"Fly 'n' Shoot"
Game Example

Excerpts from the book

PRACTICAL UML STATECHARTS IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems

Miro Samek
It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

—Franklin D. Roosevelt

This chapter presents an example project implemented entirely with UML state machines and the event-driven paradigm. The example application is an interactive “Fly ‘n’ Shoot”-type game, which I decided to include early in the book so that you can start playing (literally) with the code as soon as possible. My aim in this chapter is to show the essential elements of the method in a real, nontrivial program, but without getting bogged down in details, rules, and exceptions. At this point, I am not trying to be complete or even precise, although this example as well as all other examples in the book is meant to show a good design and the recommended coding style. I don’t assume that you know much about UML state machines, UML notation, or event-driven programming. I will either briefly introduce the concepts, as needed, or refer you to the later chapters of the book for more details.

The example “Fly ‘n’ Shoot” game is based on the Quickstart application provided in source code with the Stellaris EV-LM3S811 evaluation kit from Luminary Micro [Luminary 06]. I was trying to make the “Fly ‘n’ Shoot” example behave quite similarly to the original Luminary Micro Quickstart application so that you can directly compare the event-driven approach with the traditional solution to essentially the same problem specification.
1.1 Installing the Accompanying Code

The companion Website to this book at www.quantum-leaps.com/psicc2 contains the self-extracting archive with the complete source code of the QP event-driven platform and all executable examples described in this book; as well as documentation, development tools, resources, and more. You can uncompress the archive into any directory. The installation directory you choose will be referred henceforth as the QP Root Directory `<qp>`.

**NOTE**

Although in the text I mostly concentrate on the C implementation, the accompanying Website also contains the equivalent C++ version of virtually every element available in C. The C++ code is organized in exactly the same directory tree as the corresponding C code, except you need to look in the `<qp>\qpcpp\...` directory branch.

Specifically to the “Fly ‘n’ Shoot” example, the companion code contains two versions\(^1\) of the game. I provide a DOS version for the standard Windows-based PC (see Figure 1.1) so that you don’t need any special embedded board to play the game and experiment with the code.

**NOTE**

I’ve chosen the legacy 16-bit DOS platform because it allows programming a standard PC at the bare-metal level. Without leaving your desktop, you can work with interrupts, directly manipulate CPU registers, and directly access the I/O space. No other modern 32-bit development environment for the standard PC allows this much so easily. The ubiquitous PC running under DOS (or a DOS console within any variant of Windows) is as close as it gets to emulating embedded software development on the commodity 80x86 hardware. Additionally, you can use free, mature tools, such as the Borland C/C++ compiler.

I also provide an embedded version for the inexpensive\(^2\) ARM Cortex-M3-based Stellaris EV-LM3S811 evaluation kit (see Figure 1.2). Both the PC and Cortex-M3

\(^1\) The accompanying code actually contains many more versions of the “Fly ‘n’ Shoot” game, but they are not relevant at this point.

\(^2\) At the time of this writing the EV-LM3S811 kit was available for $49 (www.luminarymicro.com).
versions use the exact same source code for all application components and differ only in the Board Support Package (BSP).

1.2 Let’s Play

The following description of the “Fly ‘n’ Shoot” game serves the dual purpose of explaining how to play the game and as the problem specification for the purpose of designing and implementing the software later in the chapter. To accomplish these two goals I need to be quite detailed, so please bear with me.

Your objective in the game is to navigate a spaceship through an endless horizontal tunnel with mines. Any collision with the tunnel or the mine destroys the ship. You can move the ship up and down with Up-arrow and Down-arrow keys on the PC (see Figure 1.1) or via the potentiometer wheel on the EV-LM3S811 board (see Figure 1.2). You can also fire a missile to destroy the mines in the tunnel by pressing the Spacebar on the PC or the User button on the EV-LM3S811 board. Score accumulates for survival (at the rate of 30 points per second) and destroying the mines. The game lasts for only one ship.

The game starts in a demo mode, where the tunnel walls scroll at the normal pace from right to left and the “Press Button” text flashes in the middle of the screen. You need to generate the “fire missile” event for the game to begin (press Spacebar on the PC or the User button on the EV-LM3S811 board).

You can have only one missile in flight at a time, so trying to fire a missile while it is already flying has no effect. Hitting the tunnel wall with the missile brings you no points, but you earn extra points for destroying the mines.

The game has two types of mines with different behavior. In the original Luminary Quickstart application both types of mines behave the same, but I wanted to demonstrate how state machines can elegantly handle differently behaving mines.

Mine type 1 is small, but can be destroyed by hitting any of its pixels with the missile. You earn 25 points for destroying a mine type 1. Mine type 2 is bigger but is nastier in that the missile can destroy it only by hitting its center, not any of the “tentacles.” Of course, the ship is vulnerable to the whole mine. You earn 45 points for destroying a mine type 2.

When you crash the ship, by either hitting a wall or a mine, the game ends and displays the flashing “Game Over” text as well as your final score. After 5 seconds of flashing,
the “Game Over” screen changes back to the demo screen, where the game waits to be started again.

Additionally the application contains a screen saver because the OLED display of the original EV-LM3S811 board has burn-in characteristics similar to a CRT. The screen saver only becomes active if 20 seconds elapse in the demo mode without starting the game (i.e., the screen saver never appears during game play). The screen saver is a simple random pixel type rather than the “Game of Life” algorithm used in the original Luminary Quickstart application. I’ve decided to simplify this aspect of the implementation because the more elaborate pixel-mixing algorithm does not contribute any new or interesting behavior.

After a minute of running the screen saver, the display turns blank and only a single random pixel shows on the screen. Again, this is a little different from the original Quickstart application, which instead blanks the screen and starts flashing the User LED. I’ve changed this behavior because I have a better purpose for the User LED (to visualize the activity of the idle loop).

![Image of the game](image_path)

**Figure 1.1:** The “Fly ‘n’ Shoot” game running in a DOS window under Windows XP.
1.2.1 Running the DOS Version

The “Fly ‘n’ Shoot” sample code for the DOS version (in C) is located in the `<qp>/qpc/examples\80x86\dos\tcppl01\l\game\` directory, where `<qp>` stands for the installation directory in which you chose to install the accompanying software.

The compiled executable is provided, so you can run the game on any Windows-based PC by simply double-clicking the executable `game.exe` located in the directory `<qp>/qpc/examples\80x86\dos\tcppl01\l\game\dbg\`. The first screen you see is the game running in the demo mode with the text “Push Button” flashing in the middle of the display. At the top of the display you see a legend of keystrokes recognized by the application. You need to hit the `SPACEBAR` to start playing the game. Press the `ESC` key to cleanly exit the application.

If you run “Fly ‘n’ Shoot” in a window under Microsoft Windows, the animation effects in the game might appear a little jumpy, especially compared to the Stellaris version of the same game. You can make the application execute significantly more smoothly if you switch to the full-screen mode by pressing and holding the `Alt` key and then pressing the `Enter` key. You go back to the window mode via the same `Alt-Enter` key combination.

As you can see in Figure 1.1, the DOS version uses simply the standard VGA text mode to emulate the OLED display of the EV-LM3S811 board. The lower part of the DOS screen

Figure 1.2: The “Fly ‘n’ Shoot” game running on the Stellaris EV-LM3S811 evaluation board.
is used as a matrix of $80 \times 16$ character-wide “pixels,” which is a little less than the $96 \times 16$ pixels of the OLED display but still good enough to play the game. I specifically avoid employing any fancier graphics in this early example because I have bigger fish to fry for you than to worry about the irrelevant complexities of programming graphics.

My main goal is to make it easy for you to understand the event-driven code and experiment with it. To this end, I chose the legacy Borland Turbo C++ 1.01 toolset to build this example as well as several other examples in this book. Even though Turbo C++ 1.01 is an older compiler, it is adequate to demonstrate all features of both the C and C++ versions. Best of all, it is available for a free download from the Borland “Museum” at http://bdn.borland.com/article/0,1410,21751,00.html.

The toolset is very easy to install. After you download the Turbo C++ 1.01 files directly from Borland, you need to unzip the files onto your hard drive. Then you run the INSTALL.EXE program and follow the installation instructions it provides.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I strongly recommend that you install the Turbo C++ 1.01 toolset into the directory C:\tools\tcpp101\ . That way you will be able to directly use the provided project files and make scripts.</td>
</tr>
</tbody>
</table>

Perhaps the easiest way to experiment with the “Fly ‘n’ Shoot” code is to launch the Turbo C++ IDE (TC.EXE) and open the provided project file GAME-DBG.PRJ, which is located in the directory <qp>\qpc\examples\80x86\dos\tcpp101\l\game\. You can modify, recompile, execute, and debug the program directly from the IDE. However, you should avoid terminating the program stopped in the debugger, because this will not restore the standard DOS interrupt vectors for the time tick and keyboard interrupts. You should always cleanly exit the application by letting it freely run and pressing the Esc key.

The next section briefly describes how to run the embedded version of the game. If you are not interested in the Cortex-M3 version, feel free to skip to Section 1.3, where I start explaining the application code.

### 1.2.2 Running the Stellaris Version

In contrast to the “Fly ‘n’ Shoot” version for DOS running in the ancient real mode of the 80x86 processor, the exact same source code runs on one of the most modern processors in the industry: the ARM Cortex-M3.
The sample code for the Stellaris EV-LM3S811 board is located in the \examples\cortex-m3\vanilla\iar\game-ev-lm3s811\ directory, where <qp> stands for the root directory in which you chose to install the accompanying software.

The code for the Stellaris kit has been compiled with the 32KB-limited Kickstart edition of the IAR Embedded Workbench for ARM (IAR EWARM) v 5.11, which is provided with the Stellaris EV-LM3S811 kit. You can also download this software free of charge directly from IAR Systems (www.iar.com) after filling out an online registration.

The installation of IAR EWARM is quite straightforward, since the software comes with the installation utility. You also need to install the USB drivers for the hardware debugger built into the EV-LM3S811 board, as described in the documentation of the Stellaris EV-LM3S811 kit.

### NOTE

I strongly recommend that you install the IAR EWARM toolset into the directory C:\tools\iar\arm_ks_5.11. That way you will be able to directly use the provided EWARM workspace files and make scripts.

Before you program the “Fly ‘n’ Shoot” game to the EV-LM3S811 board, you might want to play a little with the original Quickstart application that comes preprogrammed with the EV-LM3S811 kit.

To program the “Fly ‘n’ Shoot” game to the Flash memory of the EV-LM3S811 board, you first connect the EV-LM3S811 board to your PC with the USB cable provided in the kit and make sure that the Power LED is on (see Figure 1.2). Next, you need to launch the IAR Embedded Workbench and open the workspace game-ev-lm3s811.eww located in the \examples\cortex-m3\vanilla\iar\game-ev-lm3s811\ directory. At this point your screen should look similar to the screenshot shown in Figure 1.3.

