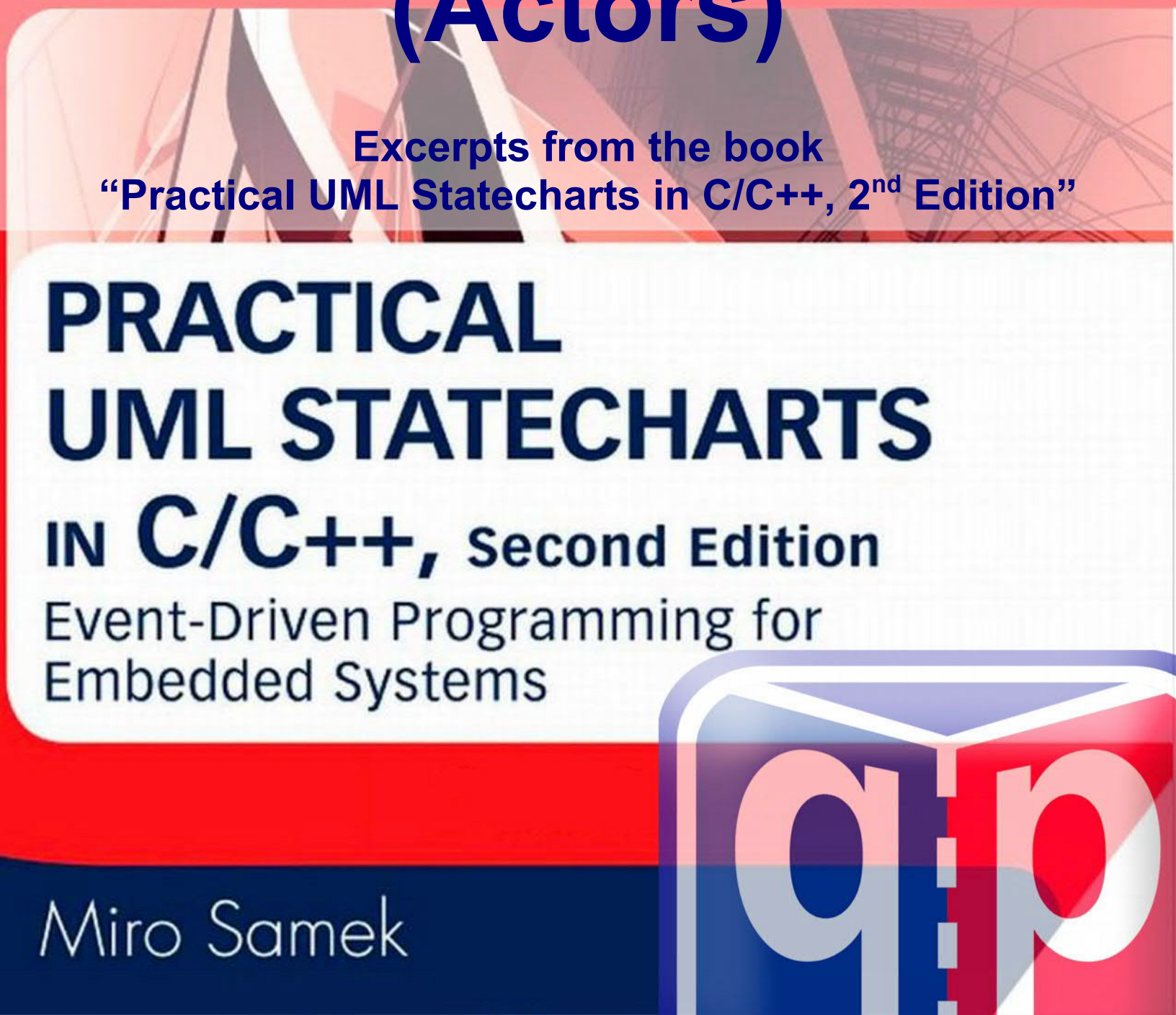
Quantum® LeaPs

innovating embedded systems

Active Objects (Actors)

Excerpts from the book
"Practical UML Statecharts in C/C++, 2nd Edition"



6.3 Active Object Computing Model

The *active object computing model* addresses most problems of the traditional event-driven architecture, retaining its good characteristics. As described in the sidebar "From Actors to Active Objects," the term *active object* comes from the UML and denotes "an object having its own thread of control" [OMG 07]. The essential idea of this model is to use multiple event-driven systems in a multitasking environment.

