Updates and Errata

Document Revision C
for QP version 4.4.00
February 2012

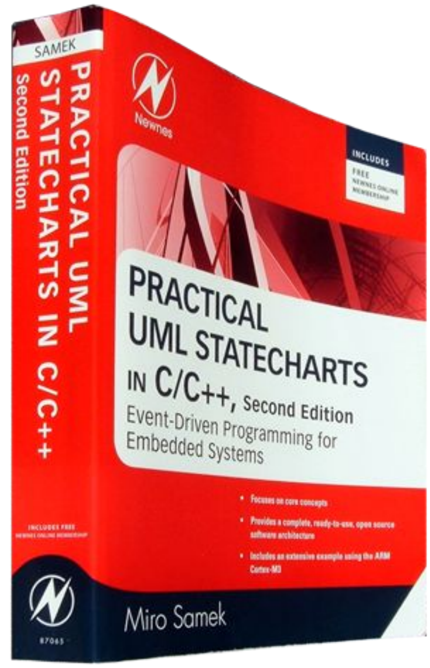
PRACTICAL UML STATECHARTS IN C/C++, Second Edition

Event-Driven Programming for Embedded Systems

Miro Samek

Index of Corrections and Updates

NOTE: The following index uses the section numbering consistent with the book. *Practical UML Statecharts in C/C++, Second Edition* [PSiCC2]. The **updated** sections are highlighted and marked with an exclamation point (!).



Introduction

This document contains updates and errata to the book:

Pracical UML Statecharts in C/C++, Second Edition: Event-driven Programming for Embedded Systems

By Miro Samek

Newnes, 2008

ISBN-10: 0750687061
ISBN-13: 978-0750687065

Companion Website: state-machine.com/psicc2

The updates cover the QP frameworks version **4.4.00**.

NOTE: The following sections are numbered consistently with the book. The updated sections are highlighted and marked with a (!).

Back Cover

Location	Is	Should be
First Bullet, "Understand State Machine Concepts"	From traditional finite state automated to modern UML state machines	From traditional finite state **automata** to modern UML state machines

Preface

Page xxi, 5th paragraph	Corterx-M3	Cortex-M3
Page xxi, 5th paragraph	form Luminary Micro	from Luminary Micro
Page xxi, last paragraph	Max OS X	Mac OS X
Page xxi, footnote 2	EKIEV-LM3S811	EKI-LM3S811

PART I UML STATE MACHINES

Location	Is	Should be
Page 2, 1st paragraph	catalogue	catalog

CHAPTER 1 Getting Started with UML State Machines and Event-Driven Programming

1.2 Let's Play

1.2.2 Running the Stellaris Version

Page 9, 6th paragraph	game-ev-lm3s811.eww	game.eww

1.3 The main() function

Page 13, Listing 1.1, explanation section (3)	Section 1.7	Section 1.6
Page 15, Listing 1.1 explanation section (17)	(17) The function `QF_poolInit()` initializes...	(17) The function `QF_psInit()` initializes...

1.4 The Design of the “Fly 'n' Shoot” Game

Page 18, Figure 1.4(13)	**HIT**_MINE(**type**)	**DESTROYED**_MINE(**score**)
Page 18, Figure 1.4(14)	DESTROYED_MINE(**type**)	DESTROYED_MINE(**score**)
Page 20, Figure 1.4 explanation section (12)	… Missile posts the `MISSILE_IMG(x, y, bmp)` event to Table.	… Missile posts the `MISSILE_IMG(x, y, bmp)` event to Tunnel.
Page 20, Figure 1.4 explanation section (13)	Table renders the Missile bitmap… `HIT_MINE(score)` ...	Tunnel renders the Missile bitmap… `DESTROYED_MINE(score)` ...
Page 20, Figure 1.4 explanation section (14)	`HIT_MINE(`**`type`**`)`.	`DESTROYED_MINE(`**`score`**`)`

1.5 Active Objects in the “Fly 'n' Shoot” Game

Page 20, last paragraph	... actions performed by an active object depend as much on the events it receives as on the internal mode of the object.	... actions performed by an active object depend as much on the internal mode of the object as on the events it receives.

1.5.1 The Missile Active Object

Location	Is	Should be
Page 22, Figure 1.5(5)	`HIT_MINE(score)`	`DESTROYED_MINE(score)`

1.5.2 The Ship Active Object

Page 24, last paragraph	argumentation	reasoning
Page 25, Figure 1.6(12)	`QActive_postFIFO(Table, ...`	`QActive_postFIFO(`**`Tunnel`**`, ...`
Page 26, Figure 1.6 explanation section (7)	The `PLAYER_TRIIGGER` internal transition…	The `PLAYER_TRIGGER` internal transition…
Page 26, Figure 1.6 explanation section (8)	… The score is not posted to the Table at this point.	… The score is not posted to the Tunnel at this point.

1.5.3 The Mine Components

Page 30, Figure 1.9 explanation section (7)	The exit action in the "used" state posts the `MINE_DISABLDED(mine_id)` event to the Tunnel active object… (see also Figure 1.9(4))... Note that generating the `MINE_DISABLDED(mine_id)` event in the exit section from "used" …	The exit action in the "used" state posts the `MINE_DISABLED(mine_id)` event to the Tunnel active object… (see also Figure 1.7(4))... Note that generating the `MINE_DISABLED(mine_id)` event in the exit section from "used" …

Page 31, Fig 1.9	The internal transition `TIME_TICK` in state "used" is:
	
	`TIME_TICK [me->x +`
	`GAME_MISSILE_SPEED_X`
	`<`
	`GAME_SCREEN_WIDTH] /`
	`me->x +=`
	`GAME_MISSILE_SPEED_X;`
	`postFIFO(Tunnel,`
	`MISSILE_IMG(me->x,`
	`me->y,`
	`MISSILE_BMP));`
		The internal transition `TIME_TICK` in state "used" should be:
		
		`TIME_TICK [me->x >=`
		`GAME_SPEED_X] /`
		`me->x -= GAME_SPEED_X;`
		`postFIFO(Tunnel,`
		`MISSILE_IMG(me->x,`
		`me->y,`
		`MINE2_BMP));`
		
		(see also state diagram below)