The game-ev-lm3s811 project is set up to use the LMI FTDI debugger, which is the piece of hardware integrated on the EV-LM3S811 board (see Figure 1.2). You can verify this setup by opening the “Options” dialog box via the Project | Options menu. Within the “Options” dialog box, you need to select the Debugger category in the panel on the left. While you’re at it, you could also verify that the Flash loading is enabled by selecting the “Download” tab. The checked “Use flash loader(s)” check box means
that the Flash loader application provided by IAR will be first loaded to the RAM of the MCU, and this application will program the Flash with the image of your application.

To start the Flash programming process, select the Project | Debug menu, or simply click the Debug button (see Figure 1.3) in the toolbar. The IAR Workbench should respond by showing the Flash programming progress bar for several seconds, as shown in Figure 1.3. Once the Flash programming completes, the IAR EWARM switches to the IAR C-Spy debugger and the program should stop at the entry to \texttt{main()}. You can start playing the game either by clicking the Go button in the debugger or you can close the debugger and reset the board by pressing the Reset button. Either way, the “Fly ‘n’ Shoot” game is now permanently programmed into the EV-LM3S811 board and will start automatically on every powerup.

![Figure 1.3: Loading the “Fly ‘n’ Shoot” game into the flash of LM3S811 MCU with IAR EWARM IDE.](image-url)
The IAR Embedded Workbench environment allows you to experiment with the “Fly ‘n’ Shoot” code very easily. You can edit the files and recompile the application at a click of a button (F7). The only caveat is that the first time after the installation of the IAR toolset you need to build the Luminary Micro driver library for the LM3S811 MCU from the sources. You accomplish this by loading the workspace ek-lm3s811.eww located in the directory <IAR-EWARM>\ARM\examples \Luminary\Stellaris\boards\ek-lm3s811, where <IAR-EWARM> stands for the directory name where you’ve installed the IAR toolset. In the ev-lm3s811.eww workspace, you select the “driverlib - Debug” project from the drop-down list at the top of the Workspace panel and then press F7 to build the library.

1.3 The main() Function

Perhaps the best place to start the explanation of the “Fly ‘n’ Shoot” application code is the main() function, located in the file main.c. Unless indicated otherwise in this chapter, you can browse the code in either the DOS version or the EV-LM3S811 version, because the application source code is identical in both. The complete main.c file is shown in Listing 1.1.

NOTE

To explain code listings, I place numbers in parentheses at the interesting lines in the left margin of the listing. I then use these labels in the left margin of the explanation section that immediately follows the listing. Occasionally, to unambiguously refer to a line of a particular listing from sections of text other than the explanation section, I use the full reference consisting of the listing number followed by the label. For example, Listing 1.1(21) refers to the label (21) in Listing 1.1.

Listing 1.1 The file main.c of the “Fly ‘n’ Shoot” game application

```c
#include "qp_port.h" /* the QP port */
#include "bsp.h" /* Board Support Package */
#include "game.h" /* this application */

/* Local-scope objects --------------------------------------------------*/
static QEvent const * l_missileQueueSto[2]; /* event queue */
static QEvent const * l_shipQueueSto[3]; /* event queue */
static QEvent const * l_tunnelQueueSto[GAME_MINES_MAX + 5]; /* event queue */
```
The “Fly ‘n’ Shoot” game is an example of an application implemented with the QP event-driven platform. Every application C-file that uses QP must include the `qp_port.h` header file. This header file contains the specific adaptation of QP to the given processor, operating system, and compiler, which is called a port. Each QP port is located in a separate directory, and the C compiler finds the right `qp_port.h` header file through the include search path provided to the compiler.
(typically via the \texttt{-I} compiler option). That way I don’t need to change the application source code to recompile it for a different processor or compiler. I only need to instruct the compiler to look in a different QP port directory for the \texttt{qp_port.h} header file. For example, the DOS version includes the \texttt{qp_port.h} header file from the directory <\texttt{qp}\textbackslash \texttt{qpc}\textbackslash \texttt{ports}\textbackslash \texttt{80x86}\textbackslash \texttt{dos}\textbackslash \texttt{tcppl01}\textbackslash \texttt{l}>, and the EV-LM3S811 version from the directory <\texttt{qp}\textbackslash \texttt{qpc}\textbackslash \texttt{ports}\textbackslash \texttt{cortex-m3}\textbackslash \texttt{vanilla}\textbackslash \texttt{iar}>.

(2) The \texttt{bsp.h} header file contains the interface to the Board Support Package and is located in the application directory.

(3) The \texttt{game.h} header file contains the declarations of events and other facilities shared among the components of the application. I will discuss this header file in the upcoming Section 1.7. This header file is located in the application directory.

The QP event-driven platform is a collection of components, such as the QEP event processor that executes state machines according to the UML semantics and the QF real-time framework that implements the active object computing model. Active objects in QF are encapsulated state machines (each with an event queue, a separate task context, and a unique priority) that communicate with one another asynchronously by sending and receiving events, whereas QF handles all the details of thread-safe event exchange and queuing. Within an active object, the events are processed by the QEP event processor sequentially in a run-to-completion (RTC) fashion, meaning that processing of one event must necessarily complete before processing the next event. (See also Section 6.3.3 in Chapter 6.)

(4-6) The application must provide storage for the event queues of all active objects used in the application. Here the storage is provided at compile time through the statically allocated arrays of immutable (\texttt{const}) pointers to events, because QF event queues hold just pointers to events, not events themselves. Events are represented as instances of the \texttt{QEvent} structure declared in the \texttt{qp_port.h} header file. Each event queue of an active object can have a different size, and you need to decide this size based on your knowledge of the application. Event queues are discussed in Chapters 6 and 7.

(7,8) The application must also provide storage for event pools that the framework uses for fast and deterministic dynamic allocation of events. Each event pool
can provide only fixed-size memory blocks. To avoid wasting memory by using oversized blocks for small events, the QF framework can manage up to three event pools of different block sizes (for small, medium, and large events). The “Fly ‘n’ Shoot” application uses only two out of the three possible event pools (the small and medium pools).

The QF real-time framework supports two event delivery mechanisms: the simple direct event posting to active objects and the more advanced mechanism called *publish-subscribe* that decouples event producers from the consumers. In the publish-subscribe mechanism, active objects subscribe to events by the framework. Event producers publish the events to the framework. Upon each publication request, the framework delivers the event to all active objects that had subscribed to that event type. One obvious implication of publish-subscribe is that the framework must store the subscriber information, whereas it must be possible to handle multiple subscribers to any given event type. The event delivery mechanisms are described in Chapters 6 and 7.

(9) The “Fly ‘n’ Shoot” application uses the publish-subscribe event delivery mechanism supported by QF, so it needs to provide the storage for the subscriber lists. The subscriber lists remember which active objects have subscribed to which events. The size of the subscriber database depends on both the number of published events, which is specified in the `MAX_PUB_SIG` constant found in the `game.h` header file, and the maximum number of active objects allowed in the system, which is determined by the QF configuration parameter `QF_MAX_ACTIVE`.

(10-12) These functions perform an early initialization of the active objects in the system. They play the role of static “constructors,” which in C you need to invoke explicitly. (C++ calls such static constructors implicitly before entering `main()`).

(13) The function `BSP_init()` initializes the board and is defined in the `bsp.c` file.

(14) The function `QF_init()` initializes the QF component and the underlying RTOS/kernel, if such software is used. You need to call `QF_init()` before you invoke any QF services.

(15,16) The function `QF_poolInit()` initializes the event pools. The parameters of this function are the pointer to the event pool storage, the size of this storage,
and the block-size of this pool. You can call this function up to three times to initialize up to three event pools. The subsequent calls to QF_poolInit() must be made in the increasing order of block size. For instance, the small block-size pool must be initialized before the medium block-size pool.

(17) The function QF_poolInit() initializes the publish-subscribe event delivery mechanism of QF. The parameters of this function are the pointer to the subscriber-list array and the dimension of this array.

The utility macro Q_DIM(a) provides the dimension of a one-dimensional array a[] computed as sizeof(a)/sizeof(a[0]), which is a compile-time constant. The use of this macro simplifies the code because it allows me to eliminate many #define constants that otherwise I would need to provide for the dimensions of various arrays. I can simply hard-code the dimension right in the definition of an array, which is the only place that I specify it. I then use the macro Q_DIM() whenever I need this dimension in the code.

(18-20) The function QActive_start() tells the QF framework to start managing an active object as part of the application. The function takes the following parameters: the pointer to the active object structure, the priority of the active object, the pointer to its event queue, the dimension (length) of that queue, and three other parameters that I explain in Chapter 7 (they are not relevant at this point). The active object priorities in QF are numbered from 1 to QF_MAX_ACTIVE, inclusive, where a higher-priority number denotes higher urgency of the active object. The constant QF_MAX_ACTIVE is defined in the QF port header file qf_port.h and currently cannot exceed 63.

I like to keep the code and data of every active object strictly encapsulated within its own C-file. For example, all code and data for the active object Ship are encapsulated in the file ship.c, with the external interface consisting of the function Ship_ctor() and the pointer AO_Ship.

(21) At this point, you have provided to the framework all the storage and information it needs to manage your application. The last thing you must do is call the function QF_run() to pass the control to the framework.

After the call to QF_run() the framework is in full control. The framework executes the application by calling your code, not the other way around. The function QF_run() never returns the control back to main(). In the DOS version of the
“Fly ‘n’ Shoot” game, you can terminate the application by pressing the Esc key, in which case QF_run() exits to DOS but not to main(). In an embedded system, such as the Stellaris board, QF_run() runs forever or till the power is removed, whichever comes first.

### NOTE

For best cross-platform portability, the source code consistently uses the *UNIX end-of-line convention* (lines are terminated with LF only, 0xA character). This convention seems to be working for all C/C++ compilers and cross-compilers, including legacy DOS-era tools. In contrast, the DOS/Windows end-of-line convention (lines terminated with the CR,LF, or 0xD,0xA pair of characters) is known to cause problems on UNIX-like platforms, especially in the multiline preprocessor macros.

### 1.4 The Design of the “Fly ‘n’ Shoot” Game

To proceed further with the explanation of the “Fly ‘n’ Shoot” application, I need to step up to the design level. At this point I need to explain how the application has been decomposed into the active objects and how these objects exchange events to collectively deliver the functionality of the “Fly ‘n’ Shoot” game.

In general, the decomposition of a problem into active objects is not trivial. As usual in any decomposition, your goal is to achieve possibly loose coupling among the active object components (ideally no sharing of any resources), and you also strive for minimizing the communication in terms of the frequency and size of exchanged events.