FROM ACTORS TO ACTIVE OBJECTS

The concept of autonomous software objects communicating by message passing dates back to the late 1970s, when Carl Hewitt and colleagues [Hewitt 73] developed a notion of an *actor*. In the 1980s, actors were all the rage within the distributed artificial intelligence community, much as agents are today. In the 1990s, methodologies like ROOM [Selic+ 94] adapted actors for real-time computing. More recently, the UML specification has introduced the concept of an *active object* that is essentially synonymous with the notion of a ROOM actor [OMG 07].

In the UML specification, an active object is "an object having its own thread of control" [OMG 07] that processes events in a run-to-completion fashion and that communicates with other active objects by asynchronously exchanging events. The UML specification further proposes the UML variant of state machines with which to model the behavior of event-driven active objects.

Active objects are most commonly implemented with *real-time frameworks*. Such frameworks have been in extensive use for many years and have proven themselves in a very wide range of real-time embedded (RTE) applications. Today, virtually every design automation tool that supports code synthesis for RTE systems incorporates a variant of a real-time framework. For instance, Real-time Object-Oriented Modeling (ROOM) calls its framework the "ROOM virtual machine" [Selic+ 94]. The VisualSTATE tool from IAR Systems calls it a "VisualSTATE engine" [IAR 00]. The UML-compliant design automation tool Rhapsody from Telelogic calls it "Object Execution Framework (OXF)" [Douglass 99].

Figure 6.5 shows a minimal active object system. The application consists of multiple active objects, each encapsulating a thread of control (event loop), a private event queue, and a state machine.

$$\text{Active object} = (\text{thread of control} + \text{event queue} + \text{state machine}).$$

The active object's event loop, shown in Figure 6.5(B), is a simplified version of the event loop from Figure 6.4. The simplified loop gets rid of the dispatcher and directly

extracts events from the event queue, which efficiently blocks the loop as long as the queue is empty. For every event obtained from the queue, the event loop calls the `dispatch()` function associated with the active object. The `dispatch()` function performs both the dispatching and processing of the event, similarly to the event-handler functions in the traditional event-driven architecture.

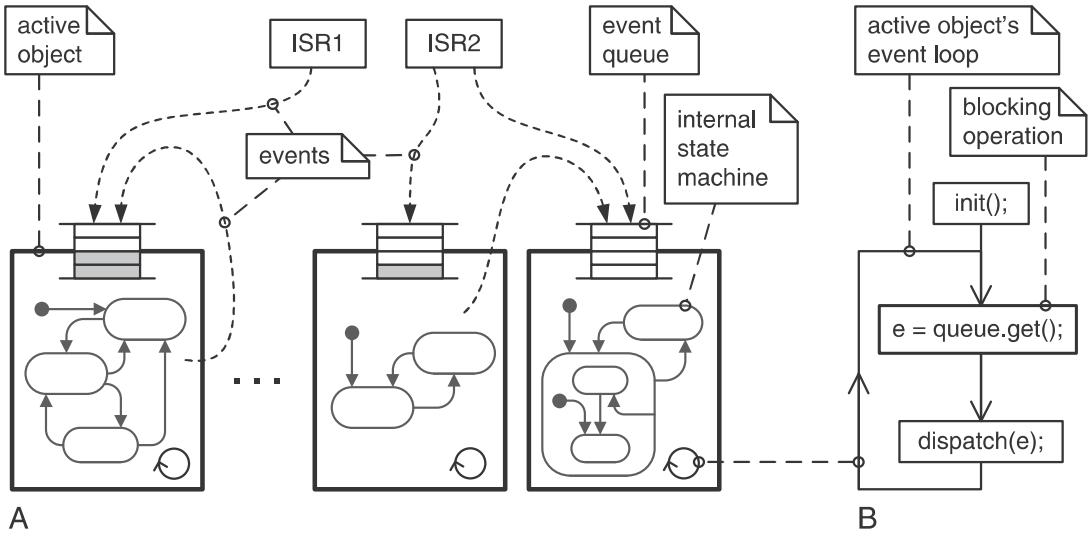


Figure 6.5: Active-object system (A) and active object's event loop (B).

6.3.1 System Structure

The event queues, event loops, and the event processor for state machines are all generic and as such are part of a generic *real-time framework*. The application consists of the specific state machine implementations, which the framework invokes indirectly through the `dispatch()`[5] state machine operation.

Figure 6.6 shows the relationship between the application, the real-time framework, and the real-time kernel or RTOS. I use the QF real-time framework as an example, but the general structure is typical for any other framework of this type. The design is layered, with an RTOS at the bottom providing the foundation for multitasking and basic services like message queues and deterministic memory partitions for storing

[5] The `dispatch()` operation is understood here generically and denotes any state machine implementation method, such as any of the techniques described in Chapters 3 or 4.

