4 Execution Models for Embedded Systems

CONTENTS

A key design point of any embedded system is the selection of an appropriate *execution model* that, generally speaking, can be defined as the set of rules and constraints that organize the execution of the embedded system's activities.

Traditionally, many embedded systems—especially the smallest and simplest ones—were designed around the *cyclic executive* execution model that is simple, efficient, and easy to understand. However, it lacks modularity and its application becomes more and more difficult as the size and complexity of the embedded system increase.

For this reason, *task-based scheduling* is nowadays becoming a strong competitor, even if it is more complex for what concerns both software development and system requirements. This chapter discusses and compares the two approaches to help the reader choose the best one for a given design problem and introduce the more general topic of choosing the right execution model for the application at hand.

4.1 THE CYCLIC EXECUTIVE

The cyclic executive execution model is fully described in Reference [16] and supports the execution of a number of *periodic tasks*. A periodic task, usually denoted as τ_i, is an activity or action to be performed repeatedly, at constant time intervals that represent the period of the process. The period of task τ_i is denoted as T_i. The same term, cyclic executive, is also used to denote the program responsible for controlling task execution.

It must be remarked that, even though periodic tasks are probably one of the simplest ways to model execution, they still represent the most important software com-

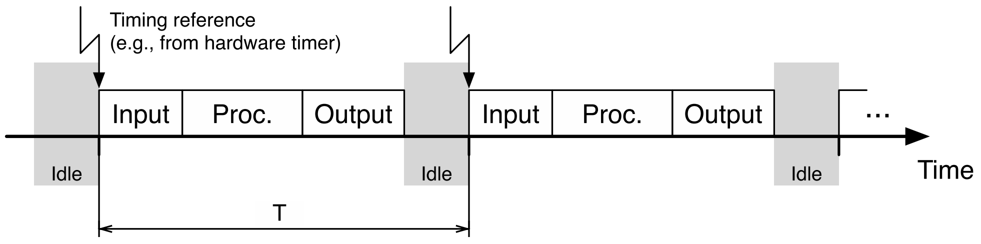

Figure 4.1 A prototypical cyclic executive.

ponents of an embedded, real-time system. For this reason, this book will mainly focus on this kind of task. Interested readers should refer to References [26, 111] for a more thorough and formal description of task scheduling theory, which includes more sophisticated and flexible task models. Reference [152] provides a comprehensive overview about the historical evolution of task scheduling theory and practice. To stay within the scope of the book, discussion will be limited to single-processor systems.

In its most basic form, a cyclic executive is very close to the typical intuitive structure of a simple embedded control system, whose time diagram is depicted in Figure 4.1. As shown in the figure, the activities carried out by the control system can be seen as three tasks, all with the same period T.

1. An *input* task interacts with input devices (e.g., sensors) to retrieve the input variables of the control algorithm.
2. A *processing* task implements the control algorithm and computes the output variables based on its inputs.
3. An *output* task delivers the values of the output variables to the appropriate output devices (e.g., actuators).

To guarantee that the execution period is fixed, regardless of unavoidable variations in task execution time, the cyclic executive makes use of a timing reference signal and synchronizes with it at the beginning of each cycle. Between the end of the output task belonging to the current cycle and the beginning of the input task belonging to the next cycle—that is, while the synchronization with the timing reference is being performed—the system is idle. Those areas are highlighted in gray in Figure 4.1.

In most embedded systems, hardware timers are available to provide accurate timing references. They can deliver timing signals by means of periodic interrupt requests or, in simple cases, the software can perform a polling loop on a counter register driven by the timer itself.

For what concerns the practical implementation, the cyclic executive corresponding to the time diagram shown in Figure 4.1 can be coded as an infinite loop containing several function calls.

```
while(1)
{
    WaitForReference();
    Input();
    Processing();
    Output();
}
```

In the listing above, the function `WaitForReference` is responsible for synchronizing loop execution with real time and ensuring that it is executed once every period T. The three functions `Input`, `Processing`, and `Output` correspond to the input, processing, and output tasks. The same code can also be seen as a *scheduling table*, a table of function calls in which each call represents a task.

Even though the example discussed so far is extremely simple, it already reveals several peculiar features of a typical cyclic executive. They must be remarked upon right from the beginning because they are very different from what task-based scheduling does, which is to be discussed in Section 4.4. Moreover, these differences have an important impact on programmers and affect the way they think about and implement their code.

In the cyclic executive implementation, tasks are mapped onto functions, which are called sequentially from the main loop. Task interleaving is decided offline, when the cyclic executive is designed, and then performed in a completely deterministic way. Since the schedule is laid out completely once and for all, it is a "proof by construction" that all tasks can be successfully executed and satisfy their timing constraints.

Moreover, task functions can freely access global variables to exchange data. Since the function execution sequence is fixed, no special care is needed to read and write these variables. For the same reason, enforcing producer–consumer constraints, that is, making sure that the value of a variable is not used before it is set, is relatively easy. Even complex data structures are straightforward to maintain because, by construction, a function will never be stopped "midway" by the execution of another function. As a consequence, a function will never find the data structure in an inconsistent state. The only exception to this general rule is represented by asynchronously executed code, such as signal and interrupt handlers.

4.2 MAJOR AND MINOR CYCLES

In theory, nothing prevents a programmer from crafting a cyclic executive completely by hand, according to his/her own design. From the practical point of view, it is often useful to make use of a well-understood structure, in the interest of software clarity and maintainability. For this reason, most cyclic executives are designed according to the following principles.

In order to accommodate tasks with different periods, an issue that was not considered in the simple example of Section 4.1, the scheduling table is divided into sections, or slices. Individual slices are called *minor cycles* and have all the same duration. The table as a whole is known as *major cycle*.

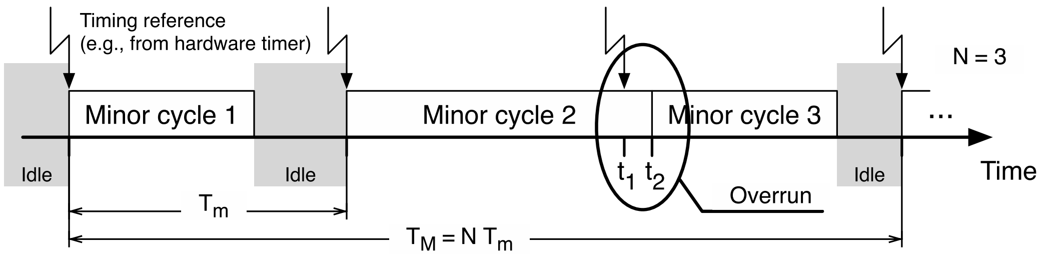

Figure 4.2 Major and minor cycles in a cyclic executive.

As shown in Figure 4.2, which depicts a cyclic executive in which the major cycle is composed of 3 minor cycles, minor cycle boundaries are synchronization points. At these points, the cyclic executive waits for a timing reference signal. As a consequence, the task activated at the very beginning of a minor cycle is synchronized with real time as accurately as possible. On the other hand, within a minor cycle, tasks are activated in sequence and suffer from any execution time jitter introduced by other tasks that precede them.

Minor cycle synchronization is also very useful to detect the most critical error that a cyclic executive may experience, that is, a cycle *overrun*. An overrun occurs when—contrary to what was assumed at design time—some task functions have an execution time that is longer than expected and the accuracy of the overall cyclic executive timing can no longer be guaranteed.

As an example, Figure 4.2 shows what happens if the task functions invoked in minor cycle 2 exceed the minor cycle length. If the cyclic executive does not handle this condition and simply keeps executing task functions according to the designed sequence, the whole minor cycle 3 is shifted to the right and its beginning is no longer properly aligned with the timing reference.