In the case of the “Fly ‘n’ Shoot” game, I need to first identify all objects with reactive behavior (i.e., with a state machine). I applied the simplest object-oriented technique of identifying objects, which is to pick the frequently used nouns in the problem specification. From Section 1.2, I identified Ship, Missile, Mines, and Tunnel. However, not every state machine in the system needs to be an active object (with a separate task context, an event queue, and a unique priority level), and merging them is a valid option when performance or space is needed. As an example of this idea, I ended up merging the Mines into the Tunnel active object, whereas I preserved the Mines as independent state machine components of the Tunnel active object. By doing so I applied the “Orthogonal Component” design pattern described in Chapter 5.
The next step in the event-driven application design is assigning responsibilities and resources to the identified active objects. The general design strategy for avoiding sharing of resources is to encapsulate each resource inside a dedicated active object and to let that object manage the resource for the rest of the application. That way, instead of sharing the resource directly, the rest of the application shares the dedicated active object via events.

So, for example, I decided to put the Tunnel active object in charge of the display. Other active objects and state machine components, such as Ship, Missile, and Mines, don’t draw on the display directly, but rather send events to the Tunnel object with the request to render the Ship, Missile, or Mine bitmaps at the provided \((x, y)\) coordinates of the display.

With some understanding of the responsibilities and resource allocations to active objects I can move on to devising the various scenarios of event exchanges among the objects. Perhaps the best instrument to aid the thinking process at this stage is the UML sequence diagram, such as the diagram depicted in Figure 1.4. This particular sequence diagram shows the most common event exchange scenarios in the “Fly ‘n’ Shoot” game (the primary use cases, if you will). The explanation section immediately following the diagram illuminates the interesting points.

**NOTE**

A UML sequence diagram like Figure 1.4 has two dimensions. Horizontally arranged boxes represent the various objects participating in the scenario, whereas heavy borders indicate active objects. As usual in the UML, the object name is underlined. Time flows down the page along the vertical dashed lines descending from the objects. Events are represented as horizontal arrows originating from the sending object and terminating at the receiving object. Optionally, thin rectangles around instance lines indicate focus of control.

**NOTE**

To explain diagrams, I place numbers in parentheses at the interesting elements of the diagram. I then use these labels in the left margin of the explanation section that immediately follows the diagram. Occasionally, to unambiguously refer to a specific element of a particular diagram from sections of text other than the explanation section, I use the full reference consisting of the figure number followed by the label. For example, Figure 1.4(12) refers to the element (12) in Figure 1.4.
(1) The `TIME_TICK` is the most important event in the game. This event is generated by the QF framework from the system time tick interrupt at a rate of 30 times per second, which is needed to drive a smooth animation of the display. Because the `TIME_TICK` event is of interest to virtually all objects in the application, it is published by the framework to all active objects. (The publish-subscribe event delivery in QF is described in Chapter 6.)

(2) Upon reception of the `TIME_TICK` event, the Ship object advances its position by one step and posts the event `SHIP_IMG(x, y, bmp)` to the Tunnel object. The
SHIP_IMG event has parameters $x$ and $y$, which are the coordinates of the Ship on the display, as well as the bitmap number $bmp$ to draw at these coordinates.

(3) The Missile object is not in flight yet, so it simply ignores the TIME_TICK event this time.

(4) The Tunnel object performs the heaviest lifting for the TIME_TICK event. First, Tunnel redraws the entire display from the current frame buffer. This action, performed 30 times per second, provides the illusion of animation of the display. Next, the Tunnel clears the frame buffer and starts filling it up again for the next time frame. The Tunnel advances the tunnel walls by one step and copies the walls to the frame buffer. The Tunnel also dispatches the TIME_TICK event to all its Mine state machine components.

(5) Each Mine advances its position by one step and posts the MINE_IMG($x$, $y$, $bmp$) event to the Tunnel to render the appropriate Mine bitmap at the position ($x$, $y$) in the current frame buffer. Mines of type 1 send the bitmap number MINE1_BMP, whereas mines of type 2 send MINE2_BMP.

(6) Upon receipt of the SHIP_IMG($x$, $y$, $bmp$) event from the Ship, the Tunnel object renders the specified bitmap in the frame buffer and checks for any collision between the ship bitmap and the tunnel walls. Tunnel also dispatches the original SHIP_IMG($x$, $y$, $bmp$) event to all active Mines.

(7) Each Mine determines whether the Ship is in collision with that Mine.

(8) The PLAYER_TRIGGER event is generated when the Player reliably presses the button (button press is debounced). This event is published by the QF framework and is delivered to the Ship and Tunnel objects, which both subscribe to the PLAYER_TRIGGER event.

(9) Ship generates the MISSILE_FIRE($x$, $y$) event to the Missile object. The parameters of this event are the current ($x$, $y$) coordinates of the Ship, which are the starting point for the Missile.

(10) Tunnel receives the published PLAYER_TRIGGER event as well because Tunnel occasionally needs to start the game or terminate the screen saver mode based on this stimulus.

(11) Missile reacts to the MISSILE_FIRE($x$, $y$) event by starting to fly, whereas it sets its initial position from the ($x$, $y$) event parameters delivered from the Ship.
(12) This time around, the `TIME_TICK` event arrives while Missile is in flight. Missile posts the `MISSILE_IMG(x, y, bmp)` event to the Table.

(13) Table renders the Missile bitmap in the current frame buffer and dispatches the `MISSILE_IMG(x, y, bmp)` event to all the Mines to let the Mines test for the collision with the Missile. This determination depends on the type of the Mine. In this scenario a particular Mine[n] object detects a hit and posts the `HIT_MINE(score)` event to the Missile. The Mine provides the score earned for destroying this particular mine as the parameter of this event.

(14) Missile handles the `HIT_MINE(score)` event by becoming immediately ready to launch again and lets the Mine do the exploding. Because I decided to make the Ship responsible for the scorekeeping, the Missile also generates the `DESTROYED_MINE(score)` event to the Ship, to report the score for destroying the Mine.

(15) Upon reception of the `DESTROYED_MINE(score)` event, the Ship increments the score by the value received from the Missile.

(16) The Ship object handles the `PLAYER_SHIP_MOVE(x, y)` event by updating its position from the event parameters.

(17) When the Tunnel object handles the `SHIP_IMG(x, y, bmp_id)` event next time around, it detects a collision between the Ship and the tunnel wall. In that case it posts the event `HIT_WALL` to the Ship.

(18) The Ship responds to the `HIT_WALL` event by transitioning to the “exploding” state.

Even though the sequence diagram in Figure 1.4 shows merely some selected scenarios of the “Fly ‘n’ Shoot” game, I hope that the explanations give you a big picture of how the application works. More important, you should start getting the general idea about the thinking process that goes into designing an event-driven system with active objects and events.

1.5 Active Objects in the “Fly ‘n’ Shoot” Game

I hope that the analysis of the sequence diagram in Figure 1.4 makes it clear that actions performed by an active object depend as much on the events it receives as on the internal mode of the object. For example, the Missile active object handles the `TIME_TICK` event very differently when the Missile is in flight (Figure 1.4(12)) compared to the time when it is not (Figure 1.4(3)).
The best-known mechanism for handling such modal behavior is through state machines because a state machine makes the behavior explicitly dependent on both the event and the state of an object. Chapter 2 introduces UML state machine concepts more thoroughly. In this section, I give a cursory explanation of the state machines associated with each object in the “Fly ‘n’ Shoot” game.

1.5.1 The Missile Active Object

I start with the Missile state machine shown in Figure 1.5 because it turns out to be the simplest one. The explanation section immediately following the diagram illuminates the interesting points.

NOTE

A UML state diagram like Figure 1.5 preserves the general form of the traditional state transition diagrams, where states are represented as nodes and transitions as arcs connecting the nodes. In the UML notation the state nodes are represented as rectangles with rounded corners. The name of the state appears in bold type in the name compartment at the top of the state. Optionally, right below the name, a state can have an internal transition compartment separated from the name by a horizontal line. The internal transition compartment can contain entry actions (actions following the reserved symbol “entry”), exit actions (actions following the reserved symbol “exit”), and other internal transitions (e.g., those triggered by \texttt{TIME\_TICK} in Figure 1.5(3)). State transitions are represented as arrows originating at the boundary of the source state and pointing to the boundary of the target state. At a minimum, a transition must be labeled with the triggering event. Optionally, the trigger can be followed by event parameters, a guard, and a list of actions.

(1) The state transition originating at the black ball is called the \textit{initial transition}. Such transition designates the first active state after the state machine object is created. An initial transition can have associated actions, which in the UML notation are enlisted after the forward slash ( / ). In this particular case, the Missile state machine starts in the “armed” state and the actions executed upon the initialization consist of subscribing to the event \texttt{TIME\_TICK}. Subscribing to an event means that the framework will deliver the specified event to the Missile active object every time the event is published to the framework. Chapter 7 describes the implementation of the publish-subscribe event delivery in QF.
The arrow labeled with the MISSILE_FIRE(x, y) event denotes a state transition, that is, a change of state from “armed” to “flying.” The MISSILE_FIRE(x, y) event is generated by the Ship object when the Player triggers the Missile (see the sequence diagram in Figure 1.4). In the MISSILE_FIRE event, Ship provides Missile with the initial coordinates in the event parameters (x, y).

NOTE
The UML intentionally does not specify the notation for actions. In practice, the actions are often written in the programming language used for coding the particular state machine. In all state diagrams in this book, I assume the C programming language. Furthermore, in the C expressions I refer to the data members associated with the state machine object through the “me->” prefix and to the event parameters through the “e->” prefix. For example, the action “me->x = e->x;” means that the internal data member x of the Missile active object is assigned the value of the event parameter x.
(3) The event name `TIME_TICK` enlisted in the compartment below the state name denotes an internal transition. Internal transitions are simple reactions to events performed without a change of state. An internal transition, as well as a regular transition, can have a guard condition, enclosed in square brackets. Guard condition is a Boolean expression evaluated at runtime. If the guard evaluates to TRUE, the transition is taken. Otherwise, the transition is not taken and no actions enlisted after the forward slash (/) are executed. In this particular case, the guard condition checks whether the $x$-coordinate propagated by the Missile speed is still visible on the screen. If so, the actions are executed. These actions include propagation of the Missile position by one step and posting the `MISSILE_IMG` event with the current Missile position and the `MISSILE_BMP` bitmap number to the Tunnel active object. Direct event posting to an active object is accomplished by the QF function `QActive_postFIFO()`, which I discuss in Chapter 7.

(4) The same event `TIME_TICK` with the [else] guard denotes a regular state transition with the guard condition complementary to the other occurrence of the `TIME_TICK` event in the same state. In this case, the `TIME_TICK` transition to “armed” is taken if the Missile object flies out of the screen.