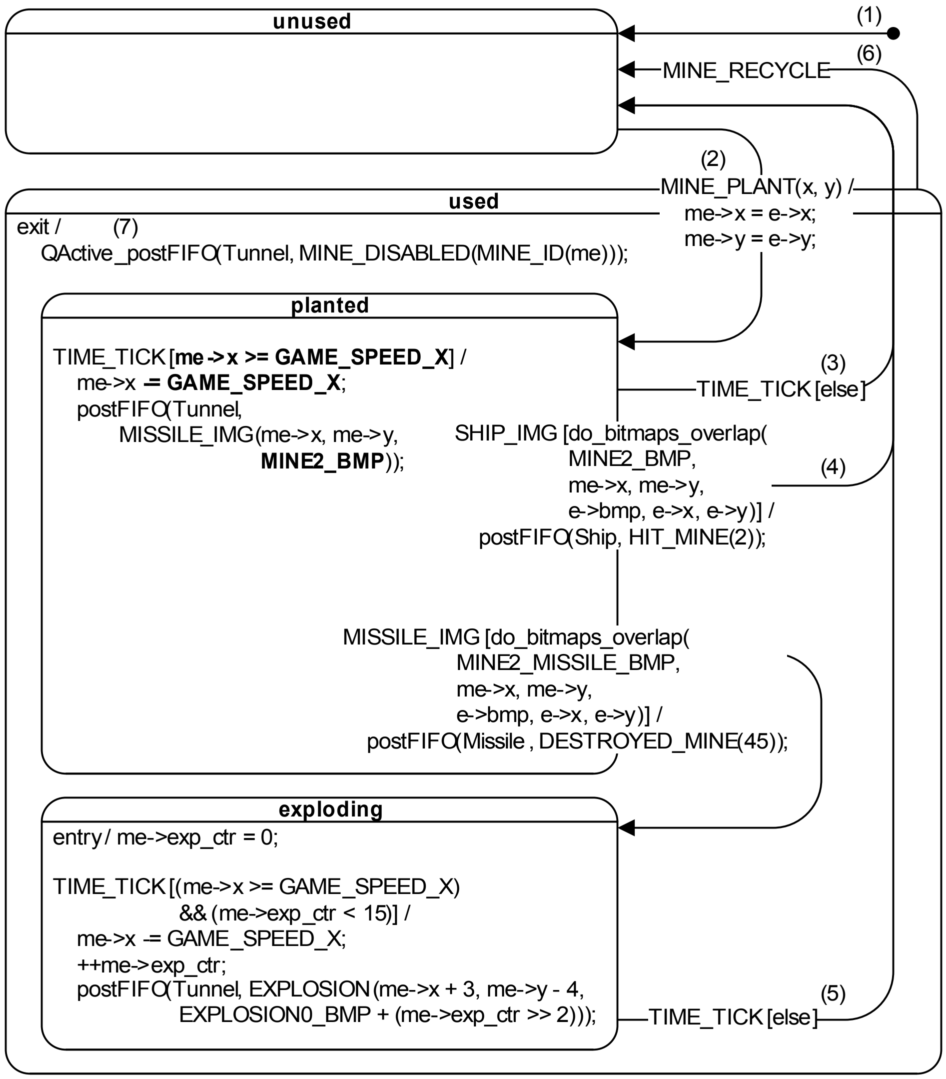


Figure 1.9: Mine2 state machine diagram.

1.6 Events in the “Fly 'n' Shoot” Game

Location	Is	Should be
Page 33, Listing 1.2	`/* From Missile the Tunnel . . . */`	`/* From Missile to the Tunnel . . . */`

1.7 Coding Hierarchical State Machines

1.7.3 Defining State-Handler Functions

Location	Is	Should be
Page 46, Listing 1.6 explanation section (9)	`return QHandled()`	`return Q_HANDLED()`
Page 47, Listing 1.6 explanation section (16)	`return QHandled()`	`return Q_HANDLED()`

1.10 Summary

Page 53, 4th paragraph	Wile the coding …	While the coding …
Page 54, last paragraph	build-in	built-in

CHAPTER 2 A Crash Course in UML State Machines

2.2 Basic State Machine Concepts

2.2.1 States

Location	Is	Should be
Page 60, footnote 2	Ignore at this print...	Ignore at this point...

2.2.5 Guard Conditions

Page 65, end of second paragraph	… whole new column in the table...	...whole new row in the table..
Page 66, begin of second paragraph	Capturing behavior as the quantitative "state" has ...	Capturing behavior as the qualitative "state" has ...
Page 66, middle of third paragraph	This example points to the main weakness of the quantitative "state",...	This example points to the main weakness of the qualitative "state",...

2.3.15 UML State Machine Semantics: An Exhaustive Example

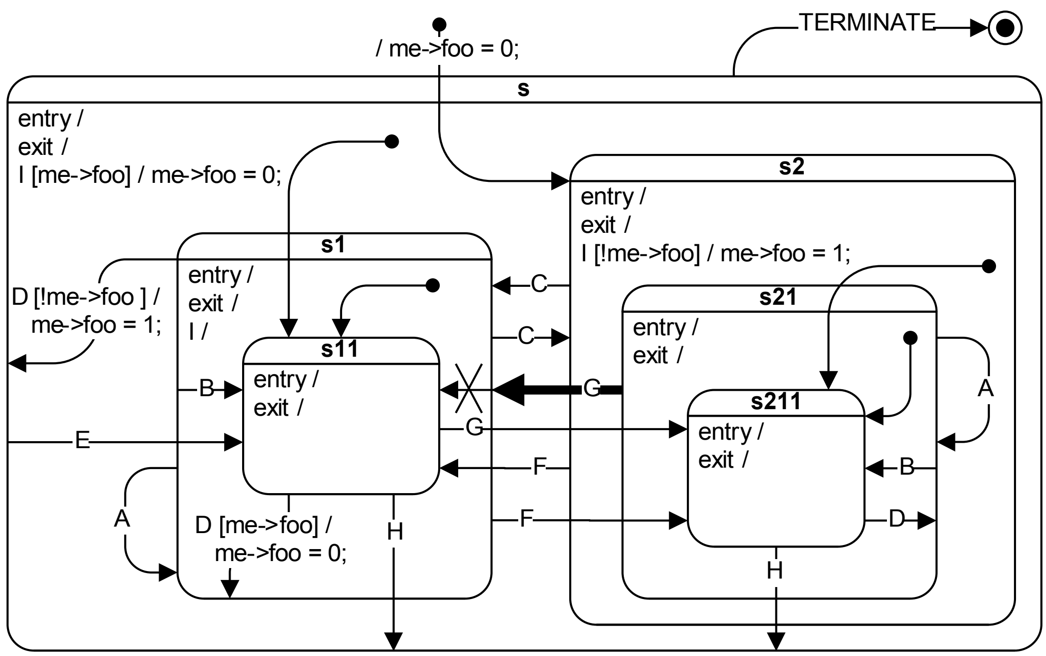


Figure 2.11: Hypothetical state machine that contains all possible state transition topologies up to four levels of state nesting.

Location	Is	Should be
Page 89, first line	...UML sate machines...	...UML state machines...
Page 89, 1st paragraph	...defined in the direct target state "**s21**."	...defined in the direct target state "**s1**."
Page 90, 1st paragraph, 4th line	...which is a test of `me->foo` against **0**,...	...which is a test of `me->foo` against **1**,...
Page 90, 4th paragraph, 2nd line	States "s2" and d both ...	States "s2" and "s" both ...

2.4.2 High-Level Design

Page 93, In TIP	OPER (operand)	OPER (operator)
Page 94, footnote 10	factorize out	factor out

2.4.6 Final Touches

Page 96, Section 2.4.6, end of paragraph	The actual C implementation...	The actual C++ implementation...