Overrun detection at a synchronization point occurs in two different ways depending on how synchronization is implemented:

1. If the timing reference consists of an interrupt request from a hardware timer, the cyclic executive will detect the overrun as soon as the interrupt occurs, that is, at time t_1 in Figure 4.2.
2. If the cyclic executive synchronizes with the timing reference by means of a polling loop, it will be able to detect the overrun only when minor cycle 2 eventually ends, that is, when the last task function belonging to that cycle returns. At that point, the cyclic executive should wait until the end of the cycle, but it will notice that the time it should wait for is already in the past. In the figure, this occurs at time t_2.

Generally speaking, the first method provides a more timely *detection* of overruns, as well as being an effective way to deal with indefinite non-termination of a task function. However, a proper *handling* of an overrun is often more challenging than detection itself. Referring back to Figure 4.2, it would be quite possible for the cyclic

Table 4.1
A Simple Task Set for a Cyclic Executive

$T_m = 20\,\text{ms}$, $T_M = 120\,\text{ms}$

Task τ_i	**Period** T_i **(ms)**	**Execution time** C_i **(ms)**	k_i
τ_1	20	6	1
τ_2	40	4	2
τ_3	60	2	3
τ_4	120	6	6

executive to abort minor cycle 2 at t_1 and immediately start minor cycle 3, but this may lead to significant issues at a later time. For instance, if a task function was manipulating a shared data structure when it was aborted, the data structure was likely left in an inconsistent state.

In addition, an overrun is most often indicative of a *design*, rather than an execution issue, at least when it occurs systematically. For this reason, overrun handling techniques are often limited to error reporting to the supervisory infrastructure (when it exists), followed by a system reset or shutdown.

Referring back to Figure 4.2, if we denote the minor cycle period as T_m and there are N minor cycles in a major cycle ($N = 3$ in the figure), the major cycle period is equal to $T_M = N T_m$. Periodic tasks with different periods can be placed within this framework by invoking their task function from one or more minor cycles. For instance, invoking a task function from all minor cycles leads to a task period $T = T_m$, while invoking a task function just in the first minor cycle gives a task period $T = T_M$.

Given a set of s periodic tasks $\tau_1, \dots, \tau_s$, with their own periods $T_1, \dots, T_s$, it is interesting to understand how to choose T_m and T_M appropriately, in order to accommodate them all. Additional requirements are to keep T_m as big as possible (to reduce synchronization overheads), and keep T_M as small as possible (to decrease the size of the scheduling table).

It is easy to show that an easy, albeit not optimal, way to satisfy these constraints is to set the minor cycle length to the *greatest common divisor* (GCD) of the task periods, and set the major cycle length to their *least common multiple* (LCM). Reference [16] contains information on more sophisticated methods. In formula:

$$T_m = \gcd(T_1, \dots, T_s) \tag{4.1}$$

$$T_M = \text{lcm}(T_1, \dots, T_s) \tag{4.2}$$

When T_m and T_M have been chosen in this way, the task function corresponding to τ_i must be invoked every $k_i = T_i / T_m$ minor cycles. Due to Equation (4.1), it is guaranteed that every k_i is an integer and a sub-multiple of N, and hence, the schedule can actually be built in this way.

As an example, let us consider the task set listed in Table 4.1. Besides the task periods, the table also lists their execution time and the values k_i, computed as dis-

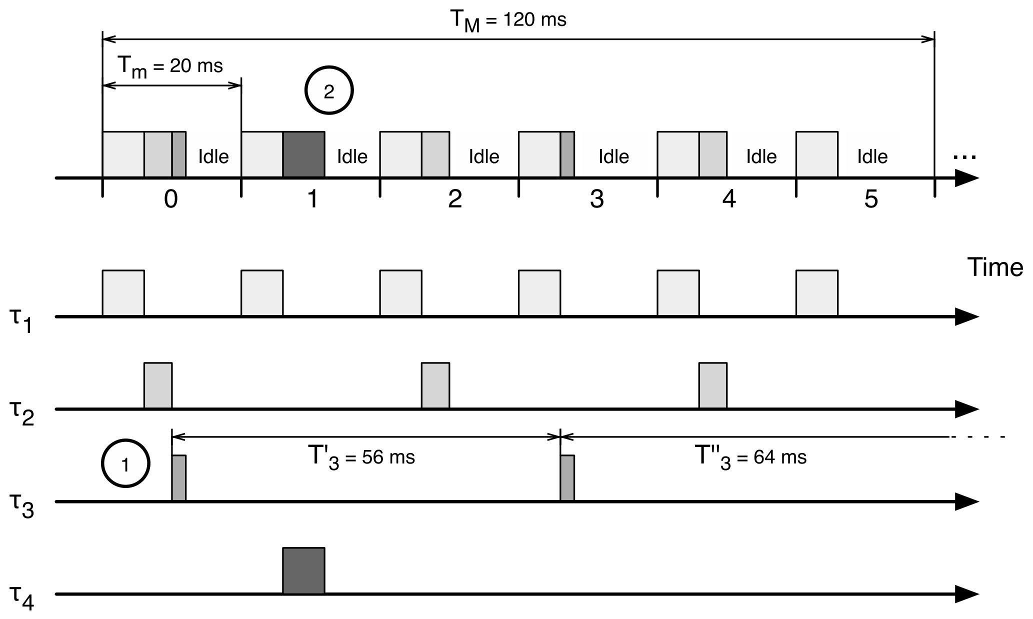

Figure 4.3 The cyclic executive schedule for the task set of Table 4.1.

cussed above. The execution time—usually denoted as C_i—is defined as the amount of CPU time it takes to execute the task function corresponding to τ_i. It is especially important to evaluate or, at least, estimate these values at design time, in order to allocate task functions in the right minor cycles and avoid overruns.

In this example, from Equations (4.1) and (4.2), it is:

$$\begin{aligned} T_m &= \gcd(20,40,60,120) = 20\,\text{ms} \\ T_M &= \text{lcm}(20,40,60,120) = 120\,\text{ms} \end{aligned}$$

and there are $N = 6$ minor cycles in every major cycle.

The corresponding cyclic executive schedule is shown at the top of Figure 4.3. The bottom part of the figure shows the execution timeline of individual tasks and demonstrates that all tasks are executed properly. This example is also useful to remark on two general properties of cyclic executives:

1. It is generally *impossible* to ensure that all tasks will be executed with their exact period for every instance, although this is true on average. In the schedule shown in Figure 4.3 this happens to τ_3, which is executed alternatively at time intervals of $T_3' = 56\,\text{ms}$ and $T_3'' = 64\,\text{ms}$. In other words, on average the task period is still $T_3 = 60\,\text{ms}$ but every instance will suffer from an activation *jitter* of $\pm 4\,\text{ms}$ around the average period.

   In this particular example, the jitter is due to the presence of τ_2 in minor cycle 0, whereas it is absent in minor cycle 3. In general, it may be possible to *choose* which tasks should suffer from jitter up to a certain extent, but it cannot be removed completely. In this case, swapping the activation of τ_2 and τ_3 in minor cycle 0 removed jitter from τ_3, but introduces some jitter on τ_2. Moving τ_2 so that

Table 4.2
A Task Set That Requires Task Splitting to Be Scheduled

$T_m = 20\,\text{ms}$, $T_M = 120\,\text{ms}$

Task τ_i	**Period T_i (ms)**	**Execution time C_i (ms)**	k_i
τ_1	20	6	1
τ_2	40	4	2
τ_3	60	2	3
τ_4	120	**16**	6

it is activated in the odd minor cycles instead of the even ones does not completely solve the problem, either.