(5) The event `HIT_MINE(score)` triggers another transition to the “armed” state. The action associated with this transition posts the `DESTROYED_MINE` event with the parameter `e->score` to the Ship object, to report destroying the mine.

(6) The event `HIT_WALL` triggers a transition to the “exploding” state, with the purpose of animating the explosion bitmaps on the display.

(7) The label “entry” denotes the entry action to be executed unconditionally upon the entry to the “exploding” state. This action consists of clearing the explosion counter `(me->exp_ctr)` member of the Missile object.

(8) The `TIME_TICK` internal transition is guarded by the condition that the explosion does not scroll off the screen and that the explosion counter is lower than 16. The actions executed include propagation of the explosion position and posting the `EXPLOSION_IMG` event to the Tunnel active object. Please note that the bitmap of the explosion changes as the explosion counter gets bigger.

(9) The `TIME_TICK` regular transition with the complementary guard changes the state back to the “armed” state. This transition is taken after the animation of the explosion completes.
1.5.2 The Ship Active Object

The state machine of the Ship active object is shown in Figure 1.6. This state machine introduces the profound concept of hierarchical state nesting. The power of state nesting derives from the fact that it is designed to eliminate repetitions that otherwise would have to occur.

One of the main responsibilities of the Ship active object is to maintain the current position of the Ship. On the original EV-LM3S811 board, this position is determined by the potentiometer wheel (see Figure 1.2). The PLAYER_SHIP_MOVE(x, y) event is generated whenever the wheel position changes, as shown in the sequence diagram (Figure 1.4). The Ship object must always keep track of the wheel position, which means that all states of the Ship state machine must handle the PLAYER_SHIP_MOVE(x, y) event.

In the traditional finite state machine (FSM) formalism, you would need to repeat the Ship position update from the PLAYER_SHIP_MOVE(x, y) event in every state. But such repetitions would bloat the state machine and, more important, would represent multiple points of maintenance both in the diagram and the code. Such repetitions go against the DRY (Don’t Repeat Yourself) principle, which is vital for flexible and maintainable code [Hunt+ 00].

Hierarchical state nesting remedies the problem. Consider the state “active” that surrounds all other states in Figure 1.6. The high-level “active” state is called the superstate and is abstract in that the state machine cannot be in this state directly but only in one of the states nested within, which are called the substates of “active.” The UML semantics associated with state nesting prescribe that any event is first handled in the context of the currently active substate. If the substate cannot handle the event, the state machine attempts to handle the event in the context of the next-level superstate. Of course, state nesting in UML is not limited to just one level and the simple rule of processing events applies recursively to any level of nesting.

Specifically to the Ship state machine diagram shown in Figure 1.6, suppose that the event PLAYER_SHIP_MOVE(x, y) arrives when the state machine is in the “parked” state. The “parked” state does not handle the PLAYER_SHIP_MOVE(x, y) event. In the traditional finite state machine this would be the end of the story—the PLAYER_SHIP_MOVE(x, y) event would be silently discarded. However, the state machine in Figure 1.6 has another layer of the “active” superstate. Per the semantics of state nesting, this higher-level superstate handles the PLAYER_SHIP_MOVE(x, y) event, which is exactly what’s needed. The same exact argumentation applies for any other substate of the “active” superstate, such as “flying”
or “exploding,” because none of these substates handle the \texttt{PLAYER\_SHIP\_MOVE}(x, y) event. Instead, the “active” superstate handles the event in one single place, without repetitions.

Figure 1.6: Ship state machine diagram.

1. Upon the initial transition, the Ship state machine enters the “active” superstate and subscribes to events \texttt{TIME\_TICK} and \texttt{PLAYER\_TRIGGER}.

2. At each level of nesting, a superstate can have a private initial transition that designates the active substate after the superstate is entered directly. Here the

```c
me->x = e->x;
me->y = e->y;
```

3. \texttt{PLAYER\_SHIP\_MOVE}(x, y) / (3)

```c
++me->score;
if ((me->score % 10) == 0)
  QActive_postFIFO(Tunnel, SCORE(me->score));
```

4. \texttt{TIME\_TICK} / (6)

```c
++me->exp_ctr;
QActive_postFIFO(Tunnel,
  EXPLOSION0_BMP + (me->exp_ctr >> 2));
```

5. \texttt{TIME\_TICK} [me->exp_ctr < 16] / (11)

```c
++me->exp_ctr;
QActive_postFIFO(Tunnel,
  EXPLOSION(me->x, me->y + SHIP\_HEIGHT -1,
  EXPLOSION0_BMP + (me->exp_ctr >> 2)));
```

6. \texttt{TIME\_TICK} [else] / (12)

Figure 1.6: Ship state machine diagram.

1. Upon the initial transition, the Ship state machine enters the “active” superstate and subscribes to events \texttt{TIME\_TICK} and \texttt{PLAYER\_TRIGGER}.

2. At each level of nesting, a superstate can have a private initial transition that designates the active substate after the superstate is entered directly. Here the
initial transition of state “active” designates the substate “parked” as the initial active substate.

(3) The “active” superstate handles the \texttt{PLAYER\_SHIP\_MOVE}(x, y) event as an internal transition in which it updates the internal data members \texttt{me->x} and \texttt{me->y} from the event parameters \texttt{e->x} and \texttt{e->y}, respectively.

(4) The \texttt{TAKE\_OFF} event triggers transition to “flying.” This event is generated by the Tunnel object when the Player starts the game (see the description of the game in Section 1.2).

(5) The entry actions to “flying” include clearing the \texttt{me->score} data member and posting the event \texttt{SCORE} with the event parameter \texttt{me->score} to the Tunnel active object.

(6) The \texttt{TIME\_TICK} internal transition causes posting the event \texttt{SHIP\_IMG} with current Ship position and the \texttt{SHIP\_BMP} bitmap number to the Tunnel active object. Additionally, the score is incremented for surviving another time tick. Finally, when the score is “round” (divisible by 10) it is also posted to the Tunnel active object. This decimation of the \texttt{SCORE} event is performed just to reduce the bandwidth of the communication, because the Tunnel active object only needs to give an approximation of the running score tally to the user.

(7) The \texttt{PLAYER\_TRIIGGER} internal transition causes posting the event \texttt{MISSILE\_FIRE} with current Ship position to the Missile active object. The parameters (\texttt{me->x}, \texttt{me->y}) provide the Missile with the initial position from the Ship.

(8) The \texttt{DESTROYED\_MINE}(score) internal transition causes update of the score kept by the Ship. The score is not posted to the Table at this point, because the next \texttt{TIME\_TICK} will send the “rounded” score, which is good enough for giving the Player the score approximation.

(9) The \texttt{HIT\_WALL} event triggers transition to “exploding.”

(10) The \texttt{HIT\_MINE}(type) event also triggers transition to “exploding.”

(11) The “exploding” state of the Ship state machine is very similar to the “exploding” state of Missile (see Figure 1.5(7-9)).

(12) The \texttt{TIME\_TICK[else]} transition is taken when the Ship finishes exploding. Upon this transition, the Ship object posts the event \texttt{GAME\_OVER}(me->score) to the Tunnel active object to terminate the game and display the final score to the Player.
1.5.3 The Tunnel Active Object

The Tunnel active object has the most complex state machine, which is shown in Figure 1.7. Unlike the previous state diagrams, the diagram in Figure 1.7 shows only the high level of abstraction and omits a lot of details such as most entry/exit actions, internal transitions, guard conditions, or actions on transitions. Such a “zoomed out” view is always legal in the UML because UML allows you to choose the level of detail that you want to include in your diagram.

The Tunnel state machine uses state hierarchy more extensively than the Ship state machine in Figure 1.6. The explanation section immediately following Figure 1.7 illuminates the new uses of state nesting as well as the new elements not explained yet in the other state diagrams.

![Figure 1.7: Tunnel state machine diagram.](www.newnespress.com)
An initial transition can target a substate at any level of state hierarchy, not necessarily just the next-lower level. Here the topmost initial transition goes down two levels to the substate “demo.”

The superstate “active” handles the `PLAYER_QUIT` event as a transition to the final state (see explanation of element (3)). Please note that the `PLAYER_QUIT` transition applies to all substates directly or transitively nested in the “active” superstate. Because a state transition always involves execution of all exit actions from the states, the high-level `PLAYER_QUIT` transition guarantees the proper cleanup that is specific to the current state context, whichever substate happens to be active at the time when the `PLAYER_QUIT` event arrives.

The final state is indicated in the UML notation as the bull’s-eye symbol and typically indicates destruction of the state machine object. In this case, the `PLAYER_QUIT` event indicates termination of the game.

The `MINE_DISABLED(mine_id)` event is handled at the high level of the “active” state, which means that this internal transition applies to the whole sub-machine nested inside the “active” superstate. (See also the discussion of the Mine object in the next section.)

The entry action to the “demo” state starts the screen `time event` (timer) `me->screenTimeEvt` to expire in 20 seconds. Time events are allocated by the application, but they are managed by the QF framework. QF provides functions to arm a time event, such as `QTimeEvt_postIn()` for one-shot timeout, and `QTimeEvt_postEvery()` for periodic time events. Arming a time event is in effect telling the QF framework, for instance, “Give me a nudge in 20 seconds.” QF then posts the time event (the event `me->screenTimeEvt` in this case) to the active object after the requested number of clock ticks. Chapters 6 and 7 talk about time events in detail.

The exit action from the “demo” state disarms the `me->screenTimeEvt` time event. This cleanup is necessary when the state can be exited by a different event than the time event, such as the `PLAYER_TRIGGER` transition.

The `SCREEN_TIMEOUT` transition to “screen_saver” is triggered by the expiration of the `me->screenTimeEvt` time event. The signal `SCREEN_TIMEOUT` is assigned to this time event upon initialization and cannot be changed later.

The transition triggered by `PLAYER_TRIGGER` applies equally to the two substates of the “screen_saver” superstate.
1.5.4 The Mine Components

Mines are also modeled as hierarchical state machines, but are not active objects. Instead, Mines are components of the Tunnel active object and share its event queue and priority level. The Tunnel active object communicates with the Mine components synchronously by directly dispatching events to them via the function QHsm_dispatch(). Mines communicate with Tunnel and all other active objects asynchronously by posting events to their event queues via the function QActive_postFIFO().

NOTE

Active objects exchange events asynchronously, meaning that the sender of the event merely posts the event to the event queue of the recipient active object without waiting for the completion of the event processing. In contrast, synchronous event processing corresponds to a function call (e.g., QHsm_dispatch()), which processes the event in the caller’s thread of execution.