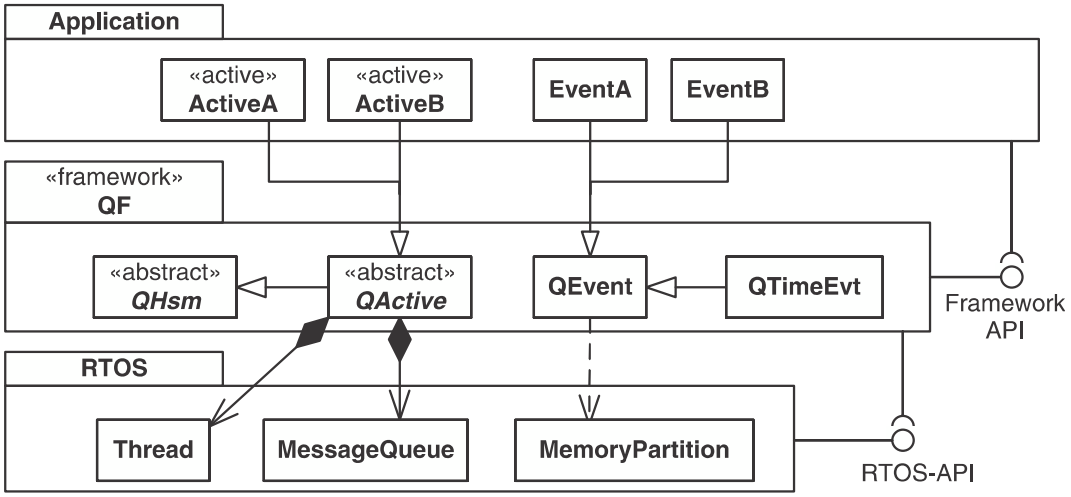


Figure 6.6: Package diagram illustrating the relationship among the real-time framework, the kernel/RTOS, and the application.

events. Based on these services, the QF real-time framework supplies the `QActive` class for derivation of active objects. The `QActive` class in turn derives from the `QHsm` base class, which means that active objects are state machines and inherit the `dispatch()` operation defined in the `QHsm` base class (see Chapter 4). Additionally, `QActive` contains a thread of execution and an event queue, typically based on the message queue of the underlying RTOS. An application extends the real-time framework by deriving active objects from the `QActive` base class and deriving events with parameters from the `QEvent` class.

NOTE

Most frameworks rely heavily on the object-oriented concepts of classes and inheritance as the key technique for extending and customizing the framework. If you program in C and the concepts are new to you, refer to the sidebar "Single Inheritance in C" in Chapter 1. In Chapter 7 you'll see that the QF real-time framework and the applications derived from it can be quite naturally implemented in standard, portable C.

The application uses the QF communication and timing services through the framework API (indicated by the ball-and-socket UML notation); however, the application typically should not need to directly access the RTOS API. Thus, a real-time framework can serve as an *RTOS abstraction layer*. The framework effectively insulates

applications from the underlying RTOS. Changing the RTOS on which the framework is built requires porting the framework but does not affect applications. I'll demonstrate this aspect in Chapter 8, where I discuss porting QF.

6.3.2 Asynchronous Communication

As shown in Figure 6.5(A), active objects receive events exclusively through their event queues. All events are delivered *asynchronously*, meaning that an event producer merely posts an event to the event queue of the recipient active object but doesn't wait in line for the actual processing of the event.

The system makes no distinction between external events generated from interrupts and internal events originating from active objects. As shown in Figure 6.5(A), an active object can post events to any other active object, including to self. All events are treated uniformly, regardless of their origin.

6.3.3 Run-to-Completion

Each active object processes events in run-to-completion (RTC) fashion, which is guaranteed by the structure of the active object's event loop. As shown in Figure 6.5(B), the `dispatch()` operation must necessarily complete and return to the event loop before the next event from the queue can be extracted. RTC event processing is the essential requirement for proper execution of state machines.

In the case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of RTC from the concept of thread preemption [OMG 07]. In particular, RTC does not mean that the active object thread has to monopolize the CPU until the RTC step is complete. Under a preemptive multitasking kernel, an RTC step can be preempted by another thread executing on the same CPU. This is determined by the scheduling policy of the underlying multitasking kernel, not by the active object computing model. When the suspended thread is assigned CPU time again, it resumes its event processing from the point of preemption and, eventually, completes its event processing. As long as the preempting and the preempted threads don't share any resources, there are no concurrency hazards.

6.3.4 Encapsulation

Perhaps the most important characteristic of active objects, from which active *objects* actually derive their name, is their strict *encapsulation*. Encapsulation means that

active objects don't share data or any other resources. Figure 6.5(A) illustrates this aspect by a thick, opaque encapsulation shell around each active object and by showing the internal state machines in gray, since they are really not supposed to be visible from the outside.

