, with no arguments.
     The task handles are not used, hence they are discarded
  */
  if(xTaskCreate(rel_delay, "REL", STACK_SIZE, NULL,
                 tskIDLE_PRIORITY, &dummy) != pdPASS)
    printf("* Cannot create task rel_delay\n");

  else if(xTaskCreate(abs_delay, "ABS", STACK_SIZE, NULL,
                      tskIDLE_PRIORITY, &dummy) != pdPASS)
    printf("* Cannot create task abs_delay\n");

  else
  {
    vTaskStartScheduler();
    printf("* vTaskStartScheduler() failed\n");
  }

  /* Since this is just an example, always return a success
     indication, even this might not be true.
  */
  return EXIT_SUCCESS;
}
```

8.5 Summary

In this chapter, we filled the gap between the abstract concepts of multiprogramming and IPC, presented in Chapters 3 through 6, and what real-time operating systems actually offer programmers when the resources at their disposal are severely constrained.

FreeRTOS, an open-source, real-time operating system targeted to small

embedded systems has been considered as a case study. This is in sharp contrast to what was shown in Chapter 7, which deals instead with a full-fledged, POSIX-compliant operating system like Linux. This also gives the readers the opportunity of comparing several real-world code examples, written in C for these two very dissimilar execution environments.

The first important difference is about how FreeRTOS implements multiprocessing. In fact, to cope with hardware limitations and to simplify the implementation, FreeRTOS does not support multiple processes but only threads, or *tasks*, all living within the same address space. With respect to a POSIX system, task creation and deletion are much simpler and less sophisticated, too.

For what concerns IPC, the primitives provided by FreeRTOS are rather established, and do not depart significantly from any of the abstract concepts discussed earlier in the book. The most important aspect that is worth noting is that FreeRTOS sometimes maps a single abstract concept into several distinct, concrete objects.

For example, the abstract *semaphore* corresponds to four different "flavors" of semaphore in FreeRTOS, each representing a different trade-off between the flexibility and power of the object and the efficiency of its implementation. This is exactly the reason why this approach is rather common and is also taken by most other, real-world operating systems.

Another noteworthy difference is that, quite unsurprisingly, time plays a central role in a real-time operating system. For this reason, all abstract primitives that may block a process (such as a `P()` on a semaphore) have been extended to support a *timeout* mechanism. In this way, the caller can specify a maximum amount of time it is willing to block for any given primitive and do not run the risk of being blocked forever if something goes wrong.

Last but not least, FreeRTOS also provides a couple of primitives to synchronize a task with the elapsed time. Even if those primitives were not discussed in abstract terms, they are especially important anyway in a real-time system because they lay out the foundation for executing any kind of periodic activity in the system. Moreover, they also provide a convenient way to insert a controlled delay in a task without wasting processing power for it.