As shown in Figure 1.8, Tunnel maintains the data member mines[], which is an array of pointers to hierarchical state machines (QHsm *). Each of these pointers can point either to a Mine1 object, a Mine2 object, or NULL, if the entry is unused. Note that Tunnel “knows” the Mines only as generic state machines (pointers to the QHsm structure defined in QP). Tunnel dispatches events to Mines uniformly, without differentiating between different types of Mines. Still, each Mine state machine handles the events in its specific way. For example, Mine type 2 checks for collision with the Missile differently than with the Ship, whereas Mine type 1 handles both identically.

Figure 1.8: The Table active object manages two types of Mines.
NOTE

The last point is actually very interesting. Dispatching the same event to different Mine objects results in different behavior, specific to the type of the Mine, which in OOP is known as *polymorphism*. I’ll have more to say about this in Chapter 3.

Each Mine object is fairly autonomous. The Mine maintains its own position and is responsible for informing the Tunnel object whenever the Mine gets destroyed or scrolls out of the display. This information is vital for the Tunnel object so that it can keep track of the unused Mines.

Figure 1.9 shows a hierarchical state machine of Mine2 state machine. Mine1 is very similar, except that it uses the same bitmap for testing collisions with the Missile and the Ship.

1. The Mine starts in the “unused” state.
2. The Tunnel object plants a Mine by dispatching the `MINE_PLANT(x, y)` event to the Mine. The Tunnel provides the \((x, y)\) coordinates as the original position of the Mine.
3. When the Mine scrolls off the display, the state machine transitions to “unused.”
4. When the Mine hits the Ship, the state machine transitions to “unused.”
5. When the Mine finishes exploding, the state machine transitions to “unused.”
6. When the Mine is recycled by the Tunnel object, the state machine transitions to “unused.”
7. The exit action in the “used” state posts the `MINE_DISABLED(mine_id)` event to the Tunnel active object. Through this event, the Mine informs the Tunnel that it’s becoming disabled, so that Tunnel can update its `mines[]` array (see also Figure 1.9(4)). The `mine_id` parameter of the event becomes the index into the `mines[]` array. Note that generating the `MINE_DISABLED(mine_id)` event in the exit action from “used” is much safer and more maintainable than repeating this action in each individual transition (3), (4), (5), and (6).
Figure 1.9: Mine2 state machine diagram.
1.6 Events in the “Fly ‘n’ Shoot” Game

The key events in the “Fly ‘n’ Shoot” game have been identified in the sequence diagram in Figure 1.4. Other events have been invented during the state machine design stage. In any case, you must have noticed that events consist really of two parts. The part of the event called the signal conveys the type of the occurrence (what happened). For example, the TIME_TICK signal conveys the arrival of a time tick, whereas the PLAYER_SHIP_MOVE signal conveys that the player wants to move the Ship. An event can also contain additional quantitative information about the occurrence in form of event parameters. For example, the PLAYER_SHIP_MOVE signal is accompanied by the parameters \((x, y)\) that contain the quantitative information as to where exactly to move the Ship.

In QP, events are represented as instances of the QEvent structure provided by the framework. Specifically, the QEvent structure contains the member \(\text{sig}\), to represent the signal of that event. Event parameters are added in the process of inheritance, as described in the sidebar “Single Inheritance in C.”

### SINGLE INHERITANCE IN C

Inheritance is the ability to derive new structures based on existing structures in order to reuse and organize code. You can implement single inheritance in C very simply by literally embedding the base structure as the first member of the derived structure. For example, Figure 1.10(A) shows the structure ScoreEvt derived from the base structure QEvent by embedding the QEvent instance as the first member of ScoreEvt. To make this idiom better stand out, I always name the base structure member super.

```c
typedef struct QEventTag {
    QSignal sig;
    . . .
} QEvent;

typedef struct ScoreEvtTag {
    QEvent super;
    uint16_t score;
} ScoreEvt;
```

Figure 1.10: (A) Derivation of structures in C, (B) memory alignment, and (C) the UML class diagram.
As shown in Figure 1.10(B), such nesting of structures always aligns the data member `super` at the beginning of every instance of the derived structure, which is actually guaranteed by the C standard. Specifically, WG14/N1124 Section 6.7.2.1.13 says: “... A pointer to a structure object, suitably converted, points to its initial member. There may be unnamed padding within a structure object, but not at its beginning” [ISO/IEC 9899:TC2]. The alignment lets you treat a pointer to the derived `ScoreEvt` structure as a pointer to the `QEvent` base structure. All this is legal, portable, and guaranteed by the C standard. Consequently, you can always safely pass a pointer to `ScoreEvt` to any C function that expects a pointer to `QEvent`. (To be strictly correct in C, you should explicitly cast this pointer. In OOP such casting is called upcasting and is always safe.) Therefore, all functions designed for the `QEvent` structure are automatically available to the `ScoreEvt` structure as well as other structures derived from `QEvent`. Figure 1.10(C) shows the UML class diagram depicting the inheritance relationship between `ScoreEvt` and `QEvent` structures.

QP uses single inheritance quite extensively not just for derivation of events with parameters, but also for derivation of state machines and active objects. Of course, the C++ version of QP uses the native C++ support for class inheritance rather than “derivation of structures.” You’ll see more examples of inheritance later in this chapter and throughout the book.

Because events are explicitly shared among most of the application components, it is convenient to declare them in the separate header file `game.h` shown in Listing 1.2. The explanation section immediately following the listing illuminates the interesting points.

### Listing 1.2 Signals, event structures, and active object interfaces defined in file `game.h`

```
enum GameSignals { /* signals used in the game */
    TIME_TICK_SIG = Q_USER_SIG, /* published from tick ISR */
    PLAYER_TRIGGER_SIG, /* published by Player (ISR) to trigger the Missile */
    PLAYER_QUIT_SIG,    /* published by Player (ISR) to quit the game */
    GAME_OVER_SIG,      /* published by Ship when it finishes exploding */
    /* insert other published signals here ... */
    MAX_PUB_SIG,        /* the last published signal */
    PLAYER_SHIP_MOVE_SIG, /* posted by Player (ISR) to the Ship to move it */
    BLINK_TIMEOUT_SIG,  /* signal for Tunnel’s blink timeout event */
    SCREEN_TIMEOUT_SIG, /* signal for Tunnel’s screen timeout event */
    TAKE_OFF_SIG,       /* from Tunnel to Ship to grant permission to take off */
    HIT_WALL_SIG,       /* from Tunnel to Ship when Ship hits the wall */
    HIT_MINE_SIG,       /* from Mine to Ship or Missile when it hits the mine */
    SHIP_IMG_SIG,       /* from Ship to the Tunnel to draw and check for hits */
    MISSILE_IMG_SIG,    /* from Missile the Tunnel to draw and check for hits */
    MINE_IMG_SIG,       /* sent by Mine to the Tunnel to draw the mine */
}
```
In QP, signals of events are simply enumerated constants. Placing all signals in a single enumeration is particularly convenient to avoid inadvertent overlap in the numerical values of different signals.

MISSILE_FIRE_SIG, /* sent by Ship to the Missile to fire */
DESTROYED_MINE_SIG, /* from Missile to Ship when Missile destroyed Mine */
EXPLOSION_SIG, /* from any exploding object to render the explosion */
MINE_PLANT_SIG, /* from Tunnel to the Mine to plant it */
MINE_DISABLED_SIG, /* from Mine to Tunnel when it becomes disabled */
MINE_RECYCLE_SIG, /* sent by Tunnel to Mine to recycle the mine */
SCORE_SIG, /* from Ship to Tunnel to adjust game level based on score */
/* insert other signals here ... */
MAX_SIG /* the last signal (keep always last) */
};

typedef struct ObjectPosEvtTag {
    QEvent super; /* extend the QEvent class */
    uint8_t x; /* the x-position of the object */
    uint8_t y; /* new y-position of the object */
} ObjectPosEvt;

typedef struct ObjectImageEvtTag {
    QEvent super; /* extend the QEvent class */
    uint8_t x; /* the x-position of the object */
    int8_t y; /* the y-position of the object */
    uint8_t bmp; /* the bitmap ID representing the object */
} ObjectImageEvt;

typedef struct MineEvtTag {
    QEvent super; /* extend the QEvent class */
    uint8_t id; /* the ID of the Mine */
} MineEvt;

typedef struct ScoreEvtTag {
    QEvent super; /* extend the QEvent class */
    uint16_t score; /* the current score */
} ScoreEvt;

/* opaque pointers to active objects in the application */
extern QAActive * const AO_Tunnel;
extern QAActive * const AO_Ship;
extern QAActive * const AO_Missile;

/* active objects' *constructors" */
void Tunnel_ctor(void);
void Ship_ctor(void);
void Missile_ctor(void);
(2) The application-level signals do not start from zero but rather are offset by the constant Q_USER_SIG. This is because QP reserves the lowest few signals for the internal use and provides the constant Q_USER_SIG as an offset from which user-level signals can start. Also note that by convention, I attach the suffix _SIG to all signals so that I can easily distinguish signals from other constants. I drop the suffix _SIG in the state diagrams to reduce the clutter.

(3) The constant MAX_PUB_SIG delimits the published signals from the rest. The publish-subscribe event delivery mechanism consumes some RAM, which is proportional to the number of published signals. I save some RAM by providing the lower limit of published signals to QP (MAX_PUB_SIG) rather than the maximum of all signals used in the application. (See also Listing 1.1(9)).

(4) The last enumeration MAX_SIG indicates the maximum of all signals used in the application.

(5) The event structure ObjectPosEvt defines a “class” of events that convey the object’s position on the display in the event parameters.

(6) The structure ObjectPosEvt derives from the base structure QEvent, as explained in the sidebar “Single Inheritance in C.”

(7,8) The structure ObjectPosEvt adds parameters x and y, which are coordinates of the object on the display.

### NOTE

Throughout this book I use the following standard exact-width integer types (WG14/N843 C99 Standard, Section 7.18.1.1) [ISO/IEC 9899:TC2]:

<table>
<thead>
<tr>
<th>Exact Size</th>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bits</td>
<td>uint8_t</td>
<td>int8_t</td>
</tr>
<tr>
<td>16-bits</td>
<td>uint16_t</td>
<td>int16_t</td>
</tr>
<tr>
<td>32-bits</td>
<td>uint32_t</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

If your (pre-standard) compiler does not provide the `<stdint.h>` header file, you can always typedef the exact-width integer types using the standard C data types such as signed/unsigned char, short, int, and long.
These global pointers represent active objects in the application and are used for posting events directly to active objects. Because the pointers can be initialized at compile time, I like to declare them `const`, so that they can be placed in ROM. The active object pointers are “opaque” because they cannot access the whole active object, only the part inherited from the `QActive` structure. I’ll have more to say about this in the next section.

These functions perform an early initialization of the active objects in the system. They play the role of static “constructors,” which in C you need to call explicitly, typically at the beginning of `main()`. (See also Listing 1.1 (10-12).)