2. There may be some *freedom* in where to place a certain task within the schedule. In the example, τ_4 may be placed in any minor cycle. In this case, the minor cycle is chosen according to secondary goals. For instance, choosing one of the "emptiest" minor cycles—as has been done in the example—is useful to reduce the likelihood of overrun, in case some of the C_i have been underestimated.

4.3 TASK SPLITTING AND SECONDARY SCHEDULES

In some cases, it may be impossible to successfully lay out a cyclic executive schedule by considering tasks as *indivisible* execution blocks, as has been done in the previous example. This is trivial in the case when the execution time of a task τ_i exceeds the minor cycle length, that is, $C_i > T_m$, but it may also happen in less extreme cases, when other tasks are present and must also be considered in the schedule.

Table 4.2 lists the characteristics of a task set very similar to the one used in the previous example (and outlined in Table 4.1), with one important difference. The execution time of τ_4 has been increased to $C_4 = 16\,\text{ms}$. Since the task periods have not been changed, T_m and T_M are still the same as before.

Even though it is $C_4 < T_m$, it is clearly impossible to fit τ_4 *as a whole* in any minor cycle, because none of them has enough idle time to accommodate it. It should also be remarked that the issue is not related to the availability of sufficient idle time in general, which would make the problem impossible to solve. Indeed, it is easy to calculate that in T_M, which is also the period of τ_4, the idle time left by the execution of τ_1, τ_2, and τ_3 amounts to 68 ms, which is much greater than C_4.

In cases like this, the only viable solution is to *split* the task (or tasks) with a significant C_i into two (or more) slices and allocate individual slices in different minor cycles.

As shown in Figure 4.4 it is possible, for example, to split τ_4 into two slices with equal execution time and allocate them in two adjacent minor cycles. Even though the overall execution timeline is still satisfactory, task splitting has two important, negative consequences:

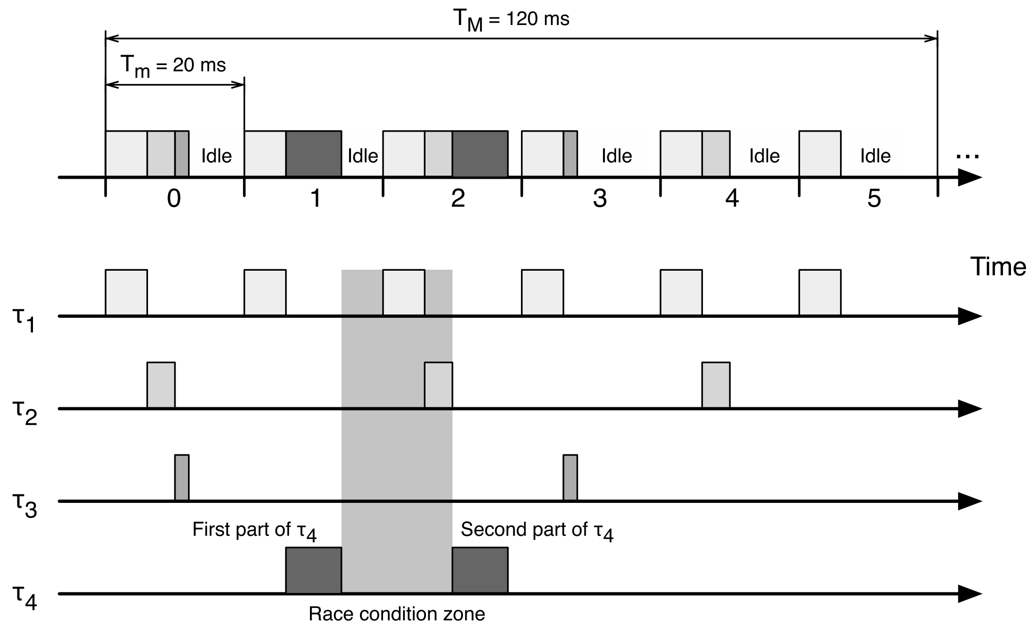

Figure 4.4 A possible task split for the task set of Table 4.2.

1. The criteria used to decide where a task must be split depend on task *timing* and not on its *behavior*. For this reason, when a task is split, it may be necessary to cut through its code in a way that has little to do with the internal, logic structure of the task itself. As a result, the code will likely be harder to understand and maintain.
2. Even more importantly, splitting a task as has been done with τ_4 in the example, introduces a *race condition zone*, or window, depicted as a gray area in Figure 4.4. Generally speaking, a race condition occurs whenever two tasks are allowed to access a shared data structure in an uncontrolled way. A race condition window is a time frame in which, due to the way tasks are scheduled, a race condition *may* occur.

   In this case, if τ_4 shares a data structure with τ_1 or τ_2—the tasks executed within the race condition zone—special care must be taken.

   - If τ_4 modifies the data structure in any way, the first part of τ_4 must be implemented so that it leaves the data structure in a consistent state, otherwise τ_1 and τ_2 may have trouble using it.
   - If τ_1 or τ_2 update the shared data structure, the first and second part of τ_4 must be prepared to find the data structure in two different states and handle this scenario appropriately.

   In both cases, it is clear that splitting a task does not merely require taking its code, dividing it into pieces and putting each piece into a separate function. Instead, some non-trivial modifications to the code are needed to deal with race conditions. These modifications add to code complexity and, in a certain sense, compromise one important feature of cyclic executives recalled in Section 4.1, that is, their ability to handle shared data structures in a straightforward, intuitive way.

Table 4.3
A Task Set Including a Task with a Large Period

$T_m = 20\,\text{ms}$, $T_M = \mathbf{1200}\,\text{ms}$

Task τ_i	**Period** T_i **(ms)**	**Execution time** C_i **(ms)**	k_i
τ_1	20	6	1
τ_2	40	4	2
τ_3	60	2	3
τ_4	**1200**	6	**60**

It is also useful to remark that what has been discussed so far is merely a simple introduction to the data sharing issues to be confronted when a more sophisticated and powerful execution model is adopted. This is the case of task-based scheduling, which will be presented in Section 4.4. To successfully solve those issues, a solid grasp of concurrent programming techniques is needed. This will be the topic of Chapter 5.

Besides tasks with a significant execution time, tasks with a large period, with respect to the others, may be problematic in cyclic executive design, too. Let us consider the task set listed in Table 4.3. The task set is very close to the one used in the first example (Table 4.1) but T_4, the period of τ_4, has been increased to 1200 ms, that is, ten times as before.

It should be remarked that tasks with a large period, like τ_4, are not at all uncommon in real-time embedded systems. For instance, such a task can be used for periodic status and data logging, in order to summarize system activities over time. For these tasks, a period on the order of 1 s is quite common, even though the other tasks in the same system, dealing with data acquisition and control algorithm, require much shorter periods.

According to the general rules to determine T_m and T_M given in Equations (4.1) and (4.2), it should be:

$$\begin{aligned} T_m &= \gcd(20, 40, 60, 1200) = 20\,\text{ms} \\ T_M &= \text{lcm}(20, 40, 60, 1200) = 1200\,\text{ms} \end{aligned}$$

In other words, the minor cycle length T_m would still be the same as before, but the major cycle length T_M would increase tenfold, exactly like T_4. As a consequence, a major cycle would now consist of $N = T_M/T_m = 60$ minor cycles instead of 6. What is more, as shown in Figure 4.5, 59 minor cycles out of 60 would still be exactly the same as before, and would take care of scheduling τ_1, τ_2, and τ_3, whereas just one (minor cycle 1) would contain the activation of τ_4.