As described in the previous section, no sharing of any resources (encapsulation) allows active objects to freely preempt each other without the risk of corrupting memory or other resources. The only allowed means of communication with the external world and among active objects is asynchronous event exchange. The event exchange and queuing are controlled entirely by the real-time framework, perhaps with the help of the underlying multitasking kernel, and are guaranteed to be *thread-safe*.

Even though encapsulation has been traditionally associated with object-oriented programming (OOP), it actually predates OOP and does not require object-oriented languages or any fancy tools. Encapsulation is not an abstract, theoretical concept but simply a disciplined way of designing systems based on the concept of *information hiding*. Experienced software developers have learned to be extremely wary of shared (global) data and various mutual exclusion mechanisms (such as semaphores). Instead, they bind the data to the tasks and allow the tasks to communicate only via message passing. For example, the embedded systems veteran, Jack Ganssle, offers the following advice [Ganssle 98].

> "Novice users all too often miss the importance of the sophisticated messaging mechanisms that are a standard part of all commercial operating systems. Queues and mailboxes let tasks communicate safely... the operating system's communications resources let you cleanly pass a message without fear of its corruption by other tasks. Properly implemented code lets you generate the real-time analogy of object-oriented programming's (OOP) first tenet: encapsulation. Keep all of the task's data local, bound to the code itself and hidden from the rest of the system."
>
> —Jack Ganssle

Although it is certainly true that the operating system mechanisms, such as message queues, critical sections, semaphores, or condition variables, can serve in the construction of a real-time framework, application programmers do not need to directly use these often troublesome mechanisms. Encapsulation lets programmers implement the internal structure of active objects without concern for multitasking. For example, application programmers don't need to know how to correctly use a semaphore or even know what it is. Still, as long as active objects are encapsulated, an active object system can execute safely, taking full advantage of all the benefits of multitasking, such as optimal responsiveness to events and good CPU utilization.

In Chapter 9 I will show you how to organize the application source code so that the internal structure of active objects is hidden and inaccessible to the rest of the application.

6.3.5 Support for State Machines

Event-driven systems are in general more difficult to implement with standard languages, such as C or C++, than procedure-driven systems [Rumbaugh+ 91]. The main difficulty comes from the fact that an event-driven application must return control after handling each event, so the code is fragmented and expected sequences of events aren't readily visible.

For example, Figure 6.7(A) shows a snippet of a sequential pseudocode, whereas panel (B) shows the corresponding flowchart. The boldface statements in the code and