CHAPTER 3 Standard State Machine Implementations

3.1 The Time Bomb Example

3.1.1 Executing the Example Code

Location	Is	Should be
Page 104, 1st paragraph, 2 occurrences	tcpp101\bomb	tcpp101\l\bomb
Page 104, 2nd paragraph	Section 1.1	Section 1.2.1

3.3.3 Variations of the Technique

Page 113, 6th paragraph	qpc\examples\cortex-m3\dos\iar\game\bsp.c	qpc\examples\cortex-m3\vanilla\iar\game-ev-lm3s811\bsp.c

3.4.1 Generic State-Table Event Processor

Page 116, Listing 3.2 explanation section (3)	This `typedef` defines Tran type as a pointer to the `StateTable` struct and a pointer to the Event struct as arguments and returns `uint8_t`. The value returned from the transition function represents the next state for the state machine after executing the transition…	This `typedef` defines `Tran` type as a pointer to the `StateTable` struct and a pointer to the Event struct as arguments and returns `void`.…

3.4.2 Application-Specific Code

Page 121, Listing 3.4 explanation section (17)	The sate table …	The state table …

3.5 Object-Oriented State Design Pattern

Page 125, Figure 3.6, to immediate right of “Bomb 3” class	state_	state

3.5.3 Variations of the Technique

Location	Is	Should be
Page 132, Figure 3.8, to immediate right of the “Bomb” class	state_	state

3.6.1 Generic QEP Event Processor

Page 137, Listing 3.7(7)	`(void)(*me->state)(me, &QEP_reservedEvt_[`**`Q_EXIT_SIG`**`])`	`(void)(*me->state)(me, &QEP_reservedEvt_[`**`Q_ENTRY_SIG`**`])`

3.6.2 Application-Specific Code

Page 140, Listing 3.8 explanation section (1)	`"qp_port.h"`	`"qep_port.h"`
Page 141, Listing 3.8 explanation section (14)	Figure 3.1(1)	Figure 3.2(1)

CHAPTER 4 Hierarchical Event Processor Implementation

4.2 QEP Code Structure

4.2.1 QEP Source Code Organization

Location	Is	Should be
Page 153, Listing 4.1	80x88	80x86
Page 153, Listing 4.1	`qep_pkg.h` - internal, packet-scope interface	`qep_pkg.h` - internal, package-scope interface

4.3 Events

4.3.1 Event Signal (QSignal)

Page 155, In code snippet	`QEP_SIGNAL_SIZE`	`Q_SIGNAL_SIZE`

4.3.2 QEvent Structure in C

Page 156, footnote no. 4	Casting from **subclass to superclass** is called in OOP downcasting…	Casting from **superclass to subclass** is called in OOP downcasting…
Page 156, last paragraph	became	become

4.4 Hierarchical State-Handler Functions

4.4.2 Hierarchical State-Handler Function Example in C

Page 160, Listing 4.4 Explanation Section (6)	`Q_HANLDED()`	`Q_HANDLED()`

4.4.3 Hierarchical State-Handler Function Example in C++

Page 160, 5th paragraph	state "operand1"	state "int1"

4.5 Hierarchical State Machine Class

4.5.1 Hierarchical State Machine in C (Structure QHsm)

Location	Is	Should be
Page 162, Listing 4.6 Label (5)	`uint8_t QHsm_isIn(QHsm *me, QHsmState state);`	`uint8_t QHsm_isIn(QHsm *me, QStateHandler state);`

4.5.2 Hierarchical State Machine in C++ (Structure QHsm)

Page 163, Listing 4.7 Label (4)	`uint8_t isIn( QHsmState state);`	`uint8_t isIn( QStateHandler state);`

4.5.4 Entry/Exit Actions and Nested Initial Transitions

Page 168, first NOTE box, third line	`QEQ_EMPTY_SIG`	`QEP_EMPTY_SIG`

4.5.5 Reserved Events and Helper Macros in QEP

Page 168, code snippet (update for QP 4.3.00 and later)

```
QEvent const QEP_reservedEvt_[] {
    { (QSignal)QEP_EMPTY_SIG_, (uint8_t)0, (uint8_t)0 },
    { (QSignal)Q_ENTRY_SIG_,   (uint8_t)0, (uint8_t)0 },
    { (QSignal)Q_EXIT_SIG_,    (uint8_t)0, (uint8_t)0 },
    { (QSignal)Q_INIT_SIG_,    (uint8_t)0, (uint8_t)0 }
}
```

Page 169, in code snippet, 2nd macro	... /* QS software tracing instrumentation for state **entry** */	... /* QS software tracing instrumentation for state **exit** */
Page 169, in code snippet 3rd macro	... /* QS software tracing instrumentation for state **exit** */	... /* QS software tracing instrumentation for state **entry** */
Page 170, first paragraph	Myrphy	Murphy

4.5.6 Topmost Initial Transition (QHsm_init())

Page 170, Section 4.5.6 Bullet 4.	Execution of the entry actions to the “result” state	Execution of the entry actions to the “ready” state
Page 170, Section 4.5.6 Bullet 5.	Execution of the actions associated with the initial transition defined in the “result” state	Execution of the actions associated with the initial transition defined in the “ready” state

4.5.7 Dispatching Events (QHsm_dispatch(), General Structure)

Location	Is	Should be
Page 175, Listing 4.11 explanation section (2)	initial transition	transition

4.5.8 Executing a Transition in the State Machine (QHsm_dispatch(), Transitoin)

Page 182, Listing 4.12 explanation section (5)	… and involves only **entry** to the target but no exit from the source.	… and involves only **exit** from the source but no entry to the target.
Page 183, Listing 4.12 explanation section (10)	(10) The topologies shown in 4.6(G) and (H) require traversal of the target state hierarchy stored in the array path[] to find the match with any of the superstates of the source.	(10) Because every scan for a match with a given superstate of the source exhausts all possible matches for the LCA, the source's superstate can be safely exited.
Page 183, Listing 4.11 explanation section (11)	(11) Because every scan for a match with a given superstate of the source exhausts all possible matches for the LCA, the source's superstate can be safely exited.	(11) The topologies shown in 4.6(G) and (H) require traversal of the target state hierarchy stored in the array path[] to find the match with any of the superstates of the source.

4.6 Summary of Steps for Implementing HSMs with QEP

4.6.2 Step 2: Defining Events

Page 185, last line	<qp>\qpc\include\qevent.h	<qp>\qpcpp\include\qevent.h

4.6.5 Step 5: Defining the State-Handler Functions

Page 188, last paragraph before the NOTE	...Such state **disignate** `&QHsm::top` as the argument to the `Q_SUPER()` macro.	...Such state **designate** `&QHsm::top` as the argument to the `Q_SUPER()` macro.

4.6.9 Coding Regular Transitions

Page 190, 4th paragraph	... Listing 4.5 provides two examples of regular state transitions.	... Listing 4.5 provides an example of a regular state transition.

4.6.10 Coding Guard Conditions

Page 191, 1st paragraph	`Cac1::begin()`	`Calc1::begin()`

4.7 Pitfalls to Avoid While Coding State Machines with QEP

4.7.7 Code Outside the switch Statement

Location	Is	Should be
Page 196, 2nd paragraph	...event that dispatched the state machine...	...event that is dispatched to the state machine...

4.7.8 Suboptimal Signal Granularity

Page 197, end of 3rd paragraph	IDC_1_9_SIG	DIGIT_1_9_SIG

CHAPTER 5 State Patterns

5.1 Ultimate Hook

5.1.4 Sample Code

Location	Is	Should be
Page 210, Listing 5.1 explanation section (1)	Every QEP application needs to include qep_port**h**.h	Every QEP application needs to include qep_port.h

5.2 Reminder

5.2.4 Sample Code

Page 213, 2nd paragraph	**file** REMINDER.EXE file	REMINDER.EXE file
Page 217, NOTE	Windows GUI applications can call the PostMessage() Win32 API to queue messages and the WM_TIMER message to receive timer updates.	Windows GUI applications can call the PostMessage() Win32 API to queue messages and provide a WM_TIMER case in the window procedure to receive timer updates.