The most important consequence of N being large is that the scheduling table becomes large, too. This not only increases the memory footprint of the system, but it also makes the schedule more difficult to visualize and understand. In many cases,

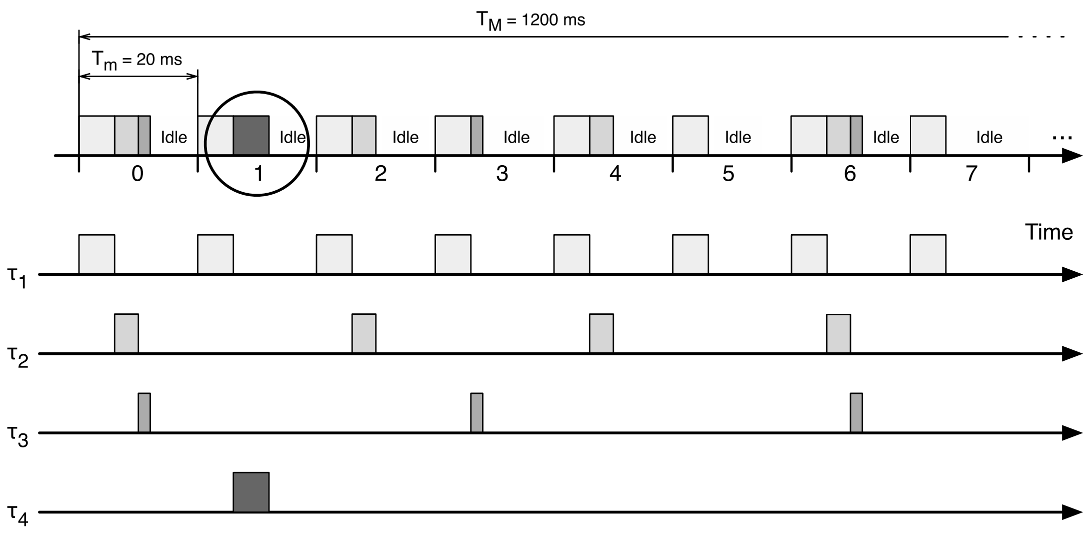

Figure 4.5 Schedule of the task set of Table 4.3, without secondary schedules.

T_M and N can be brought down to a more acceptable value by means of a *secondary schedule*.

In its simplest form, a secondary schedule can be designed by calculating $T_m^{'}$ and $T_M^{'}$ “as if” the task with a large period, τ_4 in our example, had a period equal to an integer sub-multiple of the real one, that is, a period $T_4^{'} = T_4/l$ where l is a positive integer.

The value l must be chosen so that it is as small as possible—leading to a $T_4^{'}$ that is as close as possible to T_4—but giving a major cycle length $T_M^{'}$ that is still equal to the length T_M calculated according to the other tasks to be scheduled. When no suitable l can be found in this way, minor adjustments to the design period T_4 are often sufficient to work around the issue.

In the current example, the value of T_M calculated according to τ_1, τ_2, and τ_3 is

$$T_M = \text{lcm}(20, 40, 60) = 120\,\text{ms}$$

Being $T_4 = 1200\,\text{ms}$, the suitable value of l is $l = 10$ (leading to $T_4^{'} = 120\,\text{ms}$ and $T_M^{'} = 120\,\text{ms}$) because any value lower than this would make $T_M^{'} > T_M$.

Task τ_4 is then placed in the schedule according to the modified period $T_4^{'}$ and its task function is surrounded by a *wrapper*. The wrapper invokes the actual task function once every l invocations and returns immediately in all the other cases. The resulting schedule is shown in Figure 4.6. As can be seen by looking at the figure, the schedule is still compact, even though τ_4 has been successfully accommodated.

On the downside, it should be noted that even though the wrapper returns immediately to the caller without invoking the corresponding task function most of the time (9 times out of 10 in the example), the execution time that must be considered during the cyclic executive design to accommodate the secondary schedule is still equal to C_4, that is, the execution time of τ_4.

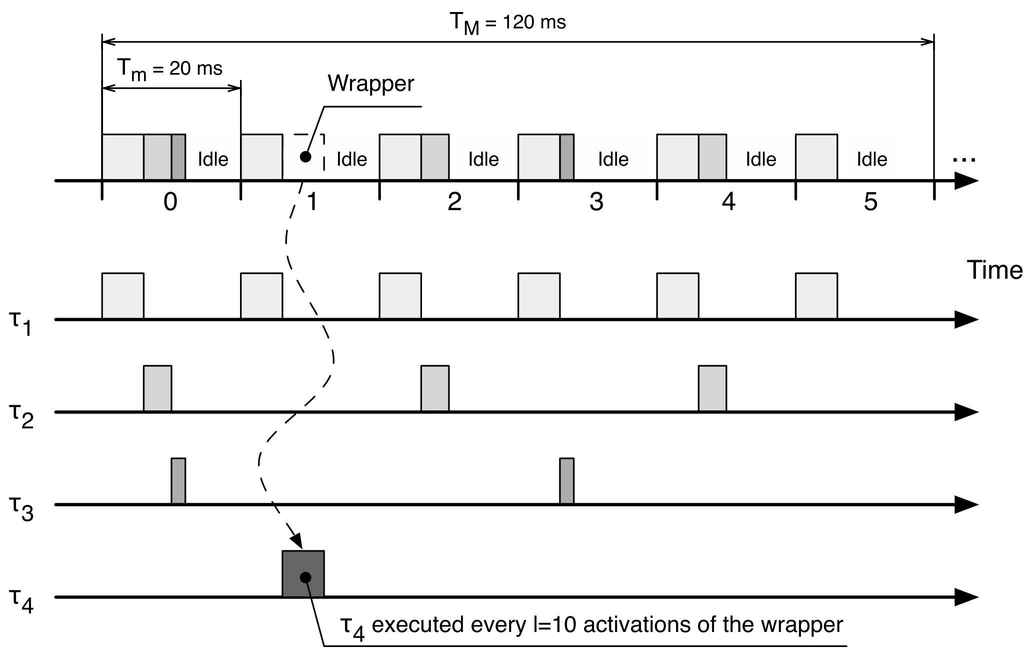

Figure 4.6 Schedule of the task set of Table 4.3, using a secondary schedule.

This is an extremely conservative approach because τ_4 is actually invoked only once every l executions of the wrapper (once out of 10 times in our example), and may make the cyclic executive design more demanding than necessary, or even infeasible.

Moreover, secondary schedules are unable to solve the issue of large T_M in general terms. For instance, when T_M is calculated according to Equation 4.2 and task periods are mutually prime, it will unavoidably be equal to the product of all task periods unless they are adjusted to make them more favorable.

4.4 TASK-BASED SCHEDULING

As seen in Section 4.1, in a cyclic executive, all tasks (or parts of them) are executed in a predefined and fixed order, based on the contents of the scheduling table. The concept of task is somewhat lost at runtime, because there is a single thread of execution across the main program, which implements the schedule, and the various task functions that are called one at a time.

In this section we will discuss instead a different, more sophisticated approach for task execution, namely *task-based scheduling*, in which the abstract concept of task is still present at runtime. In this execution model, tasks become the primary unit of scheduling under the control of an independent software component, namely the real-time *operating system*.

The main differences with respect to a cyclic executive are:

- The operating system itself (or, more precisely, one of its main components known as *scheduler*) is responsible for switching from one task to another. The switching points from one task to another are no longer hard-coded within the code and are chosen autonomously by the scheduler. As a consequence task switch may occur *anywhere* in the tasks, with very few exceptions that will be better discussed in Chapter 5.