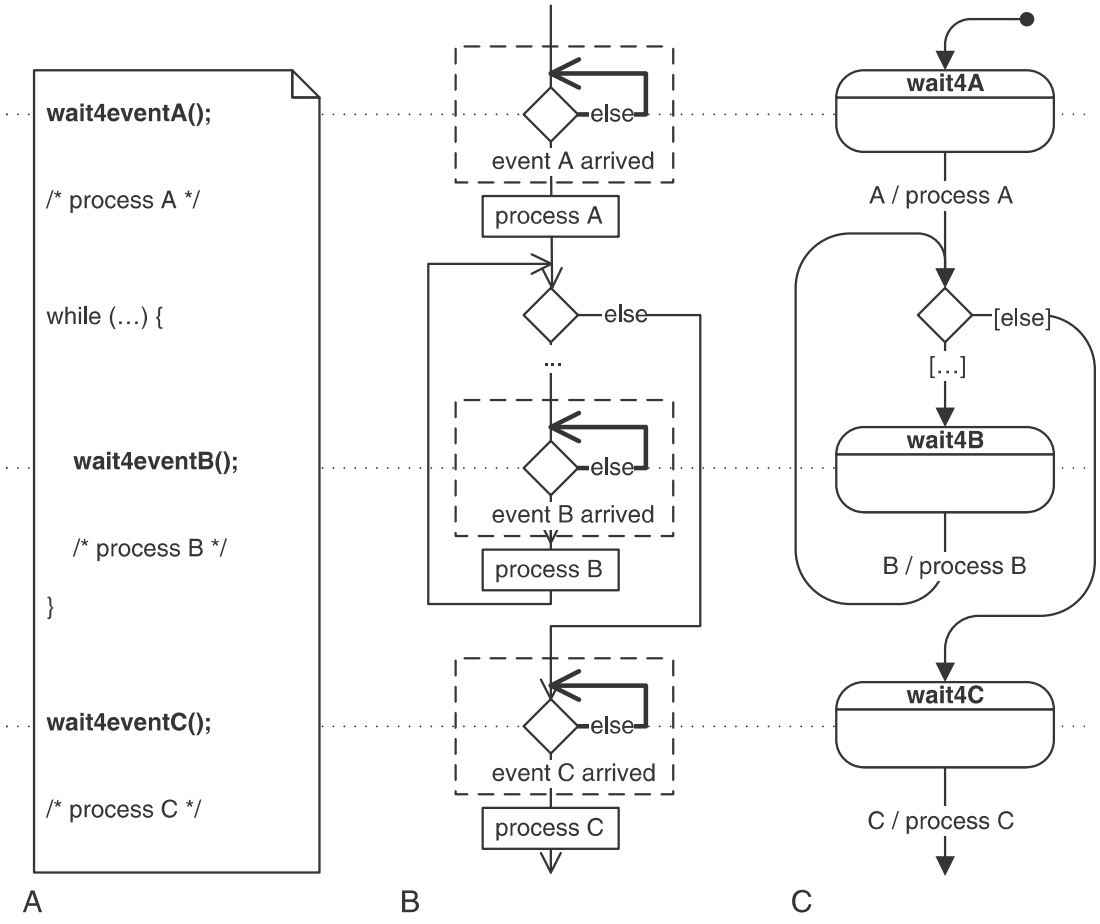


Figure 6.7: Sequential pseudocode (A), flowchart (B), and state machine (C).

heavy lines in the flowchart represent waiting for events (either polling or efficient blocking). Both the sequential code and the flowchart show the expected sequence of events (A, B...B, C) very clearly. Moreover, the sequential processing allows passing data from one processing stage to the next in temporary stack variables. Traditional programming languages and traditional multitasking kernels strongly support this style of programming that relies heavily on stack-intensive nested function calls and sophisticated flow of control (e.g., loops).

In contrast, the traditional event-driven code representing essentially the same behavior consists of three event-handler functions `onA()`, `onB()`, and `onC()`, and it is not at all clear that the expected sequence of calls should be `onA()`, `onB()...onB()`, `onC()`. This information is hidden inside the event-hander functions. Moreover, the functions must use static variables to pass data from one function to the next, because the stack context disappears when each function returns to the event loop. This programming style is harder to implement with standard languages because you get virtually no support for handling the execution context stored in static variables.

And this is where state machines beautifully complement the traditional programming languages. State machines are exactly designed to represent the execution context (state and extended-state variables) in static data. As you can see in Figure 6.7(C), the state machine clearly shows the expected event sequence, so this program structure becomes visible again. But unlike the sequential code, a state machine does not rely on the stack and the program counter to preserve the context from one state to the next. State machines are inherently event-driven.

NOTE

You can think of state machines, and specifically of the hierarchical event processor implementation described in Chapter 4, as an essential *extension* of the C and C++ programming languages to better support event-driven programming.

As opposed to the traditional event-driven architecture, the active object computing model is compatible with state machines. The active object event loop specifically eliminates the event dispatcher (Figure 6.5(B)) because demultiplexing events based on the event signal is not a generic operation but instead always depends on the internal *state* of an active object. Therefore, event dispatching must be left to the specific active object's state machine.

6.3.6 Traditional Preemptive Kernel/RTOS

In the most common implementations of the active object computing model, active objects map to threads of a traditional preemptive RTOS or OS. For example, the real-time framework inside the Telelogic Rhapsody design automation tool provides standard bindings to VxWorks, QNX, and Linux, to name a few [Telelogic 07]. In this standard configuration the active object computing model can take full advantage of the underlying RTOS capabilities. In particular, if the kernel is preemptive, the active object system achieves exactly the same optimal task-level response as traditional tasks.

Consider how the preemptive kernel scenario depicted in Figure 6.3(B) plays out in an active object system. The scenario begins with a low-priority active object executing its RTC step and a high-priority active object efficiently blocked on its empty event queue.

NOTE

The priority of an active object is the priority of its execution thread.

At point (2b) in Figure 6.3(B), an interrupt preempts the low-priority active object. The ISR executes and, among other things, posts an event to the high-priority active object (3b). The preemptive kernel called upon the exit from the ISR (4b) detects that the high-priority active object is ready to run, so it switches context to that active object (5b). The interrupt returns to the high-priority active object that extracts the just-posted event from its queue and processes the event to completion (6b). When the high-priority active object blocks again on its event queue, the kernel notices that the low-priority active object is still preempted. The kernel switches context to the low-priority active object (7b) and lets it run to completion.

Note that even though the high-priority active object preempted the low-priority one in the middle of the event processing, the RTC principle hasn't been violated. The low-priority active object resumed its RTC step exactly at the point of preemption and completed it eventually, before engaging in processing another event.