5.3 Deferred Event

5.3.4 Sample Code

Page 222, 1st paragraph	file DEFER.EXE file	DEFER.EXE file
Page 229, Listing 5.4, explanation section (4)	Listing 4.1	Listing 4.2

5.4 Orthogonal Component

5.4.4 Sample Code

Page 234, 2nd paragraph	**file** COMP.exe file	COMP.exe file
Page 234, Figure 5.11 2nd note from bottom	... from the `Alarm` component	... from the `AlarmClock` component
Page 236, Listing 5.6	`/* the HSM version of the Alarm component */`	`/* the FSM version of the Alarm component */`

Page 237, listing 5.7	`/* HSM definition...`	`/* FSM definition...`
P. 239, Listing 5.8 caption	(file clock.c)	(file comp.c)

5.4.5 Consequences

Location	Is	Should be
Page 244, footnote 8	must explicitly instantiate all components **explicitly**	must explicitly instantiate all components

5.5 Transition to History

5.5.3 Solution

Page 245, last paragraph	doorClosed_history (abbreviated to history in Figure 5.12).	doorClosed_history.

5.5.4 Sample Code

Page 246	**file** HISTORY.exe file	HISTORY.exe file
Page 247, Listing 5.9 Comment line	HSM definitio----…	HSM definition----…
Page 250, last paragraph	requires setting doorClosed_history to &ToasterOven_toasting in the exit action from "toasting" to &ToasterOven_baking in the exit action from "baking," and so on	requires setting doorClosed_history to &ToasterOven_toasting in the exit action from "toasting," and likewise to &ToasterOven_baking in the exit action from "baking," and so on

CHAPTER 6 Real-Time Framework Concepts

6.1 Inversion of Control

Location	Is	Should be
Page 256, last paragraph	control is key part	control is **the** key part

6.2 CPU Management

6.2.2 Traditional Multitasking Systems

Page 261, 3rd paragraph	easer	easier

6.5 Event Memory Management

6.5.6 Event Ownership

Page 288, Figure 6.14	retrun-from-dispatch(e)	return-from-dispatch(e)

6.7 Error and Exception Handling

6.7.1 Design by Contract

Page 295, 2nd paragraph from the bottom	… code to "**wonder** around," silently taking care of …	… code to "**wander** around," silently taking care of …

6.7.2 Errors versus Exceptional Conditions

Page 297, 1st paragraph	(see Section 6.1.6)	(see Section 6.7.6)

6.7.3 Customizable Assertions in C and C++

Page 298, Listing 6.1 explanation section (1)	When disabled, all assertion macros expand to empty statements that don't generate any code.	When disabled, all assertion macros, except `Q_ALLEGE()`, expand to empty statements that don't generate any code.
Page 300, Listing 6.1 explanation section (11)	(see [Murphy 01])	(see [Murphy 01a])

CHAPTER 7 Real-Time Framework Implementation

7.1 Key Features of the QF Real-Time Framework

7.1.1 Source Code

Location	Is	Should be
Page 309, 1st paragraph	www.quantum-leaps.com/doc/AN_QL_Coding_Standard.pdf	www.quantum-leaps.com/resources/AN_QL_Coding_Standard.pdf

7.1.2 Portability

Page 310, 1st paragraph	QP-nano in Chapter 11	QP-nano in Chapter 12

7.1.6 Zero-Copy Event Memory Management

Page 312, Section 7.1.6	Perhaps that most ...	Perhaps the most ...

7.1.12 Low-Power Architecture

Page 314, 1st paragraph	power-savings features	power-saving features

7.2 QF Structure

Page 315, 1st paragraph	As all real-time frameworks, QF provides the central base class QActive...	QF provides the central base class QActive...
Page 315, left side, near bottom	Star Wars application	Fly 'n' Shoot application

7.2.1 QF Source Organization

Page 317, Listing 7.1, in description of "qte_darm.c"	`QTimeEvt_darm()`	`QTimeEvt_disarm()`

!7.3 Critical Sections in QF

NOTE: This section has been updated for QP **4.3.00** (01-Nov-11), which changed the names of critical section macros and introduces macros for unconditional interrupt disabling/enabling. This was done to simplify and speed up the built-in Vanilla and QK kernels, which no longer are dependent on the interrupt disabling policy.

QF, just like any other system-level software, must protect certain sequences of instructions against preemptions to guarantee thread-safe operation. The sections of code that must be executed indivisibly are called critical sections.

In an embedded system environment, QF uses the simplest and most efficient way to protect a section of code from disruptions, which is to disable interrupts on entry to the critical section and re-enable interrupts at the exit from the critical section. In systems where locking interrupts is not allowed, QF can employ other mechanisms supported by the underlying operating system, such as a mutex.

NOTE
The maximum time spent in a critical section directly affects the system's responsiveness to external events (interrupt latency). All QF critical sections are carefully designed to be as short as possible and are of the same order as critical sections in any commercial RTOS. Of course, the length of critical sections depends on the processor architecture and the quality of the code generated by the compiler.

To hide the actual critical section implementation method available for a particular processor, compiler, and operating system, the QF platform abstraction layer includes two macros, `QF_INT_DISABLE()` and `QF_INT_ENABLE()`, to disable and enable interrupts, respectively.

!7.3.1 Saving and Restoring Critical Section Status

The most general critical section implementation involves saving the critical section status before entering the critical section and restoring the status upon exit from the critical section. Listing 7.2 illustrates the use of this critical section type.

Listing 7.2 Example of the "saving and restoring critical section status" policy

```
     {
(1)      unsigned int crit_stat;
         . . .
(2)      crit_stat = get_int_status();
(3)      disable_interrupts();
         . . .
(4)      /* critical section of code */
         . . .
(5)      set_int_status(crit_stat);
         . . .
     }
```

(1) The temporary variable `crit_stat` holds the interrupt status across the critical section.

(2) Right before entering the critical section, the current interrupt status is obtained from the CPU and saved in the `crit_stat` variable. Of course, the name of the actual function to obtain the interrupt status can be different in your system. This function could actually be a macro or inline assembly statement.

(3) Interrupts are disabled using the mechanism provided by the compiler.

(4) This section of code executes indivisibly because it cannot be interrupted..

(5) The original interrupt status is restored from the `crit_stat` variable. This step re-enables interrupts only if they were enabled at step 2. Otherwise, interrupts remain disabled.

Listing 7.3 shows an example of the "saving and restoring critical section status" policy.

Listing 7.3 QF macro definitions for the "saving and restoring critical section status" policy

```
(1) #define QF_CRIT_STAT_TYPE unsigned int
(2) #define QF_CRIT_ENTRY(stat_) do { \
        (stat_) = get_int_status();          \
        disable_interrupts(); \
    } while (0)
(3) #define QF_CRIT_EXIT(stat_) set_int_status(stat_)
```

(1) The macro `QF_CRIT_STAT_TYPE` denotes a data type of the "criticasl section status" variable, which holds the critical section status. Defining this macro in the `qf_port.h` header file indicates to the QF framework that the policy of "saving and restoring critical section status" is used, as opposed to the policy of "unconditional disabling and enabling interrupts" described in the next section.

(2) The macro `QF_CRIT_ENTRY()` encapsulates the mechanism of entering the critical section. The macro takes the parameter `stat_`, into which it saves the critical section status.

NOTE

The `do {. . .} while (0)` loop around the `QF_CRIT_ENTRY()` macro is the standard practice for syntactically correct grouping of instructions. You should convince yourself that the macro can be used safely inside the `if-else` statement (with the semicolon after the macro) without causing the "dangling-else" problem. I use this technique extensively in many QF macros.

(3) The macro `QF_CRIT_EXIT()` encapsulates the mechanism of restoring the interrupt status. The macro restores the critical section status from the argument `stat_`.