- In order to determine the scheduling sequence, the scheduler must follow some criteria, formally specified by means of a *scheduling algorithm*. In general terms, any scheduling algorithm bases its work on certain task characteristics or attributes. For instance, quite intuitively, a scheduling algorithm may base its decision of whether or not to switch from one task to another on their relative importance, or *priority*. For this reason, the concept of task at runtime can no longer be reduced to a mere function containing the task code, as is done in a cyclic executive, but it must include these attributes, too.

The concept of task (also called *sequential process* in more theoretical descriptions) was first introduced in Reference [48] and plays a central role in a task-based system. It provides both an abstraction and a conceptual model of a code module that is being executed. In order to represent a task at runtime, operating systems store some relevant information about it in a data structure, known as *task control block* (TCB). It must contain all the information needed to represent the execution of a sequential program as it evolves over time.

As depicted in Figure 4.7, there are four main components directly or indirectly linked to a TCB:

1. The TCB contains a full copy of the *processor* state. The operating system makes use of this piece of information to switch the processor from one task to another. This is accomplished by saving the processor state of the previous task into its TCB and then restoring the processor state of the next task, an operation known as *context switch*.
   At the same time, the processor state relates the TCB to two other very important elements of the overall task state. Namely, the *program counter* points to the next instruction that the processor will execute, within the task's program code. The *stack pointer* locates the boundary between full and empty elements in the task stack.
   As can be inferred from the above description, the processor state is an essential part of the TCB and is always present, regardless of which kind of operating system is in use. Operating systems may instead differ on the details of *where* the processor state is stored.
   Conceptually, as shown in Figure 4.7, the processor state is *directly* stored in the TCB. Some operating systems follow this approach literally, whereas others store part or all of the processor state elsewhere, for instance in the task stack, and then make it accessible from the TCB through a pointer.
   The second choice is especially convenient when the underlying processor architecture provides hardware assistance to save and restore the processor state

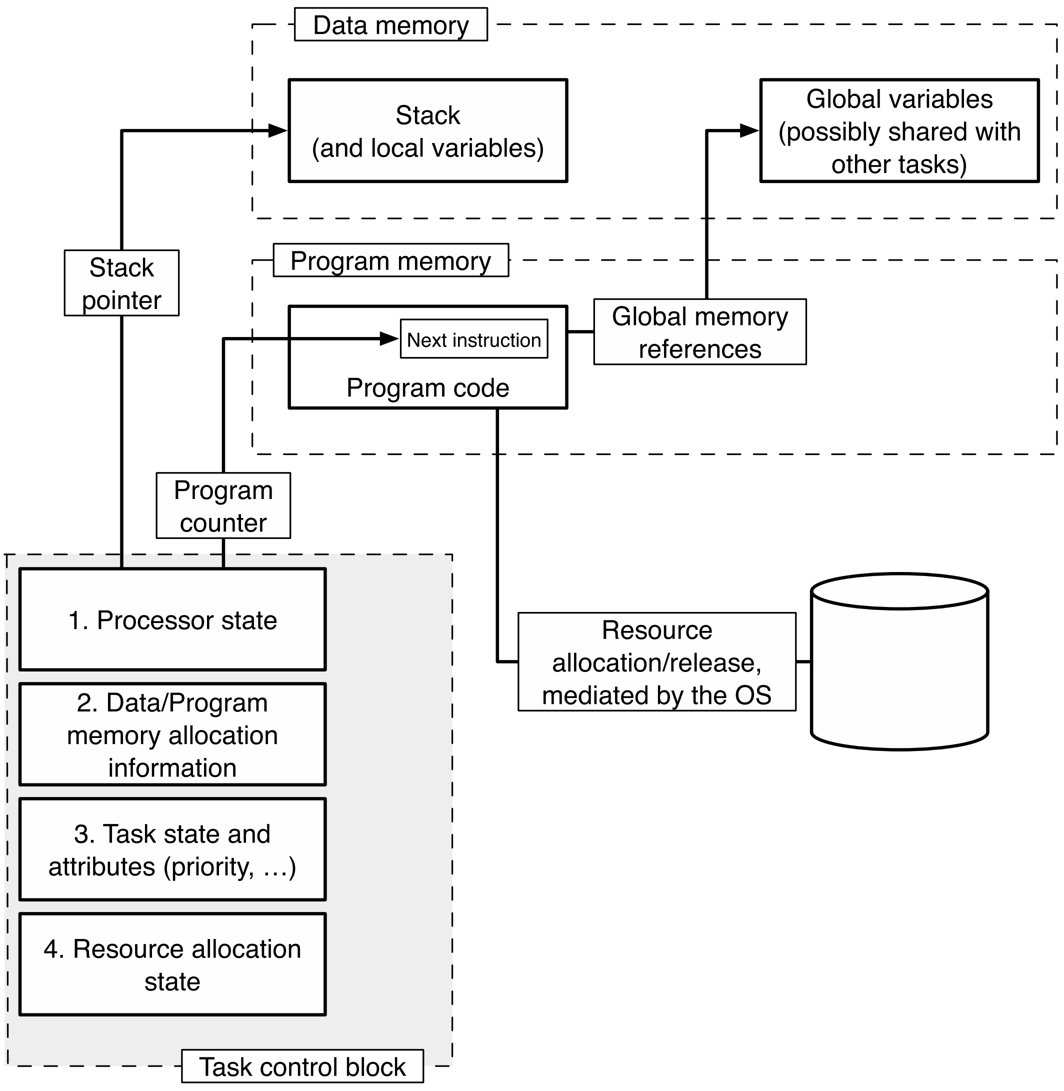

Figure 4.7 Main task control block components.

to/from a predefined, architecture-dependent location, which usually cannot be changed at will in software.

2. The *data and program memory allocation* information held in a TCB keep a record of the memory areas currently assigned to the task. The structure and content of this information heavily depends on the sophistication of the operating system.

   For instance, very simple operating systems that only support a fixed number or statically created tasks may not need to keep this information at all, because data and program memory areas are assigned to the tasks at link time and the assignment never changes over time.

   On the other hand, when the operating system supports the dynamic creation of tasks at runtime, it must at a minimum allocate the task stack dynamically and keep a record of the allocation. Moreover, if the operating system is also capable of loading executable images from a mass storage device on-demand, it should also keep a record of where the program code and global variable areas have been placed in memory.

3. The task *state* and *attributes* are used by the operating system itself to schedule tasks in an orderly way and support inter-task synchronization and communication. An informal introduction to these very important and complex topics will be given in the following, while more detailed information can be found in Chapter 5.
4. When resources allocation and release are mediated by the operating system, the task control block also contains the *resource allocation* state pertaining to the task. The word resource is used here in a very broad sense. It certainly includes all hardware devices connected to the system, but it may also refer to *software* resources.

   Having the operating system work as a mediator between tasks and resources regarding allocation and release is a universal and well-known feature of virtually all general-purpose operating systems. After all, the goal of this kind of operating system is to support the coexistence of multiple application tasks, developed by a multitude of programmers.

   Tasks can be started, stopped, and reconfigured at will by an interactive user, and hence, their characteristics and resource requirements are largely unknown to the operating system (and sometimes even to the user) beforehand. It is therefore not surprising that an important goal of the operating system is to keep resource allocation under tight control.

   In a real-time embedded system, especially small ones, the scenario is very different because the task set to be executed is often fixed and well known in advance. Moreover, the user's behavior usually has little influence on its characteristics because, for instance, users are rarely allowed to start or stop real-time tasks in an uncontrolled way.

   Even the relationship between tasks and resources may be different, leading to a reduced amount of *contention* for resource use among tasks. For instance, in a general-purpose operating system it is very common for application tasks to compete among each other to use a graphics coprocessor, and sharing this resource in an appropriate way is essential.