NOTE

In Chapter 8, I show how to adapt the QF real-time framework to work with a typical preemptive kernel (µC/OS-II) as well as a standard POSIX operating system (e.g., Linux, QNX, Solaris).

6.3.7 Cooperative Vanilla Kernel

The active object computing model can also work with nonpreemptive kernels. In fact, one particular cooperative kernel matches the active object computing model exceptionally well and can be implemented in an absolutely portable manner. For lack of a better name, I will call this kernel *plain vanilla* or just *vanilla*. I explain first how the vanilla kernel works and later I compare its execution profile with the profile of a traditional nonpreemptive kernel from Figure 6.3(A). Chapter 7 describes the QF implementation of the vanilla kernel.

NOTE

The vanilla kernel is so simple that many commercial real-time frameworks don't even call it a kernel. Instead this configuration is simply referred to as *without an RTOS*.[6] However, if you want to understand what it means to execute active objects "without an RTOS" and what execution profile you can expect in this case, you need to realize that a simple cooperative vanilla kernel is indeed involved.

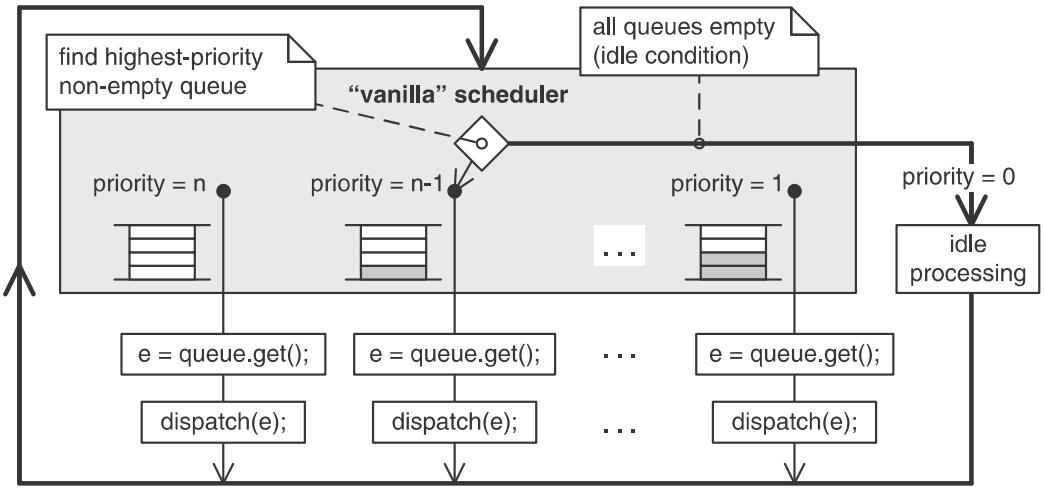


Figure 6.8: Active object system executing under the cooperative vanilla kernel.

[6] For example, the Interrupt Driven Framework (IDF) inside the Telelogic Rhapsody design automation tool executes "without an RTOS."

Figure 6.8 shows the architecture of the simple cooperative vanilla kernel. The most important element of the design is the *scheduler*, which is the part of a kernel responsible for determining which task to run next. The vanilla scheduler operates in a single loop. The scheduler constantly monitors all event queues of the active objects. Each event queue is assigned a unique priority, which is the priority of the active object that owns this queue. The scheduler always picks the highest-priority not-empty queue.

NOTE

The vanilla scheduler uses the event queues of active objects as *priority queues* and thus embodies the standard priority queue algorithm [Cormen+ 01]. Chapter 7 shows how the QF real-time framework implements the vanilla scheduler with a bitmask and a lookup table.

After finding the queue, the vanilla kernel extracts the event from the queue and dispatches it to the active object that owns this queue. Note that the queue `get()` operation cannot block because at this point the queue is guaranteed to be not empty. Of course, the vanilla kernel applies all the necessary safeguards to protect the internal state of the scheduler and the event queues from corruption by asynchronous interrupts, which can post events to the queues at any time.

The `dispatch()` operation always runs to completion and returns to the main loop. The scheduler takes over and the cycle repeats. As usual in event-driven systems, the main event loop and the event queues are all part of the vanilla kernel or the framework. The application code is not supposed to poll or block.

The vanilla scheduler very easily detects the condition when all event queues are empty. This situation is called the *idle condition* of the system. In this case, the scheduler performs idle processing, which can be customized by the application.

NOTE

In an embedded system, the idle processing is the ideal place to put the CPU into a low-power sleep mode. The power-saving hardware wakes up the CPU upon an interrupt, which is exactly right because at this point only an interrupt can provide new event(s) to the system.