The main advantage of the "saving and restoring critical section status" policy is the ability to *nest critical sections*. The QF real-time framework is carefully designed to never nest critical sections internally. However, nesting of critical sections can easily occur when QF functions are invoked from within an already established critical section, such as an interrupt service routine (ISR). Most processors disable interrupts in hardware upon the interrupt entry and enable interrupts upon the interrupt exit, so the whole ISR is a critical section. Sometimes you can re-enable interrupts inside ISRs, but often you cannot. In the latter case, you have no choice but to invoke QF services, such as event posting or publishing, with interrupts disabled. This is exactly when you must use this type of critical section.

!7.3.2 Unconditional Disabling and Enabling Interrupts

The simpler and faster critical section policy is to always unconditionally enable interrupts in `QF_CRIT_EXIT()`. Listing 7.4 provides an example of the QF macro definitions to specify this type of critical section.

Listing 7.4 QF macro definitions for the "unconditional interrupt disabling and enabling" policy

```
(1) /* QF_CRIT_STAT_KEY not defined */
(2) #define QF_CRIT_ENTRY(dummy_)  disable_interrupts()
(3) #define QF_CRIT_EXIT(dummy_)   enable_interrupts()
```

(1) The macro `QF_CRIT_STAT_KEY` is *not* defined in this case. The absence of the `QF_CRIT_STAT_KEY` macro indicates to the QF framework that the critical section status is not saved across the critical section.

(2) The macro `QF_CRIT_ENTRY()` encapsulates the mechanism of entering critical section. For consistency, the macro must take a parameter, but the parameter is not used in this case and so it is named `dummy_`.

(3) The macro `QF_CRIT_EXIT()` encapsulates the mechanism of exiting critical section. For consistency, the macro must take a parameter, but the parameter is not used in this case and so it is named `dummy_`.

The inability to nest critical sections does not necessarily mean that you cannot nest interrupts. Many processors are equipped with a prioritized interrupt controller, such as the Intel 8259A Programmable Interrupt Controller (PIC) in the 80x86-based PC or the Nested Vectored Interrupt Controller (NVIC) integrated inside the ARM Cortex-M3. Such interrupt controllers handle interrupt prioritization and nesting before the interrupts reach the processor core. Therefore, you can safely enable interrupts at the processor level, thus avoiding nesting of critical sections inside ISRs. Listing 7.5 shows the general structure of an ISR in the presence of an interrupt controller.

Listing 7.5 General structure of an ISR in the presence of a prioritized interrupt controller

```
(1) void interrupt ISR(void) {  /* entered with interrupts locked in hardware */
(2)     Acknowledge the interrupt to the interrupt controller (optional)
(3)     Clear the interrupt source, if level triggered
(4)     QF_INT_ENABLE();       /* enable the interrupts at the processor level */
(5)     body of the ISR, use QF calls, e.g., QF_tick(), Q_NEW or QF_publish()
(6)     QF_INT_DISABLE();        /* lock the interrupts at the processor level */
(7)     Write End-Of-Interrupt (EOI) instruction to the Interrupt Controller
(8) }
```

(1) Most processors enter the ISR with interrupts disabled in hardware.

(2) The interrupt controller must be notified about entering the interrupt. Often this notification happens automatically in hardware before vectoring (jumping) to the ISR. However, sometimes the interrupt controller requires a specific notification from the software. Check your processor's datasheet.

(3) You need to explicitly clear the interrupt source, if it is level triggered. Typically you do it before re-enabling interrupts at the CPU level, but a prioritized interrupt controller will prevent the same interrupt from preempting itself, so it really does not matter if you clear the source before or after enabling interrupts

(4) Interrupts are explicitly enabled at the CPU level, which is the key step of this ISR. Enabling interrupts allows the interrupt controller to do its job, that is, to prioritize interrupts. At the same time, enabling interrupts terminates the critical section established upon the interrupt entry. Note that this step is only necessary when the hardware actually disables interrupts upon the interrupt entry (e.g., the ARM Cortex-M3 leaves interrupts enabled).

(5) The main ISR body executes outside the critical section, so QF services can be safely invoked without nesting critical sections.

NOTE

The prioritized interrupt controller remembers the priority of the currently serviced interrupt and allows only interrupts of higher priority than the current priority to preempt the ISR. Lower- and same-priority interrupts are stopped at the interrupt controller level, even though the interrupts are enabled at the CPU level. The interrupt prioritization happens in the interrupt controller hardware until the interrupt controller receives the end-of-interrupt (EOI) instruction.

(6) Interrupts are locked to establish critical sections for the interrupt exit.

(7) The end-of-interrupt (EOI) instruction is sent to the interrupt controller to stop prioritizing this interrupt level.

(8) The interrupt exit synthesized by the compiler restores the CPU registers from the stack, which includes restoring the CPU status register. This step typically unlocks interrupts.

!7.3.3 Internal QF Macros for Critical Section Entry/Exit

The QF platform abstraction layer (PAL) uses the critical section entry/exit macros `QF_CRIT_ENTRY()`, `QF_CRIT_EXIT()`, and `QF_CRIT_STAT_TYPE` in a slightly modified form. The PAL defines internally the parameterless macros, shown in Listing 7.6. Please note the trailing underscores in the internal macros' names.

Listing 7.5 Internal macros for critical section entry/exit (file <qp>\qpc\qf\source\qf_pkg.h)

```
#ifndef QF_CRIT_STAT_TYPE /* simple unconditional critical section entry/exit */
    #define QF_CRIT_STAT_
    #define QF_CRIT_ENTRY_()     QF_CRIT_ENTRY(dummy)
    #define QF_CRIT_EXIT_()      QF_CIR_EXIT(dummy)
#else                      /* policy of saving and restoring interrupt status */
    #define QF_CRIT_STAT_        QF_CRIT_STAT_TYPE critStat_;
    #define QF_CRIT_ENTRY_()     QF_CRIT_ENTRY(critStat_)
    #define QF_CRIT_EXIT_()      QF_CRIT_EXIT(critStat_)
#endif
```

The internal macros `QF_CRIT_STAT_`, `QF_CRIT_ENTRY_()`, and `QF_CRIT_EXIT_()` enable me writing the same code for the case when the interrupt key is defined and when it is not. The following code snippet shows the usage of the internal QF macros. Convince yourself that this code works correctly for both critical section policies.

```
void QF_service_xyz(arguments) {
    QF_CRIT_STAT_
    . . .
    QF_CRIT_ENTRY_();
    . . .
    /* critical section of code */
    . . .
    QF_CRIT_EXIT_();
}
```

7.4 Active Objects

Location	Is	Should be
Page 326, Listing 7.7, explanation section (8)	`QEqueue`	`QEQueue`
Page 326, last paragraph	See Chapter 8, "POSIX QF Port," ...	See Section 8.4, "QF Port to Linux (Conventional POSIX-Compliant OS)," ...

7.4.3 Thread of Execution and Active Object Priority

Page 332, Listing 7.9 explanation section (4)	The argument 'stkSto' is a pointer to the storage for the private stack, and the argument 'stkSize' is the size of that stack (in bytes), respectively.	The argument 'stkSto' is a pointer to the storage for the private stack, and the argument 'stkSize' is the size of that stack (in bytes).

7.5 Event Management in QF

!7.5.1 Event Structure

NOTE: This section has been updated for QP **4.2.00** (14-Jul-11), which changed the `QEvent` structure and extened the number of event pools beyond the limit of 3.