   On the contrary, in a real-time system devices are often dedicated to a single purpose and can be used only by a single task. For example, an analog to digital converter is usually managed by a cyclic data acquisition task and is of no interest to any other tasks in the system.

   For this reason, in simple real-time operating systems resources are often permanently and implicitly allocated to the corresponding task, so that the operating system itself is not involved in their management.

It is important to note that a correct definition of the information included in a TCB is important not only to thoroughly understand what a task *is*, but also how tasks are *managed* by the operating system. In fact, TCB contents represent the information that the operating system must save and restore when it wants to switch the CPU from one task to another in a transparent way.

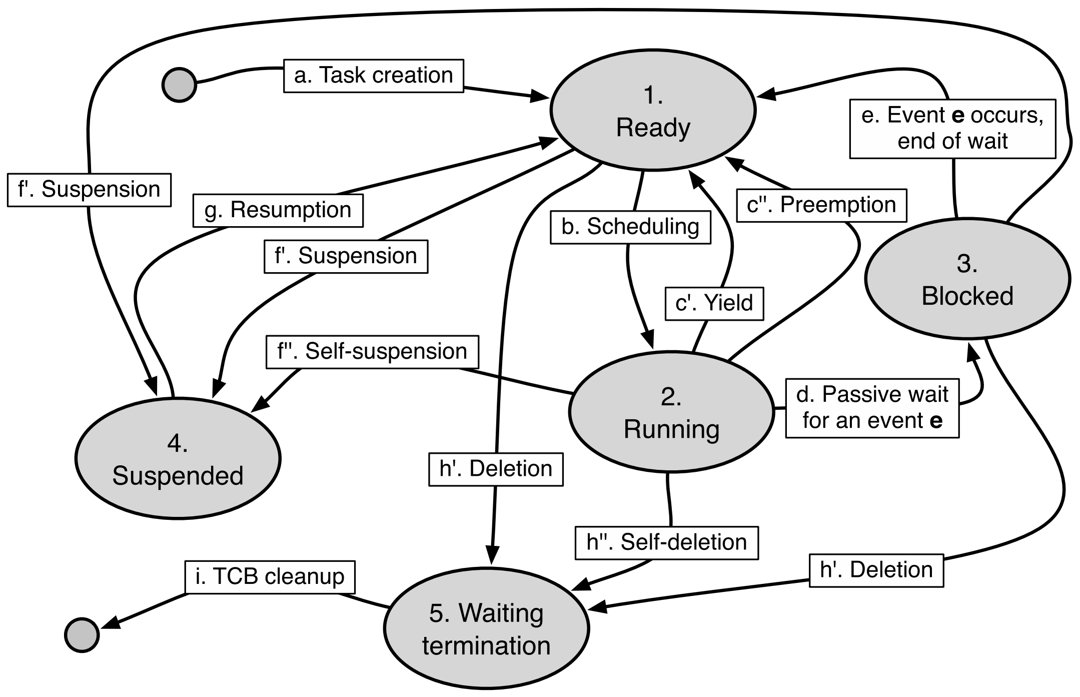

Figure 4.8 Task state diagram in the FREERTOS operating system.

4.5 TASK STATE DIAGRAM

In the previous section, we introduced the concept of task control block (TCB), a data structure managed and maintained by the operating system, which holds all the information needed to represent the execution of a sequential program. A particularly important piece of information held in the TCB is the notion of task *state*.

A commonly used way to describe in a formal way all the possible states a task may be in during its lifespan is to define a directed graph, called *task state diagram* (TSD) or, in the context of general-purpose operating systems, process state diagram (PSD). Although the details of the TSD layout vary from one operating system to another, the most important concepts are common to all of them. In particular:

- TSD *nodes* represent the possible task states.
- Its *arcs* represent transitions from one task state to another.

As a practical example, Figure 4.8 depicts the TSD defined by the FREERTOS operating system [17, 18], used as a case study throughout this book. Referring to the TSD, at any instant a process may be in one of the following states:

1. A task is in the *ready* state when it is eligible for execution but no processors are currently available to execute it, because all of them are busy with other activities. This is a common occurrence because the number of ready tasks usually exceeds the total number of processors (often just one) available in the system. A task does not make any progress when it is ready.

2. A task is *running* when it is actively being executed by a processor, and hence, makes progress. The number of tasks in the *running* state cannot exceed the total number of processors available in the system.
3. Tasks often have to wait for an external event to occur. For example:
   - A periodic task, after completing its activity in the current period, has to wait until the next period begins. In this case, the event is generated by the operating system's timing facility.
   - A task that issues an input–output operation to a device has to wait until the operation is complete. The completion event is usually conveyed from the device to the processor by means of an interrupt request.

   In these cases, tasks move to the *blocked* state. Once there, they no longer compete with other tasks for execution.
4. Tasks in the *suspended* state, like the ones in the *blocked* state, are not eligible for execution. The difference between the two states is that suspended tasks are not waiting for an event. Rather, they went into the *suspended* state either voluntarily or due to the initiative of another task. They can return to the *ready* state only when another task explicitly resumes them.

   Another difference is that there is usually an upper limit on the amount of time tasks are willing to spend in the *blocked* state. When the time limit (called timeout) set by a task expires before the event that the task was waiting for occurs, that task returns to the *ready* state with an error indication. On the contrary, there is no limit on the amount of time a task can stay in the *suspended* state.
5. When a task deletes itself or another task deletes it, it immediately ceases execution but its TCB is not immediately removed from the system. Instead, the task goes into the *waiting termination* state. It stays in that state until the operating system completes the cleanup operation associated with task termination. In FREERTOS, this is done by a system task, called the *idle task*, which is executed when the system is otherwise idle.

There are two kinds of state transition in a TSD.

- A *voluntary* transition is performed under the control of the task that undergoes it, as a consequence of one explicit action it took.
- An *involuntary* transition is not under the control of the task affected by it. Instead, it is the consequence of an action taken by another task, the operating system, or the occurrence of an external event.

Referring back to Figure 4.8, the following transitions are allowed:

a. The *task creation* transition instantiates a new TCB, which describes the task being created. After creation, the new task is not necessarily executed immediately. However, it is eligible for execution and resides in the *ready* state.

b. The operating system is responsible for picking up tasks in the *ready* state for execution and moving them into the *running* state, according to the outcome of its scheduling algorithm, whenever a processor is available for use. This action is usually called task *scheduling*.

c. A running task may voluntarily signal its willingness to relinquish the processor it is being executed on by asking the operating system to reconsider the scheduling decision it previously made. This is done by means of an operating system request known as *yield*, which corresponds to transition c′ in the figure and moves the invoking task from the *running* state to the *ready* state.

   The transition makes the processor previously assigned to the task available for use. This leads the operating system to run its scheduling algorithm and choose a task to run among the ones in the *ready* state. Depending on the scheduling algorithm and the characteristics of the other tasks in the *ready* state, the choice may or may not fall on the task that just yielded.

   Another possibility is that the operating system itself decides to run the scheduling algorithm. Depending on the operating system, this may occur periodically or whenever a task transitions into the *ready* state from some other states for any reason. The second kind of behavior is more common with real-time operating systems because, by intuition, when a task becomes ready for execution, it may be “more important” than one of the running tasks from the point of view of the scheduling algorithm.

   When this is the case, the operating system forcibly moves one of the tasks in the *running* state back into the *ready* state, with an action called *preemption* and depicted as transition c″ in Figure 4.8. Then, it will choose one of the tasks in the *ready* state and move it into the *running* state by means of transition b.