Now consider how the scenario depicted in Figure 6.3(A) plays out under the vanilla kernel. The scenario begins with a low-priority active object executing its RTC step (`dispatch()` function) and a high-priority active object having its event queue

empty. At point (2a) an interrupt preempts the low-priority active object. The ISR executes and, among other things, posts an event to the high-priority active object (3a). The interrupt returns and resumes the originally preempted low-priority active object (4a). The low-priority object runs to completion and returns to the main loop. At this point, the vanilla scheduler has a chance to run and picks the highest-priority nonempty queue, which is the queue of the high-priority active object (6a). The vanilla kernel calls the `dispatch()` function of the high-priority active object, which runs to completion.

As you can see, the task-level response of the vanilla kernel is exactly the same as any other nonpreemptive kernel. Even so, the vanilla kernel achieves this responsiveness without per-task stacks or complex context switching. The active objects naturally collaborate to share the CPU and implicitly yield to each other at the end of every RTC step. The implementation is completely portable and suitable for low-end embedded systems.

Because typically the RTC steps are quite short, the kernel can often achieve adequate task-level response even on a low-end CPU. Due to the simplicity, portability, and minimal overhead, I highly recommend the vanilla kernel as your first choice. Only if this type of kernel cannot meet your timing requirements should you move up to a preemptive kernel.

NOTE

The vanilla kernel also permits executing multiple active objects inside a single thread of a bigger multitasking system. In this case, the vanilla scheduler should efficiently block when all event queues are empty instead of wasting CPU cycles for polling the event queues. Posting an event to any of the active object queues should unblock the kernel. Of course, this requires integrating the vanilla kernel with the underlying multitasking system.

6.3.8 Preemptive RTC Kernel

Finally, if your task-level response requirements mandate a preemptive kernel, you can consider a super-simple, *run-to-completion preemptive kernel* that matches perfectly the active object computing model [Samek+ 06]. A preemptive RTC kernel implements in software exactly the same deterministic scheduling policy for tasks as most prioritized interrupt controllers implement in hardware for interrupts.

Prioritized interrupt controllers, such as the venerable Intel 8259A, the Motorola 68K and derivatives, the interrupt controllers in ARM-based MCUs by various vendors, the NVIC in the ARMv7 architecture (e.g., Cortex-M3), the M16C from Renesas, and many others allow prioritized nesting of interrupts on a *single stack*.

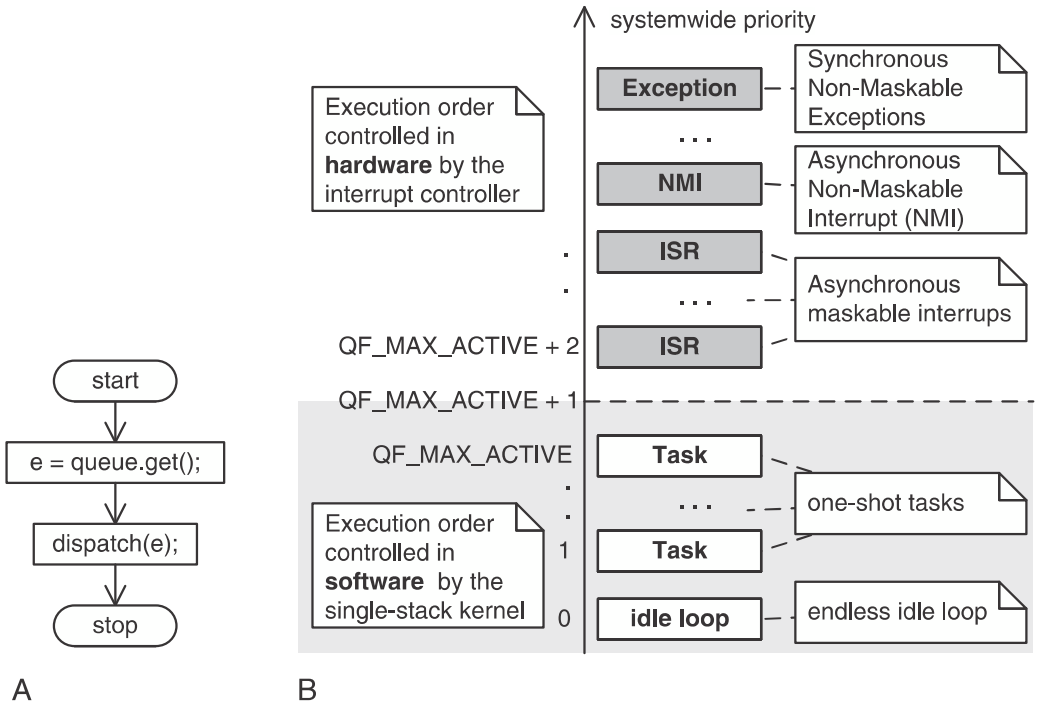


Figure 6.9: Systemwide priority of a single-stack kernel (A) and task structure (B).