### 1.6.1 Generating, Posting, and Publishing Events

The QF framework supports two types of asynchronous event exchange:

1. The simple mechanism of direct event posting supported through the functions `QActive_postFIFO()` and `QActive_postLIFO()`, where the producer of an event directly posts the event to the event queue of the consumer active object.

2. A more sophisticated publish-subscribe event delivery mechanism supported through the functions `QF_publish()` and `QActive_subscribe()`, where the producers of the events “publish” them to the framework, and the framework then delivers the events to all active objects that had “subscribed” to these events.

In QF, any part of the system, not necessarily only the active objects, can produce events. For example, interrupt service routines (ISRs) or device drivers can also produce events. On the other hand, only active objects can consume events, because only active objects have event queues.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
</table>

QF also provides “raw” thread-safe event queues (struct `QEQueue`), which can consume events as well. These “raw” thread-safe queues cannot block and are intended to deliver events to ISRs or device drivers. Refer to Chapter 7 for more details.

The most important characteristic of event management in QF is that the framework passes around only pointers to events, not the events themselves. QF never copies the
events by value ("zero-copy" policy); even in case of publishing events that often involves multicasting the same event to multiple subscribers. The actual event instances are either constant events statically allocated at compile time or dynamic events allocated at runtime from one of the event pools that the framework manages. Listing 1.3 provides examples of publishing static events and posting dynamic events from the ISRs of the "Fly ‘n’ Shoot” version for the Stellaris board (file `<qp>`\`\`\examples\`cortex-m3\`\vanilla\`iar\game-ev-lm3s811\bsp.c`). In Section 1.7.3 you will see other examples of event posting from active objects in the state machine code.

**Listing 1.3 Generating, posting, and publishing events from the ISRs in `bsp.c` for the Stellaris board**

```c
(1) void ISR_SysTick(void) {
    (2) static QEvent const tickEvt = { TIME_TICK_SIG, 0 };;
    (3) QF_publish(&tickEvt); /* publish the tick event to all subscribers */
    (4) QF_tick(); /* process all armed time events */
}
/*.....................................................................*/
(5) void ISR_ADC(void) {
    static uint32_t adcLPS = 0; /* Low-Pass-Filtered ADC reading */
    static uint32_t wheel = 0; /* the last wheel position */
    unsigned long tmp;
    ADCIntClear(ADC_BASE, 3); /* clear the ADC interrupt */
    ADCSequenceDataGet(ADC_BASE, 3, &tmp); /* read the data from the ADC */
    /* 1st order low-pass filter: time constant ~= 2^n samples */
    /* TF = (1/2^n)/(z-(2^n-1)/2^n)),
    * e.g., n=3, y(k+1) = y(k) - y(k)/8 + x(k)/8 => y += (x - y)/8 */
    (6) adcLPS += (((int)tmp - (int)adcLPS + 4) >> 3); /* Low-Pass-Filter */
    /* compute the next position of the wheel */
    (7) tmp = (((1 << 10) - adcLPS)*(BSP_SCREEN_HEIGHT - 2)) >> 10;
    if (tmp != wheel) { /* did the wheel position change? */
        (8) ObjectPosEvt * ope = Q_NEW(ObjectPosEvt, PLAYER_SHIP_MOVE_SIG);
        ope->x = (uint8_t)GAME_SHIP_X; /* x-position is fixed */
        ope->y = (uint8_t)tmp;
        (9) QActive_postFIFO(AO_ship, (QEvent *) ope); /* post to the Ship AO */
        wheel = tmp; /* save the last position of the wheel */
    }
    ...
}
```
In the case of the Stellaris board, the function `ISR_SysTick()` services the system clock tick ISR generated by the Cortex-M3 system tick timer.

The `TIME_TICK` event never changes, so it can be statically allocated just once. This event is declared as `const`, which means that it can be placed in ROM. The initializer list for this event consists of the signal `TIME_TICK_SIG` followed by zero. This zero informs the QF framework that this event is static and should never be recycled to an event pool.

The ISR calls the framework function `QF_publish()`, which takes the pointer to the `tickEvt` event to deliver to all subscribers.

The ISR calls the function `QF_tick()`, in which the framework manages the armed time events.

The function `ISR_ADC()` services the ADC conversions, which ultimately deliver the position of the Ship.

The ISR reads the data from the ADC.

A low-pass filter is applied to the raw ADC reading and the potentiometer wheel position is computed.

The QF macro `Q_NEW(ObjectPosEvt, PLAYER_SHIP_MOVE_SIG)` dynamically allocates an instance of the `ObjectPosEvt` event from an event pool managed by QF. The macro also performs the association between the signal `PLAYER_SHIP_MOVE_SIG` and the allocated event. The `Q_NEW()` macro returns the pointer to the allocated event.

The `PLAYER_SHIP_MOVE(x, y)` event is an example of an event with changing parameters. In general, such an event cannot be allocated statically (like the `TIME_TICK` event at label (2)) because it can change asynchronously next time the ISR executes. Some active objects in the system might still be referring to the event via a pointer, so the event should not be changing. Dynamic event allocation of QF solves all such concurrency issues because every time a new event is allocated. QF then recycles the dynamic events after it determines that all active objects are done with accessing the events.

The `x` and `y` parameters of the event are assigned.

The dynamic event is posted directly to the Ship active object.
1.7 Coding Hierarchical State Machines

Contrary to widespread misconceptions, you don’t need big design automation tools to translate hierarchical state machines (UML statecharts) into efficient and highly maintainable C or C++. This section explains how to hand-code the Ship state machine from Figure 1.6 with the help of the QF real-time framework and the QEP hierarchical processor, which is also part of the QP event-driven platform. Once you know how to code this state machine, you know how to code them all.

The source code for the Ship state machine is found in the file ship.c located either in the DOS version or the Stellaris version of the “Fly ‘n’ Shoot” game. I break the explanation of this file into three steps.

1.7.1 Step 1: Defining the Ship Structure

In the first step you define the Ship data structure. Just as in the case of events, you use inheritance to derive the Ship structure from the framework structure QActive (see the sidebar “Single Inheritance in C”). Creating this inheritance relationship ties the Ship structure to the QF framework.

The main responsibility of the QActive base structure is to store the information about the current active state of the state machine as well as the event queue and priority level of the Ship active object. In fact, QActive itself derives from a simpler QEP structure QHsm that represents just the current active state of a hierarchical state machine. On top of that information, almost every state machine must also store other “extended-state” information. For example, the Ship object is responsible for maintaining the Ship position as well as the score accumulated in the game. You supply this additional information by means of data members enlisted after the base structure member super, as shown in Listing 1.4.

Listing 1.4 Deriving the Ship structure in file ship.c

```c
#include "qp_port.h" /* the QP port */
#include "bsp.h" /* Board Support Package */
#include "game.h" /* this application */
/* local objects --------------------------------------------------------*/
typedef struct ShipTag {
    QActive super; /* derive from the QActive struct */
    uint8_t x; /* x-coordinate of the Ship position on the display */
    uint8_t y; /* y-coordinate of the Ship position on the display */
    uint8_t exp_ctr; /* explosion counter, used to animate explosions */
}船舶结构体;
```

Continued onto next page
Every application-level C file that uses the QP platform must include the `qp_port.h` header file.

The `bsp.h` header file contains the interface to the Board Support Package.

The `game.h` header file contains the declarations of events and other facilities shared among the components of the application (see Listing 1.2).

This structure defines the Ship active object.

```
(9)  uint16_t score;                     /* running score of the game */
(10) ) Ship;                            /* the typedef-ed name for the Ship struct */
(11) static QState Ship_active  (Ship *me, QEvent const *e);
(12) static QState Ship_parked  (Ship *me, QEvent const *e);
(13) static QState Ship_flying  (Ship *me, QEvent const *e);
(14) static QState Ship_exploding(Ship *me, QEvent const *e);
(15) static QState Ship_initial (Ship *me, QEvent const *e);
(16) static Ship l_ship;               /* the sole instance of the Ship active object */
```

```
/* state handler functions... */
```

```
(17) QActive * const AO_ship = (QActive *)&l_ship;   /* opaque pointer to Ship AO */
```

```
NOTE
```
I like to keep active objects, and indeed all state machine objects (such as Mines), strictly encapsulated. Therefore, I don’t put the state machine structure definitions in header files; rather, I define them right in the implementation file, such as `ship.c`. That way I can be sure that the internal data members of the Ship structure are not known to any other parts of the application.

```
(5)  The Ship active object structure derives from the framework structure QActive, as described in the sidebar “Single Inheritance in C.”
```

```
(6,7) The x and y data members represent the position of the Ship on the display.
```

```
(8)  The exp_ctr member is used for pacing the explosion animation (see also the “exploding” state in the Ship state diagram in Figure 1.6).
```

```
(9)  The score member stores the accumulated score in the game.
```

www.newnespress.com
(10) I use the typedef to define the shorter name Ship equivalent to struct ShipTag.

(11-14) These four functions are called state-handler functions because they correspond one to one to the states of the Ship state machine shown in Figure 1.6. For example, the Ship_active() function represents the “active” state. The QEP event processor calls the state-handler functions to realize the UML semantics of state machine execution. All state-handler functions have the same signature. A state-handler function takes the state machine pointer and the event pointer as arguments and returns the status of the operation back to the event processor—for example whether the event was handled or not. The return type QState of state-handler functions is typedef-ed to uint8_t as QState in the header file <qp>\qpc\include\qep.h.

**NOTE**

I use a simple naming convention to strengthen the association between the structures and the functions designed to operate on these structures. First, I name the functions by combining the typedef’ed structure name with the name of the operation (e.g., Ship_active). Second, I always place the pointer to the structure as the first argument of the associated function, and I always name this argument “me” (e.g., Ship_active(Ship *me, ...)).

(15) In addition to state-handler functions, every state machine must declare the initial pseudostate, which QEP invokes to execute the topmost initial transition (see Figure 1.6(1)). The initial pseudostate handler has a signature identical to the regular state-handler function.

(16) In this line I statically allocate the storage for the Ship active object. Note that the object l_ship is defined as static so that it is accessible only locally at the file scope of the ship.c file.