   One main difference between transitions c′ and c″ is therefore that the first one is voluntary, whereas the second one is involuntary.

d. The transition from the *running* to the *blocked* state is always under the control of the affected task. In particular, it is performed when the task invokes one of the operating system synchronization primitives to be discussed in Chapter 5, in order to wait for an event **e**.

   It must be remarked that this kind of wait is very different from what can be obtained by using a polling loop because, in this case, no processor cycles are wasted during the wait.

e. When event **e** eventually occurs, the waiting task is returned to the *ready* state and starts competing again for execution against the other tasks. The task is not returned directly to the *running* state because it may or may not be the most important activity to be performed at the current time. As discussed for yield and preemption, this is a responsibility of the scheduling algorithm and not of the synchronization mechanism.

   Depending on the nature of **e**, the component responsible for waking up the waiting task may be another task (when the wait is due to inter-task synchronization), the operating system timing facility (when the task is waiting for a time-related event), or an interrupt handler (when the task is waiting for an external event, such as an input–output operation).

f. The *suspension* and *self-suspension* transitions, denoted as f′ and f″ in the figure, respectively, bring a task into the *suspended* state. The only difference between the two transitions is that the first one is involuntary, whereas the second one is voluntary.

Table 4.4
A Task Set to Be Scheduled by the Rate Monotonic Algorithm

Task τ_i	Period T_i (ms)	Execution time C_i (ms)
τ_1	20	5
τ_2	40	10
τ_3	60	20

g. When a task is resumed, it unconditionally goes from the *suspended* state into the *ready* state. This happens regardless of which state it was in before being suspended.

   An interesting side effect of this behavior is that, if the task was waiting for an event when the suspend/resume sequence took place, it may resume execution even though the event did not actually take place. In this case, the task being resumed will receive an error indication from the blocking operating system primitive.

h. Tasks permanently cease execution by means of a *deletion* or *self-deletion* transition. These two transitions have identical effects on the affected task. The only difference is that, in the first case, deletion is initiated by another task, whereas in the second case the task voluntarily deletes itself. As discussed above, the TCB of a deleted task is not immediately removed from the system. A side effect of a self-deletion transition is that the operating system's scheduling algorithm will choose a new task to be executed.

i. The *TCB cleanup* transition removes tasks from the *waiting termination* state. After this transition is completed, the affected tasks completely disappear from the system and their TCB may be reused for a new task.

4.6 RACE CONDITIONS IN TASK-BASED SCHEDULING

In Section 4.3, a *race condition* was informally defined as an issue that hinders program correctness when two or more tasks are allowed uncontrolled access to some shared variables or, more generally, a *shared object*. As was remarked there, in a cyclic executive the issue can easily be kept under control because race condition zones appear only as a consequence of task splitting and, even in that case, their location in the schedule is well known in advance.

On the contrary, the adoption of task-based scheduling makes the extent and location of race condition zones hard to predict. This is because the task switching points are now chosen autonomously by the operating system scheduler instead of being hard-coded in the code. Due to this fact, the switching points may even change from one task activation to another.

For example, Figure 4.9 shows how the task set specified in Table 4.4 is scheduled by the Rate Monotonic scheduling algorithm [110]. According to this algorithm,

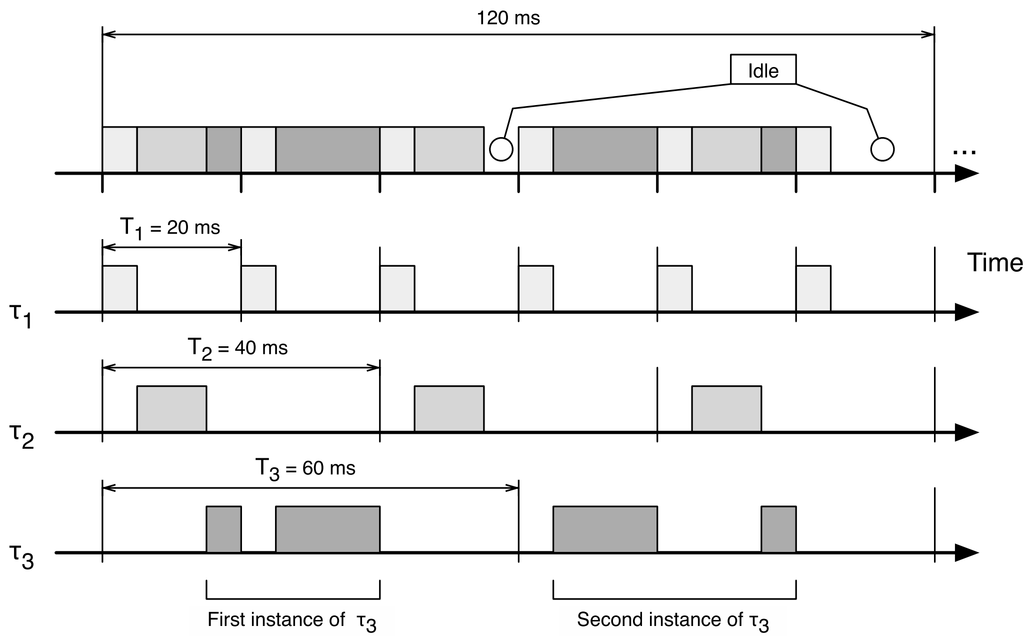

Figure 4.9 Rate Monotonic scheduling of the task set specified in Table 4.4.

which is very commonly used in real-time systems, tasks are assigned a fixed priority that is inversely proportional to their periods. As a consequence, Table 4.4 lists them in decreasing priority order. At any instant, the scheduler grants the processor to the highest-priority task ready for execution.

As can be seen in the figure, even though the task set being considered is extremely simple, the lowest-priority task τ_3 is not only preempted in different places from one instance to another, but by different tasks, too. If, for instance, tasks τ_2 and τ_3 share some variables, no race condition occurs while the first instance of τ_3 is being executed, because the first instance of τ_3 is preempted only by τ_1. On the other hand, the second instance of τ_3 is preempted by both τ_1 and τ_2 and race condition may occur in this case.

As the schedule becomes more complex due to additional tasks, it rapidly becomes infeasible to foresee all possible scenarios. For this reason, a more thorough discussion about race conditions and how to address them in a way that is *independent* from the particular scheduling algorithm in use is given here. Indeed, dealing with race conditions is perhaps the most important application of the task synchronization techniques to be outlined in Chapter 5.

For the sake of simplicity, in this book race conditions will be discussed in rather informal terms, looking mainly at their implication from the concurrent programming point of view. Interested readers should refer, for instance, to the works of Lamport [105, 106] for a more formal description. From the simple example presented above, it is possible to identify two general, *necessary* conditions for a race condition to occur:

1. Two or more tasks must be executed concurrently, leaving open the possibility of a context switch occurring among them. In other words, they must be within a race condition zone, as defined above.
2. These tasks must be actively working on the same shared object when the context switch occurs.

It should also be noted that the conditions outlined above are not yet *sufficient* to cause a race condition because, in any case, the occurrence of a race condition is also a *time-dependent* issue. In fact, even though both necessary conditions are satisfied, the context switch must typically occur at very specific locations in the code to cause trouble. In turn, this usually makes the race condition probability very low and makes it hard to reproduce, analyze, and fix.

The second condition leads to the definition of *critical region* related to a given shared object. This definition is of great importance not only from the theoretical point of view, but also for the practical design of concurrent programs. The fact that, for a race condition to occur, two or more tasks must be actively working on the same shared object leads to classifying the code belonging to a task into two categories.