QF uses the same event representation as the QEP event processor described in Part I. Events in QF are represented as instances of the `QEvent` structure (shown in Listing 7.10), which contains the event signal `sig` and two additional bytes `poolId_` and `refCtr_` to represent the internal "bookkeeping" information about the event.

Listing 7.10 QEvent structure defined in <qp>\qpc\include\qevent.h

```
typedef struct QEventTag {
    QSignal sig;                            /* signal of the event instance */
    uint8_t poolId_;                        /* pool ID (0 for static event) */
    uint8_t refCtr_;                                   /* reference counter */
} QEvent;
```

The QF framework uses the `QEvent.poolId_` data byte to store the event pool ID of the event The pool ID of zero is reserved for static events, that is, events that do not come from any event pool. With this representation, a static event has a unique, easy-to-check signature (`QEvent.poolId_ == 0`). Conversely, the signature (`QEvent.poolId_ != 0`) unambiguously identifies a dynamic event.

NOTE: The **Figure 7.4** on page 334 is now obsolete.

NOTE

The data members `QEvent.poolId_` and `QEvent.refCtr_` are used only by the QF framework for managing dynamic events (see the following section). For every static event, you must initialize the poolId_ member to zero. Otherwise, the `QEvent.poolId_` or `QEvent.refCtr_` data members should never be of interest to the application code.

To encapsulate the access to the "private" `poolId_` and `refCtr_` members, the QF framework defines a set of internal macros shown below (file `<qp>\qpc\qf\source\qf_pkg.h`).

```
/* access to the poolId of an event */
#define EVT_POOL_ID(e_)      ((e_)->poolId_)

/* access to the refCtr of an event */
#define EVT_REF_CTR(e_)      ((e_)->refCtr_)

/* increment the refCtr of an event */
#define EVT_INC_REF_CTR(e_)  (++((QEvent *)(e_))->refCtr_)

/* decrement the refCtr of an event */
#define EVT_DEC_REF_CTR(e_)  (--((QEvent *)(e_))->refCtr_)
```

7.5.2 Dynamic Event Allocation

Location	Is	Should be
Page 335, Listing 7.11(1)	`QF_POOL_TYPE_ QF_pool_[3];`	`QF_POOL_TYPE_ QF_pool_[QF_MAX_EPOOL];`
Page 335, Listing 7.11 line before (6)	perfom	perform
Page 336, Listing 7.11 explanation section (4)	see Chapter 6, "Customized Assertions in C and C++"	see Section 6.7.3, "Customizable Assertions in C and C++"
Page 338, last paragraph	`evT_`	`ev`**`t`**`T_`

7.5.3 Automatic Garbage Collection

Page 341, explanation section (12)	`QF_EPOOL_PU_()`	`QF_EPOOL_PUT_()`

7.5.4 Deferring and Recalling Events

Page 342, Listing 7.13 explanation section (1)	`QActive_defer()` takes posts the	`QActive_defer()` posts the

7.6.1 Dirtect Event Posting

Page 344, 2nd paragraph code snippet	`AO_ship`	`AO_Ship`
Page 344, 3rd paragraph	`AO_ship`	`AO_Ship`

7.6.2 Publish-Subscribe Event Delivery

Page 345, Listing 7.14 caption	**`QF_psInit()`** (file `\qpc\`**`init`**`\qf.h`)	**`QSubscrList`** (file `\qpc\`**`include`**`\qf.h`)
Page 346, 1st paragraph	`QF_subsrcrList_`	`QF_subscrList_`
Page 346, 1st paragraph	`QF_maxSignal`	`QF_maxSignal_`
Page 348, Listing 7.17 caption	`qa_pspub.c`	`qf_pspub.c`

7.7.2 The System Clock Tick and the QF_tick() Function

Location	Is	Should be
Page 355, Listing 7.19 explanation section (8-13)	removing a link	removing a node
Page 356, Listing 7.19 explanation section (21)	The link is advanced	The time event node pointer is advanced

7.7.3 Arming and Disarming a Time Event

Page 357, Listing 7.20 explanation (3-6)	inserting a **link**	inserting a **node**
Page 358, Listing 7.21 explanation section (3-8)	removing a **link** from	removing a **node** from

7.8.1 The EQueue Structure

Page 360, 3rd paragraph	Figure 7.**8**	Figure 7.**9**
Page 360, Figure 7.9	Counterclockwise movement...	Reverse the direction of the arrow and the text in note to "Clockwise movement..."
Page 360, Last paragraph	frequently bypass**,** the buffering	frequently bypass the buffering
Page 361, 1st paragraph	**counter**clockwise	clockwise

7.8.3 The Native QF Active Object Queue

Page 363, 2nd paragraph	included directly in the level `QActive` structure	included directly in the higher-level `QActive` structure
Page 364, Listing 7.24 explanation section (1)	The function QActive_get_() returns a read-only (const) pointer to an event	The function `QActive_get_()` returns a pointer to a read-only (const) event
Page 365, Listing 7.24 explanation section (12,13)	(12,13) Additionally, a platform-specific macro...	(12) Additionally, a platform-specific macro... (13) The event pointer is returned to the caller. This pointer can never be NULL.

7.8.4 The "Raw" Thread-Safe Queue

Location	Is	Should be
Page 368, listing 7.26	`return (QState)0;`	`return Q_HANDLED();`
Page 368, listing 7.26	`return (QState)&MyAO_stateA;`	`return Q_SUPER(&MyAO_stateA);`
Page 369, Listing 7.26 explanation section (8)	you call `QEQueue_get()` to post an event	you call `QEQueue_postFIFO()` or `QEQueue_postLIFO()` to post an event

7.9.1 Obtaining a Memory Block from the Pool

Page 375, last paragraph	see Listing 7.12(3)	see Listing 7.12(5)

7.10 Native QF Priority Set

Page 377, 1st paragraph	`QPset64`	`QPSet64`

7.11 Native Cooperative "Vanilla" Kernel

Page 380, Figure 7.12, three occurrences	`prio__`	`prio`

7.11.1 The qvanilla.c Source Code

Page 383, last paragraph	Listing 7.32(20)	Listing 7.32(8)
Page 384, 3rd paragraph	Yet other class of MCUs...	Yet another class of MCUs...

7.11.2 QP Reference Manual

Page 386, 6th paragraph	www.quantumleaps.com	www.quantum-leaps.com

CHAPTER 8 Porting and Configuring QF

8.1 The QP Platform Abstractin Layer (PAL)

8.1.4 The qep_port.h Header File

Location	Is	Should be
Page 400, Listing 8.2, Explanation Section (4)	The default for `Q_SIGNAL_SIZE` is **1** (256 signals).	The default for `Q_SIGNAL_SIZE` is **2** (64K signals). NOTE: Changed in QP 4.2.00.