(17) In this line I define and initialize the global pointer AO_Ship to the Ship active object (see also Listing 1.2(10)). This pointer is “opaque” because it treats the Ship object as the generic QActive base structure rather than the specific Ship structure. The power of an “opaque” pointer is that it allows me to completely hide the definition of the Ship structure and make it inaccessible to the rest of the application. Still, the other application components can access the Ship object to post events directly to it via the QActive_postFIFO(QActive *me, QEvent const *e) function.
1.7.2 Step 2: Initializing the State Machine

The state machine initialization is divided into the following two steps for increased flexibility and better control of the initialization timeline:

1. The state machine “constructor”; and
2. The top-most initial transition.

The state machine “constructor,” such as Ship_ctor(), intentionally does not execute the topmost initial transition defined in the initial pseudostate because at that time some vital objects can be missing and critical hardware might not be properly initialized yet. Instead, the state machine “constructor” merely puts the state machine in the initial pseudostate. Later, the user code must trigger the topmost initial transition explicitly, which happens actually inside the function QActive_start() (see Listing 1.1(18-20)). Listing 1.5 shows the instantiation (the “constructor” function) and initialization (the initial pseudostate) of the Ship active object.

Listing 1.5 Instantiation and initialization of the Ship active object in ship.c

```
(1) void Ship_ctor(void) { /* instantiation */
(2)    Ship *me = &l_ship;
(3)    QActive_ctor(&me->super, (QStateHandler)&Ship_initial);
(4)    me->x = GAME_SHIP_X;
(5)    me->y = GAME_SHIP_Y;
    } /*......................................................................*/
(6) QState Ship_initial(Ship *me, QEvent const *e) { /* initialization */
(7)    QActive_subscribe((QActive *)me, TIME_TICK_SIG);
(8)    QActive_subscribe((QActive *)me, PLAYER_TRIGGER_SIG);
(9)    return Q_TRAN(&Ship_active); /* top-most initial transition */
    }
```

(1) The global function Ship_ctor() is prototyped in game.h and called at the beginning of main().

(2) The “me” pointer points to the statically allocated Ship object (see Listing 1.4(16)).

---

3 In C++, the static constructors run even before main().
Every derived structure is responsible for initializing the part inherited from the base structure. The “constructor” QActive_ctor() puts the state machine in the initial pseudostate &Ship_initial (see Listing 1.4(15)).

The Ship position is initialized.

The Ship_initial() function defines the topmost initial transition in the Ship state machine (see Figure 1.6(1)).

The Ship active object subscribes to signals TIME_TICK_SIG and PLAYER_TRIGGER_SIG, as specified in the state diagram in Figure 1.6(1).

The initial state “active” is specified by returning the QP macro Q_TRAN().

NOTE
The macro Q_TRAN() must always follow the return statement.

1.7.3 Step 3: Defining State-Handler Functions

In the last step, you actually code the Ship state machine by implementing one state at a time as a state-handler function. To determine what elements belong to any given state-handler function, you follow around the state’s boundary in the diagram (Figure 1.6). You need to implement all transitions originating at the boundary, any entry and exit actions defined in the state, and all internal transitions enlisted directly in the state. Additionally, if there is an initial transition embedded directly in the state, you need to implement it as well.

Take for example the state “flying” shown in Figure 1.6. This state has an entry action and two transitions originating at its boundary: HIT_WALL and HIT_MINE(type) as well as three internal transitions TIME_TICK, PLAYER_TRIGGER, and DESTROYED_MINE(score). The “flying” state nests inside the “active” superstate.

Listing 1.6 shows two state-handler functions of the Ship state machine from Figure 1.6. The state-handler functions correspond to the states “active” and “flying,” respectively. The explanation section immediately following the listing highlights the important implementation techniques.
Listing 1.6 State-handler functions for states “active” and “flying” in ship.c

(1) QState Ship_active(Ship *me, QEvent const *e) {
    switch (e->sig) {
(2)        case Q_INIT_SIG: { /* nested initial transition */
(3)            /* any actions associated with the initial transition */
(4)            return Q_TRAN(&Ship_parked);
(5)        }
(6)        case PLAYER_SHIP_MOVE_SIG: {
(7)            me->x = ((ObjectPosEvt const *)e)->x;
(8)            me->y = ((ObjectPosEvt const *)e)->y;
(9)            return Q_HANDLED();
    }
(10)    return Q_SUPER(&QHsm_top); /* return the superstate */
    }
/*.....................................................................*/
QState Ship_flying(Ship *me, QEvent const *e) {
    switch (e->sig) {
(11)        case Q_ENTRY_SIG: {
(12)            ScoreEvt *sev;
(13)            me->score = 0; /* reset the score */
(14)            sev = Q_NEW(ScoreEvt, SCORE_SIG);
(15)            sev->score = me->score;
(16)            QActive_postFIFO(AO_Tunnel, (QEvent *)sev);
            return Q_HANDLED();
        }
(17)        case TIME_TICK_SIG: { /* tell the Tunnel to draw the Ship and test for hits */
(18)            ObjectImageEvt *oie = Q_NEW(ObjectImageEvt, SHIP_IMG_SIG);
(19)            oie->x = me->x;
(20)            oie->y = me->y;
(21)            oie->bmp = SHIP_BMP;
(22)            QActive_postFIFO(AO_Tunnel, (QEvent *)oie);
(23)            ++me->score; /* increment the score for surviving another tick */
(24)            if ((me->score % 10) == 0) { /* is the score "round"? */
(25)                ScoreEvt *sev = Q_NEW(ScoreEvt, SCORE_SIG);
(26)                sev->score = me->score;
(27)                QActive_postFIFO(AO_Tunnel, (QEvent *)sev);
            }
            return Q_HANDLED();
        }
(28)        case PLAYER_TRIGGER_SIG: { /* trigger the Missile */
(29)            ObjectPosEvt *ope = Q_NEW(ObjectPosEvt, MISSILE_FIRE_SIG);
(30)            ope->x = me->x;
(31)            ope->y = me->y + SHIP_HEIGHT - 1;
Each state handler must have the same signature, that is, it must take two parameters: the state machine pointer “me” and the pointer to QEvent. The keyword const before the * in the event pointer declaration means that the event pointed to by that pointer cannot be changed inside the state-handler function (i.e., the event is read-only). A state-handler function must return QState, which conveys the status of the event handling to the QEP event processor.

Typically, every state handler is structured as a switch statement that discriminates based on the signal of the event e->sig.

This line of code pertains to the nested initial transition Figure 1.6(2). QEP provides a reserved signal Q_INIT_SIG that the framework passes to the state-handler function when it wants to execute the initial transition.

You can enlist any actions associated with this initial transition (none in this particular case).

You designate the target substate with the Q_TRAN() macro. This macro must always follow the return statement, through which the state-handler function informs the QEP event processor that the transition has been taken.
The initial transition must necessarily target a direct or transitive substate of a given state. An initial transition cannot target a peer state or go up in state hierarchy to higher-level states, which in the UML would represent a “malformed” state machine.

(6) This line of code pertains to the internal transition `PLAYER_SHIP_MOVE_SIG (x, y)` in Figure 1.6(3).

(7,8) You access the data members of the Ship state machine via the “me” argument of the state-handler function. You access the event parameters via the “e” argument. You need to cast the event pointer from the generic `QEvent` base class to the specific event structure expected for the `PLAYER_SHIP_MOVE_SIG`, which is `ObjectPosEvt` in this case.

The association between the event signal and event structure (event parameters) is established at the time the event is generated. All recipients of that event must know about this association to perform the cast to the correct event structure.

(9) You terminate the case statement with “`return QHandled ()`”, which informs the QEP processor that the event has been handled but no transition has been taken.

(10) The final `return` from a state-handler function designates the `superstate` of that state by means of the QEP macro `Q_SUPER()`. The final `return` statement from a state-handler function represents the single point of maintenance for changing the nesting level of a given state. The state “active” in Figure 1.6 has no explicit superstate, which means that it is implicitly nested in the “top” state. The “top” state is a UML concept that denotes the ultimate root of the state hierarchy in a hierarchical state machine. QEP provides the “top” state as a state-handler function `QHsm_top()`, and therefore the `Ship_active()` state handler uses the pointer `&QHsm_top` as the argument of the macro `Q_SUPER()`.
NOTE

In C and C++, a pointer-to-function $\text{QHsm}\_\text{top}()$ can be written either as $\text{QHsm}\_\text{top}$ or $\&\text{QHsm}\_\text{top}$. Even though the notation $\text{QHsm}\_\text{top}$ is more succinct, I prefer adding the ampersand explicitly, to leave absolutely no doubt that I mean a pointer-to-function $\&\text{QHsm}\_\text{top}$.

(11) This line of code pertains to the entry action into state “flying” (Figure 1.6(5)). QEP provides a reserved signal $\text{Q\_ENTRY\_SIG}$ that the framework passes to the state-handler function when it wants to execute the entry actions.

(12) The entry action to “flying” posts the $\text{SCORE}$ event to the Tunnel active object (Figure 1.6(5)). This line defines a temporary pointer to the event structure $\text{ScoreEvt}$.

(13) The QF macro $\text{Q\_NEW}(\text{ScoreEvt, \text{SCORE\_SIG}})$ dynamically allocates an instance of the $\text{ScoreEvt}$ from an event pool managed by QF. The macro also performs the association between the signal $\text{SCORE\_SIG}$ and the allocated event. The $\text{Q\_NEW}()$ macro returns the pointer to the allocated event.

(14) The $\text{score}$ parameter of the $\text{ScoreEvt}$ is set from the state machine member $\text{me}\rightarrow\text{score}$.

(15) The $\text{sev}$ event is posted directly to the Tunnel active object by means of the QP function $\text{QActive\_postFIFO}()$. The arguments of this function are the recipient active object ($\text{AO\_Tunnel}$ in this case) and the pointer to the event (the temporary pointer $\text{sev}$ in this case).

(16) You terminate the case statement with $\text{return Q\_Handled}()$, which informs QEP that the entry actions have been handled.

(17,18) These two lines of code pertain to the state transitions from “flying” to “exploding” (Figure 1.6(9, 10)).

(19) You can enlist any actions associated with the transition (none in this particular case).

(20) You designate the target of the transition with the $\text{Q\_TRAN}()$ macro.
The final return from a state-handler function designates the \textit{superstate} of that state. The state “flying” in Figure 1.6 nests in the state “active,” so the state handler \texttt{Ship\_flying()} returns the pointer \texttt{&Ship\_active}.

When implementing state-handler functions you need to keep in mind that the QEP event processor is in charge here rather than your code. QEP will invoke a state-handler function for various reasons: for hierarchical event processing, for execution of entry and exit actions, for triggering initial transitions, or even just to elicit the superstate of a given state handler. Therefore, you should not assume that a state handler would be invoked only for processing signals enlisted in the \texttt{case} statements. You should avoid any code outside the \texttt{switch} statement, especially code that would have side effects.

\section*{1.8 The Execution Model}

As you saw in Listing 1.1, the \texttt{main()} function eventually gives control to the event-driven framework by calling \texttt{QF\_run()} to execute the application. In this section, I briefly explain how QF allocates the CPU cycles to various tasks within the system and what options you have in choosing the execution model.