1. A usually large part of a task's code implements operations that are *internal* to the task itself, and hence, do not make access to any shared data. By definition, all these operations cannot lead to any race condition because the second necessary condition described above is not satisfied.
   From the software development point of view, an important consequence of this observation is that these pieces of code can be safely disregarded when the code is analyzed to reason about and avoid race conditions.
2. Other parts of the task's code indeed make access to shared data. Therefore, those regions of code must be looked at more carefully because they may be responsible for a race condition if the other necessary conditions are met, too.
   For this reason, they are called *critical regions* or *critical sections* with respect to the shared object(s) they are associated with.

Keeping in mind the definition of critical region just given, we can imagine that race conditions on a certain shared object can be avoided by allowing only one task to be within a critical region, pertaining to that object, within a race condition zone. Since, especially in a task-based system, race condition zones may be large and it may be hard to determine their locations in advance, a more general solution consists of enforcing the *mutual exclusion* among all critical regions pertaining to the same shared object *at any time*, without considering race condition zones at all.

Traditionally, the implementation of mutual exclusion among critical regions is based on a *lock-based* synchronization protocol, in which a task that wants to access a shared object, by means of a certain critical region, must first of all *acquire* some sort of *lock* associated with the shared object and possibly wait, if it is not immediately available.

Afterwards, the task is allowed to use the shared object freely. Even though a context switch occurs at this time, it will not cause a race condition because any other task trying to enter a critical region pertaining to the same shared object will be blocked.

When the task has completed its operation on the shared object and the shared object is again in a consistent state, it must *release* the lock. In this way, any other task can acquire it and be able to access the shared object in the future.

In other words, in order to use a lock-based synchronization protocol, critical regions must be "surrounded" by two auxiliary pieces of code, usually called the critical region *entry* and *exit* code, which take care of acquiring and releasing the lock, respectively. For some kinds of task synchronization techniques, better described in Chapter 5, the entry and exit code must be invoked explicitly by the task itself, and hence, the overall structure of the code strongly resembles the one outlined above. In other cases, for instance when using message passing primitives among tasks, critical regions as well as their entry/exit code may be "hidden" within the inter-task communication primitives and be invisible to the programmer, but the concept is still the same.

For the sake of completeness, it must also be noted that mutual exclusion implemented by means of lock-based synchronization is by far the most common, but it is not the only way to solve the race condition problem. It is indeed possible to avoid race conditions *without* any lock, by using *lock-free* or *wait-free* inter-task communication techniques.

Albeit a complete description of lock-free and wait-free communication is outside the scope of this book—interested readers may refer to References [2, 3, 4, 70, 72, 104] for a more detailed introduction to this subject and its application to real-time embedded systems—some of their advantages against lock-based synchronization are worth mentioning anyway. Moreover, a simple, hands-on example of how wait-free communication works—in the context of communication between a device driver and a hardware controller—will be given in Section 8.5.

It has already been mentioned that, during the lock acquisition phase, a task τ_a blocks if another task τ_b is currently within a critical region associated with the same lock. The block takes place regardless of the relative priorities of the tasks. It lasts at least until τ_b leaves the critical region and possibly more, if other tasks are waiting to enter their critical region, too.

As shown in Figure 4.10 if, for any reason, τ_b is delayed (or, even worse, halted due to a malfunction) while it is within its critical region, τ_a and any other tasks willing to enter a critical region associated with the same lock will be blocked and possibly be unable to make any further progress.

Even though τ_b proceeds normally, if the priority of τ_a is higher than the priority of τ_b, the way mutual exclusion is implemented goes against the concept of task priority, because a higher-priority task is forced to wait until a lower-priority task has completed part of its activities. Even though, as will be shown in Chapter 6, it is possible to calculate the worst-case blocking time suffered by higher-priority tasks for this reason and place an upper bound on it, certain classes of applications may not tolerate it in any case.

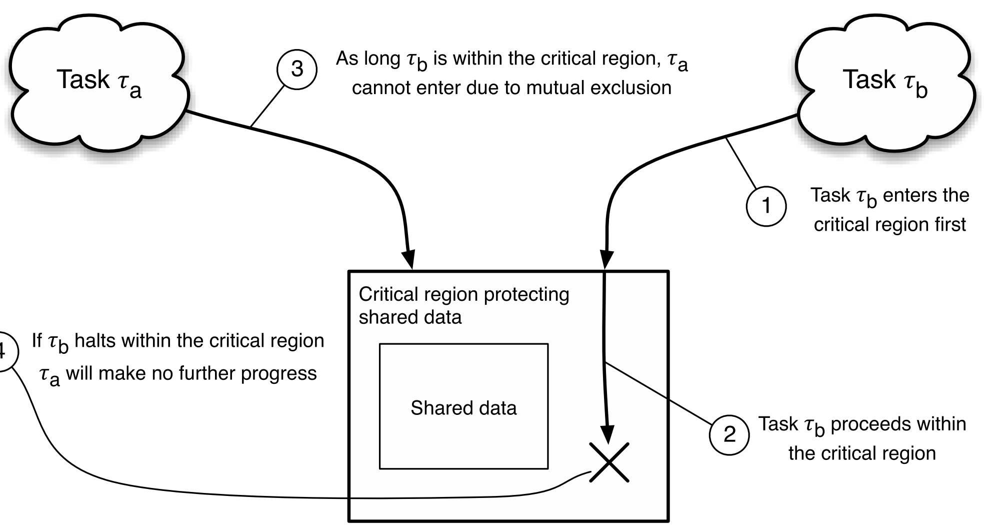
Task τ_a
Task τ_b
3 As long τ_b is within the critical region, τ_a cannot enter due to mutual exclusion
1 Task τ_b enters the critical region first
Critical region protecting shared data
Shared data
4 If τ_b halts within the critical region τ_a will make no further progress
2 Task τ_b proceeds within the critical region

Figure 4.10 Shortcoming of lock-based synchronization when a task halts.

Lock-free and wait-free communication do not require mutual exclusion, and hence, they completely solve the blocking time issue. With lock-free communication it is indeed possible that an operation on a shared object fails, due to a concurrent object access from another task. However, the probability of such a failure is usually very low and it is possible to make the overall failure probability negligible by retrying the operation a small number of times.

The main drawback of lock-free and wait-free communication is that their inner workings are considerably more complex than any traditional inter-task communication and synchronization mechanisms. However, the recent development and widespread availability of open-source libraries containing a collection of lock-free data structures, for example, the Concurrent Data Structures library (libcds) [100] will likely bring what is today considered an advanced topic in concurrent programming into the mainstream.

4.7 SUMMARY

This chapter provided an overview of two main execution models generally adopted in embedded system design and development. The first one, described in Sections 4.1 and 4.2, has the clear advantages of being intuitive for programmers and efficient for what concerns execution.

On the other hand, it also has several shortcomings, outlined in Section 4.3, which hinder its applicability as system size and complexity grow. Historically, this consideration led to the introduction of more sophisticated execution models, in which the concept of task is preserved and plays a central role at runtime.

The basic concepts of task-based scheduling were illustrated in Sections 4.4, while Section 4.5 contains more information about the task state diagram, a key data structure that represents the evolution of a task through its lifetime in the system.

However, task-based scheduling also brings out a whole new class of programming complications, which arose only in rare cases with the cyclic executive model related to race conditions in shared data access.

In this chapter, Section 4.6 introduced the reader to the general problem of race conditions in task-based scheduling and how to address it by correctly identifying critical regions in the code and managing them appropriately by means of mutual exclusion. The same section also provided some pointers to alternate ways of solving the problem by means of lock- and wait-free inter-task communication techniques.

Staying with traditional techniques, a proper implementation of mutual exclusion requires an adequate knowledge of a programming technique known as concurrent programming, which will be the subject of the next chapter.