In an RTC kernel tasks and interrupts are nearly symmetrical: both tasks and ISRs are one-shot, RTC functions (Figure 6.9(A)). In fact, an RTC kernel views interrupts very much like tasks of "super-high" priority, except that interrupts are prioritized in hardware by the interrupt controller, whereas tasks are prioritized in software by the kernel (Figure 6.9(B)).

NOTE
In all traditional kernels, tasks are generally structured as endless loops. An RTC kernel breaks with this arrangement entirely. Under an RTC kernel, tasks are one-shot functions that run to completion and return, very much like ISRs managed by a prioritized interrupt controller.

By requiring that all tasks run to completion and enforcing fixed-priority scheduling, an RTC kernel can use the machine's natural stack protocol. Whenever a task is preempted by a higher-priority task (perhaps as a result of the currently running task posting an event to a higher-priority task), the RTC kernel uses a regular C-function call to build the higher-priority task context on top of the preempted-task stack context.

Whenever an interrupt preempts a task, the kernel uses the already established interrupt stack frame on top of which to build the higher-priority task context, again using a regular C-function call. This simple form of context management is adequate because every task, just like every ISR, runs to completion. Because the preempting task must also run to completion, the lower-priority stack context will never be needed until the preempting task (and any higher-priority task that might preempt it) has completed—at which time the preempted task will naturally be at the top of the stack, ready to be resumed. This simple mechanism works for exactly the same reason that a prioritized hardware-interrupt system works [Samek+ 06].

NOTE

Such a close match between the active object computing model and prioritized, nested interrupt handling implemented directly in hardware suggests that active objects are in fact quite a basic concept. In particular, the RTC processing style and no need for blocking in active objects map better to actual processor architectures and incur less overhead than traditional blocking kernels. In this respect, traditional blocking tasks must be viewed as a higher-level, more heavyweight concept than active objects.

One obvious consequence of the stack-use policy, and the most severe limitation of an RTC kernel, is that *tasks cannot block*. The kernel cannot leave a high-priority task context on the stack and at the same time resume a lower-priority task. The lower-priority task context simply won't be accessible on top of the stack unless the higher-priority task completes. But as I keep repeating *ad nauseam* throughout this book, event-driven programming is all about writing nonblocking code. Event-driven active objects don't have a need for blocking.

In exchange for not being able to block, an RTC kernel offers many advantages over traditional blocking kernels. By nesting all task contexts in a single stack, the RTC kernel can be super-simple because it doesn't need to manage multiple stacks and all their associated bookkeeping. The result is not just significantly less RAM required for the stacks and task control blocks but a faster context switch and, overall, less CPU overhead. At the same time, an RTC kernel is as deterministic and responsive as any other fully preemptive priority-based kernel. In Chapter 10, I describe an RTC kernel called QK, which is part of the QP platform. QK is a tiny preemptive, priority-based RTC kernel specifically designed to provide preemptive multitasking support to the QF real-time framework.

If you are using a traditional preemptive kernel or RTOS for executing event-driven systems, chances are that you're overpaying in terms of CPU and memory overhead. You can achieve the same execution profile and determinism with a much simpler RTC kernel. The only real reason for using a traditional RTOS is compatibility with existing software. For example, traditional device drivers, communication stacks (such as TCP/IP, USB, CAN, etc.), and other legacy subsystems are often written with the blocking paradigm. A traditional blocking RTOS can support both active object and traditional blocking code, which the RTOS executes outside the real-time framework.

NOTE

Creating entirely event-driven, nonblocking device drivers and communication stacks is certainly possible but requires standardizing on specific event-queuing and event-passing mechanisms rather than blocking calls. Such widespread standardization simply hasn't occurred yet in the industry.

6.4 Event Delivery Mechanisms

One of the main responsibilities of every real-time framework is to efficiently deliver events from producers to consumers. The event delivery is generally asynchronous, meaning that the producers of events only insert them into event queues but do not wait for the actual processing of the events.

In addition, any part of the system can usually produce events, not necessarily only the active objects. For example, ISRs, device drivers, or legacy code running outside the framework can produce events. On the other hand, only active objects can consume events, because only active objects have event queues.

NOTE

A framework can also provide "raw" thread-safe event queues without active objects behind them. Such "raw" thread-safe queues can consume events as well, but they never block and are intended to deliver events to ISRs, that is, provide a communication mechanism from the task level to the ISR level.