\subsection*{1.8.1 Simple Nonpreemptive “Vanilla” Scheduler}

The “Fly ‘n’ Shoot” example uses the simplest QF configuration, in which QF runs on a bare-metal target processor without any underlying operating system or kernel.\textsuperscript{4} I call such a QF configuration “plain vanilla” or just “vanilla.”

QF includes a simple nonpreemptive “vanilla” kernel, which executes one active object at a time in the infinite loop (similar to the “superloop”). The “vanilla” kernel is engaged after each event is processed in the run-to-completion (RTC) fashion to choose the next highest-priority active object ready to process the next event. The “vanilla” scheduler is cooperative, which means that all active objects cooperate to share a single CPU and implicitly yield to each other after every RTC step. The kernel is nonpreemptive, meaning that every active object must completely process an event before any other active object can start processing another event.

\footnote{The 80×86 version of the “Fly ‘n’ Shoot” game runs on top of DOS, but DOS does not provide any multitasking support.}
The ISRs can preempt the execution of active objects at any time, but due to the simplistic nature of the “vanilla” kernel, every ISR returns to exactly the preemption point. If the ISR posts or publishes an event to any active object, the processing of this event won’t start until the current RTC step completes. The maximum time an event for the highest-priority active object can be delayed this way is called the task-level response. With the nonpreemptive “vanilla” kernel, the task-level response is equal to the longest RTC step of all active objects in the system. Note that the task-level response of the “vanilla” kernel is still a lot better than the traditional “superloop” (a.k.a. main+ISRs) architecture. I’ll have more to say about this in the upcoming Section 1.9, where I compare the event-driven “Fly ‘n’ Shoot” example to the traditionally structured Quickstart application.

The task-level response of the simple “vanilla” kernel turns out to be adequate for surprisingly many applications because state machines by nature handle events quickly without a need to busy-wait for events. (A state machine simply runs to completion and becomes dormant until another event arrives.) Also note that often you can make the task-level response as fast as you need by breaking up longer RTC steps into shorter ones (e.g., by using the “Reminder” state pattern described in Chapter 5).

1.8.2 The QK Preemptive Kernel

In some cases, breaking up long RTC steps into short enough pieces might be very difficult, and consequently the task-level response of the nonpreemptive “vanilla” kernel might be too long. An example system could be a GPS receiver. Such a receiver performs a lot of floating-point number crunching on a fixed-point CPU to calculate the GPS position. At the same time, the GPS receiver must track the GPS satellite signals, which involves closing control loops in submillisecond intervals. It turns out that it’s not easy to break up the position-fix computation into short enough RTC steps to allow reliable signal tracking.

But the RTC semantics of state machine execution do not mean that a state machine has to monopolize the CPU for the duration of the RTC step. A preemptive kernel can perform a context switch in the middle of the long RTC step to allow a higher-priority active object to run. As long as the active objects don’t share resources, they can run concurrently and complete their RTC steps independently (see also Section 6.3.3 in Chapter 6).

The QP event-driven platform includes a tiny, fully preemptive, priority-based real-time kernel component called QK, which is specifically designed for processing
events in the RTC fashion. Configuring QP to use the preemptive QK kernel is very easy, but as with any fully preemptive kernel you must be very careful with any resources shared among active objects. The “Fly ‘n’ Shoot” example has been purposely designed to avoid any resource sharing among active objects, so the application code does not need to change at all to run on top of the QK preemptive kernel, or any other preemptive kernel or RTOS for that matter. The accompanying code contains the “Fly ‘n’ Shoot” example with QK in the following directory: 
<qp>\qpc\examples\80x86\qk\tcpp101\l\game\. You can execute this example in a DOS-console on any standard Windows-based PC.

1.8.3 Traditional OS/RTOS

QP can also work with a traditional operating system (OS), such as Windows or Linux, or virtually any real-time operating system (RTOS) to take advantage of the existing device drivers, communication stacks, and other middleware.

QP contains a platform abstraction layer (PAL), which makes adapting QP to virtually any operating system easy. The carefully designed PAL allows tight integration with the underlying OS/RTOS by reusing any provided facilities for interrupt management, message queues, and memory partitions. I cover porting QP in Chapter 8.

1.9 Comparison to the Traditional Approach

The “Fly ‘n’ Shoot” game behaves intentionally almost identically to the Quickstart application provided in source code with the Luminary Micro Stellaris EV-LM3S811 evaluation kit [Luminary 06]. In this section I’d like to compare the traditional approach represented by the Quickstart application with the state machine-based solution exemplified in the “Fly ‘n’ Shoot” game.

Figure 1.11(A) shows schematically the flowchart of the Quickstart application; Figure 1.11(B) shows the flowchart of the “Fly ‘n’ Shoot” game running on top of the cooperative “vanilla” kernel. At the highest level, the flowcharts are similar in that they both consist of an endless loop surrounding the entire processing. But the internal structure of the main loop is very different in the two cases. As indicated by the heavy

---

5 QK provides a mutex facility for enforcing a mutually exclusive access to shared resources. The QK mutex uses the priority-ceiling protocol to avoid priority inversions. Refer to Chapter 10 for more information.
lines in the flowcharts, the Quickstart application spends most of its time in the tight “event loops” designed to busy-wait for certain events, such as the screen update event. In contrast, the “Fly ‘n’ Shoot” application spends most of its time right in the main loop. The QP framework dispatches any available event to the appropriate state machine that handles the event and returns quickly to the main loop without ever waiting for events internally.

The Quickstart application has much more convoluted flow of control than the “Fly ‘n’ Shoot” example because the traditional solution is very specific to the problem at hand whereas the state-machine approach is generic. The Quickstart application is structured very much like a traditional sequential program that tries to stay in control
from the beginning to the end. From time to time, the application pauses to busy-wait for a certain event, whereas the code is generally not ready to handle any other events than the one it chooses to wait for. All this contributes to the inflexibility of the design. Adding new events is hard because the whole structure of the intervening code is designed to accept only very specific events and would need to change dramatically to accommodate new events. Also, while busy-waiting for the screen update event (equivalent to the `TIME_TICK` event in “Fly ‘n’ Shoot” example), the application is really not responsive to any other events. The task-level response is hard to characterize and generally depends on the event type. The timing established by the hard-coded waiting for the existing events might not work well for new events.

In contrast, the “Fly ‘n’ Shoot” application has a much simpler control flow that is purely event-driven and completely generic (see Figure 1.11(B)). The context of each active object component is represented as the current state of a state machine, rather than as a certain place in the code. That way, hanging in tight “event loops” around certain locations in the code corresponding to the current context is unnecessary. Instead, a state machine remembers the context very efficiently as a small data item (the state-variable; see Chapter 3). After processing of each event, the state machine can return to the common event loop that is designed generically to handle all kinds of events. For every event, the state machine naturally picks up where it left off and moves on to the next state, if necessary. Adding new events is easy in this design because a state machine is responsive to any event at any time. An event-driven, state-machine-based application is incomparably more flexible and resilient to change than the traditional one.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>The generic event loop can also very easily detect the situation when no events are available, in which case the QP framework calls the <code>QF_onIdle()</code> function (see Figure 1.11(B)). This callback function is designed to be customized by the application and is the ideal place to put the CPU in a low-power sleep mode to conserve the power. In contrast, the traditional approach does not offer any single place to transition to the low-power sleep mode and consequently is much less friendly for implementing truly low-power designs.</td>
</tr>
</tbody>
</table>

### 1.10 Summary

If you’ve never done event-driven programming before, the internal structure of the “Fly ‘n’ Shoot” game must certainly represent a big paradigm shift for you. In fact,
I hope that it actually blows your mind, because otherwise I’m not sure that you really appreciate the complete reversal of control of an event-driven program compared to the traditional sequential code. This reversal of control, known as the Hollywood Principle (don’t call us, we’ll call you), baffles many newcomers, who often find it “mind-boggling,” “backward,” or “weird.”

The “Fly ‘n’ Shoot” game is by no means a big application, but at the same time it is definitely not trivial, either. You shouldn’t worry if you don’t fully understand it at the first reading. In the upcoming chapters, I will provide a closer look at the state machine design and coding techniques. In Part II, I discuss the features, implementation, and porting of the QF real-time framework.

My main goal in this chapter was just to introduce you to the event-driven paradigm and the modern state machines to convince you that these powerful concepts aren’t particularly hard to implement directly in C or C++. Indeed, I hope you noticed that the actual coding of the nontrivial “Fly ‘n’ Shoot” game wasn’t a big deal at all. All you needed to know was just a few cookie-cutter rules for coding state machines and familiarity with a few framework services for implementing the actions.

While the coding turned out to be essentially a nonissue; the bulk of the programming effort was spent on the design of the application. At this point, I hope that the “Fly ‘n’ Shoot” example helps you get the big picture of how the method works. Under the event-driven model, the program structure is divided into two rough groups: events and state machine components (active objects). An event represents the occurrence of something interesting. A state machine codifies the reactions to the events, which generally depend both on the nature of the event and on the state of the component. While events often originate from the outside of your program, such as time ticks or button presses in the “Fly ‘n’ Shoot” game, events can also be generated internally by the program itself. For example, the Mine components generate notification events when they detect a collision with the Missile or the Ship.

An event-driven program executes by constantly checking for possible events and, when an event is detected, dispatching the event to the appropriate state machine component (see Figure 1.11(B)). For this approach to work, the events must be checked continuously and frequently. This implies that the state machines must execute quickly so that the program can get back to checking for events. To meet this requirement, a state machine cannot go into a condition where it is busy-waiting for some long or indeterminate time. The most common example of this would be a while loop inside a state-handler function, where the condition for termination was not under program...
control—for instance, the button press. This kind of program structure, an indefinite loop, is referred to as “blocking” code,\(^6\) and you saw examples of it in the Quickstart application (see Figure 1.11(A)). For the event-driven programming model to work, you must only write “nonblocking” code [Carrier 05].

Finally, the “Fly ‘n’ Shoot” example demonstrates the use of the event-driven platform called QP, which is a collection of components for building event-driven applications. The QF real-time framework component embodies the Hollywood Principle by calling the application code, not the other way around. Such an arrangement is very typical for event-driven systems and application frameworks similar to QF are at the heart of virtually every design automation tool on the market today.

The QF framework operates in the “Fly ‘n’ Shoot” game in its simplest configuration, in which QF runs on a bare-metal target processor without any operating system. QF can also be configured to work with the build-in preemptive real-time kernel called QK (see Chapter 10) or can be easily ported to almost any traditional OS or RTOS (see Chapter 8). In fact, you can view the QF framework itself as a high-level, event-driven, real-time operating system.

---

\(^6\) In the context of a multitasking operating system the “blocking” code corresponds to waiting on a semaphore, event flag, message mailbox, or other such operating system primitive.