!8.1.5 The qf_port.h Header File

Page 401, Listing 8.3 before label (10)		`QF_MAX_EPOOL 3` NOTE: Added in QP 4.2.00. This macro determines the maximum number of event pools in the QF with the range of 1..255.
Page 401, Listing 8.3 label (15)	`QF_INT_KEY_TYPE`	`QF_CRIT_STAT_TYPE` NOTE: Changed in QP 4.3.00.
Page 401, Listing 8.3 label (16)	`QF_INT_LOCK(key_)`	`QF_CRIT_ENTRY(stat_)` NOTE: Changed in QP 4.3.00.
Page 401, Listing 8.3 label (17)	`QF_INT_UNLOCK(key_)`	`QF_CRIT_EXIT(stat_)` NOTE: Changed in QP 4.3.00.
Page 402, Listing 8.3 Explanation section (2)	...Section 8.4, "Conventional POSIX-Compliant OS (Linux)"	...Section 8.4, "QF Port to Linux (Conventional POSIX-Compliant OS)"

NOTE: QP 4.3.00 introduced support for building sequence diagrams from QS software traces. To support this feature, the `qf_port.h` header file adds the sender parameter in event-producing functions `QF_tick()`, `QF_publish()`, and `QActive_postFIFO()`, and defines new macros `QF_TICK()`, `QF_PUBLISH()`, and `QACTIVE_POST()`.

```
#ifndef Q_SPY                                 /* QS software tracing disabled? */
    void QF_tick(void);
    void QF_publish(QEvent const *e);
    void QActive_postFIFO(QActive *me, QEvent const *e);
#else                                           /* QS software tracing enabled */
```

```
    void QF_tick(void const *sender);
    void QF_publish(QEvent const *e, void const *sender);
    void QActive_postFIFO(QActive *me, QEvent const *e,
                          void const *sender);
#endif

#ifndef Q_SPY                                /* QS software tracing disabled? */
    #define QF_TICK(dummy)                QF_tick()
    #define QF_PUBLISH(e_, dummy)         QF_publish(e_)
    #define QACTIVE_POST(me_, e_, dummy)  QActive_postFIFO((me_), (e_))
#else                                          /* QS software tracing enabled */
    #define QF_TICK(sender_)               QF_tick(sender_)
    #define QF_PUBLISH(e_, sender_)        QF_publish((e_), (sender_))
    #define QACTIVE_POST(me_, e_, sender_) \
         QActive_postFIFO((me_), (e_), (sender_))
#endif
```

By using the macros `QF_TICK()`, `QF_PUBLISH()`, and `QACTIVE_POST()`, the source code you write can always be the same, (e.g. `QF_PUBLISH(&foo_evt, me)`). However, the compiler can determine which version gets called. When QS tracing is enabled, the macro will become the call to `QF_publish(&foo_evt, me)` with the 'me' sender parameter, and when not tracing, the compiler will call `QF_publish(&foo_evt)` without the sender parameter.

!QF Critical Section Mechanism

> **NOTE:** This section has been updated for QP **4.3.00** (01-Nov-11), which changed the names of critical section macros and introduces macros for unconditional interrupt disabling/enabling. This was done to simplify and speed up the built-in Vanilla and QK kernels, which no longer are dependent on the interrupt disabling policy.

This section defines the critical section mechanism used within the QF framework, which you always need to provide. Refer to Section 7.3, "Critical Sections in QF," in Chapter 7 for the detailed discussion of critical sections in QF.

(1) The macro `QF_CRIT_STAT_TYPE` defines the data type of the critical section status variable. When you define this macro, you indicate to the QF framework that the policy of "saving and restoring critical section status" is used. Conversely, when you don't define the macro, the QF framework assumes the policy of "unconditional exiting from the critical section."

(2) The macro `QF_CRIT_ENTRY()` encapsulates the mechanism of entering a critical section. The macro takes a parameter into which it saves the critical section status. The parameter is not used if you use the simple policy of "unconditional exiting from the critical section."

(3) The macro `QF_CRIT_EXIT()` encapsulates the mechanism of exiting a critical section. The macro takes a parameter from which it restores the critical section status. The parameter is not used if you use the simple policy of "unconditional exiting from the critical section."

(4)

Active Object Event Queue Operations

Location	Is	Should be
Page 406, explanation section (24)	Section 8.3	Section 8.4

8.1.6 The qf_port.c Source File

Page 410, Listing 8.4 explanation section (8)	the size of that stack (in bytes), respectively	the size of that stack (in bytes)
Page 410, last line	**Q**S-specific	**O**S-specific

8.1.6 System Clock Tick (Calling QF_tick())

Page 413 1st paragraph in Section 8.1.9	As you design **you** port, you must decide…	As you design **your** port, you must decide…

8.2 Porting the Cooperative "Vanilla" Kernel

Page 414, 1st paragraph in Section 8.2	`qep_porth.h`	`qep_port.h`

8.2.1 The qep_port.h Header File

Page 415, in paragraph before Listing 8.7	exact-with	exact-width

8.2.2 The qf_port.h Header File

Page 416, 2nd paragraph	Section 8.2	Section 8.3

8.3 QF Port to µC/OS-II (Conventional RTOS)

Page 421, 2nd paragraph	RTOS that it is superbly documented	RTOS that is superbly documented

8.3.2 The qf_port.h Header File

Page 425, 5th paragraph (after (9))	`QF_EPPOL_TYPE`	`QF_EPOOL_TYPE`

Page 425, section (11)	`QF_EPPOL_TYPE`	`QF_EPOOL_TYPE`

8.3.2 The qf_port.c Source File

Location	Is	Should be
Page 430, explanation section (30)	whereas timeout	where a timeout

8.4 QF Port to Linux (Conventional POSIX-Compliant OS)

Page 431, 3rd paragraph	build you own	build your own

8.4.2 The qf_port.h Header File

Page 436, Listing 8.19 for lines below (6)	due to insufficient privieges	due to insufficient privileges
Page 437, Listing 8.19(17)	stopps	stops
Page 438, explanation section (2)	lock in physical memory of all the pages mapped	lock in physical memory all of the pages mapped
Page 439, top of page	(6) The "ticker" thread runs... (7) The "ticker" thread calls...	(7) The "ticker" thread runs... (8) The "ticker" thread calls...
Page 439, explanation section (14)	described in triggered	described is triggered
Page 440, paragraph following (29-33)	and the rest highest priorities	and the rest of the highest priorities

CHAPTER 9 Developing QP Applications

9.2 Dinigng Philosophers Problem

Location	Is	Should be
Page 446, Section 9.2 title	Philosopher	Philosophers

9.2.1 Step1: Requirements

Page 447, 1st paragraph	your always need	you always need
Page 447, Figure 9.1 caption	Philosopher	Philosophers

9.2.2 Step 2: Sequence Diagrams

NOTE: This section is missing in some printings of the book (missing page 448). Therefore, this section is copied here verbatim.

A good starting point in designing any event-driven system is to draw sequence diagrams for the main scenarios (main-use cases) identified from the problem specification. To draw such diagrams, you need to break up your problem into active objects with the main goal of minimizing the coupling among active objects. You seek a partitioning of the problem that avoids resource sharing and requires minimal communication in terms of number and size of exchanged events.

DPP has been specifically conceived to make the philosophers contend for the forks, which are the shared resources in this case. In active object systems, the generic design strategy for handling such shared resources is to encapsulate them inside a dedicated active object and to let that object manage the shared resources for the rest of the system (i.e., instead of directly sharing the resources, the rest of the application shares the dedicated active object). When you apply this strategy to DPP,

you will naturally arrive at a dedicated active object to manage the forks. I named this active object Table. The sequence diagram in Figure 9.2 shows the most representative event exchanges among any two adjacent Philosophers and the Table active objects.

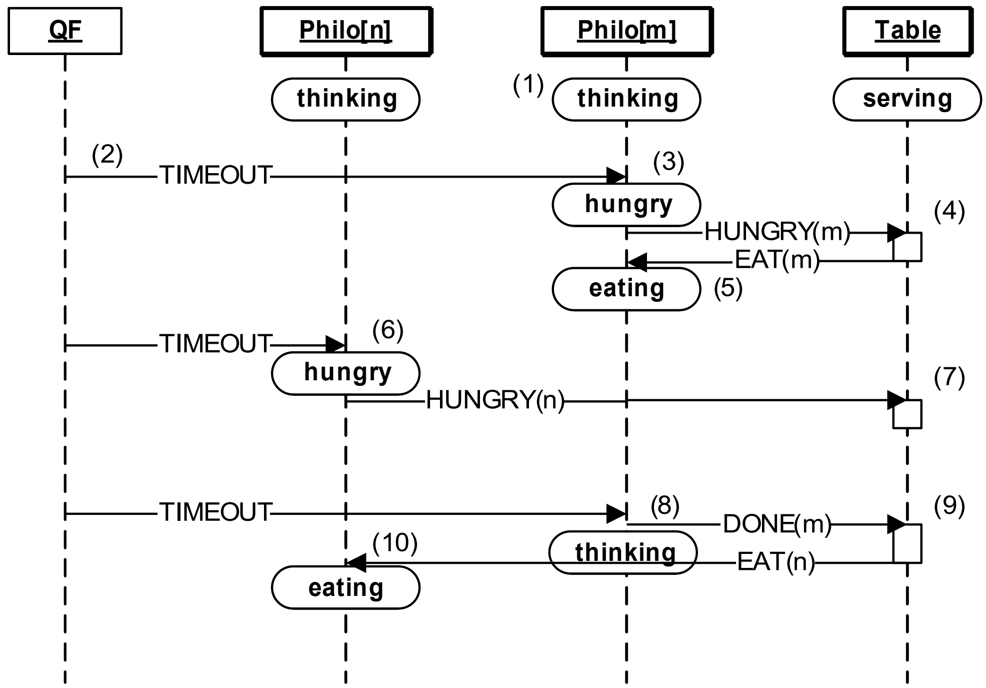


Figure 9.2: The sequence diagram of the DPP application.

(1) Each Philosopher active object starts in the "thinking" state. Upon the entry to this state, the Philosopher arms a one-shot time event to terminate the thinking.

(2) The QF framework posts the time event (timer) to Philosopher[m].

(3) Upon receiving the TIMEOUT event, Philosopher[m] transitions to "hungry" state and posts the HUNGRY(m) event to the Table active object. The parameter of the event tells the Table which Philosopher is getting hungry.

(4) The Table active object finds out that the forks for Philosopher[m] are available and grants it permission to eat by publishing the EAT(m) event.

(5) The permission to eat triggers the transition to "eating" in Philosopher[m]. Also, upon the entry to "eating," the Philosopher arms its one-shot time event to terminate the eating.

(6) The Philosopher[n] receives the TIMEOUT event and behaves exactly as Philosopher[m], that is, transitions to "hungry" and posts HUNGRY(n) event to the Table active object.

(7) This time the Table active object finds out that the forks for Philosopher[n] are not available, and so it does not grant the permission to eat. Philosopher[n] remains in the "hungry" state.

(8) The QF framework delivers the timeout for terminating the eating to Philosopher[m]. Upon the exit from "eating," Philosopher[m] publishes event DONE(m) to inform the application that it is no longer eating.

(9) The Table active object accounts for free forks and checks whether any direct neighbors of Philosopher[m] are hungry. Table posts event EAT(n) to Philosopher[n].

(10) The permission to eat triggers the transition to "eating" in Philosopher[n].

9.3 Running DPP on Various Platforms

9.3.3 µC/OS-II

Location	Is	Should be
Page 470, explanation section (1-3)	oversized all stacks of to 256 of 16-bit stack entries	oversized all stacks to have 256 16-bit stack entries

9.3.4 Linux

Page 474, 3rd line from the top	`l_delay = atol(argv[1]);`	`l_delay = atol(argv[1]);`
Page 474, listing 9.8, after (9)	`QS_EXIT();`	(remove line)
Page 475, explanation section (5)	`t_sav`	`l_tsav`

9.4.1 Sizing Event Queues

Page 477, Section 9.4.1 title	**In** Sizing Event Queues	Sizing Event Queues
Page 477, 4th paragraph	Listing 7.24(12-14)	Listing 7.25(12-14)

CHAPTER 10 Preemptive Run-to-Completion Kernel

10.1 Reasons for Choosing a Preemptive Kernel

Location	Is	Should be
Page 484, 1st paragraph	routed	rooted

10.2.3 Synchronous and Asynchronous Preemptions

Page 490, explanation section (10)	which as been	which has been

10.3.1 QK Source Code Organization

Page 498, Listing 10.1	`+-80x88\`	`+-80x86\`

10.3.2 The qk.h Header File

Page 498, Figure 10.5 three occurrences	`prio__`	`prio`
Page 502, explanation section (27)	at step 13	at step 14

10.3.4 The qk_sched.c Source File (QK Scheduler)

Page 508, listing 10.4 before (34)		`}` (missing brace)
Page 510, explanation section (25)	could have change	could have changed
Page 510, explanation section (29)	back to step (13)	back to step (15)

10.3.5 The qk.c Source File (QK Startup and Idle Loop)

Page 512, explanation section (1)	includes to the wider	includes the wider

10.4.3 Extended Context Switch (Coprocessor Support)

Location	Is	Should be
Page 520, last paragraph	`QK_scheduler_()`	`QK_schedule_()`
Page 521, last paragraph	`QK_scheduler_()`	`QK_schedule_()`
Page 523, Listing 10.9 explanation section (4)	does not **to** use	does not use

10.5 Porting QK

Page 524, 5th paragraph	`qep_porth.h`	`qep_port.h`

10.5.1 The qep_port.h Header File

Page 525, 3rd paragraph	exact-with	exact-width

10.5.1 The qf_port.h Header File

Page 525, last paragraph	`qk_porth.h`	`qk_port.h`

10.5.1 The qk_port.h Header File

Page 529, explanation section (1)	unconditional interrupt saving and restoring	unconditional interrupt locking and unlocking

10.6 Testing the QK Port

10.6.2 Priority-Ceiling Mutex

Page 535, Listing 10.13	`return (QState)0;`	`return Q_HANDLED();`

10.6.3 TLS Demonstration

Page 537, Listing 10.14 (two occurrences)	`return (QState)0;`	`return Q_HANDLED();`

10.6.4 Extended Context Switch Demonstration

Page 539, last paragraph	perform a lot	performs a lot

10.7 Summary

Page 540, 1st paragraph	embedded (RTE) stems	embedded (RTE) systems

CHAPTER 11 Software Tracing for Event-Driven Systems

11.2 Quantum Spy Software Tracing System

11.2.1 Example of a Software-Tracing Session

Location	Is	Should be
Page 545, 3rd paragraph	located in the directory	located at

11.2.2 The Human-Readable Trace Output

Page 549, last paragraph	first eight columns	first ten columns
Page 550, 3rd paragraph	`((0000135566 - 0000070346)/`**`7`**` = 65220 ~= 0x10000)`	`(0000135566 - 0000070346 = 65220 ~= 0x10000)`

11.3 QS Target Component

Page 551, 2nd paragraph	factor of two in data density	factor of two improvement in data density
Page 552, 3rd paragraph	many **the** elements of	many elements of
Page 552, 3rd paragraph	High Level Data Link Control	High-level Data Link Control
Page 552, 3rd paragraph	[HDLC]	[HDLC 07]

11.3.5 QS Filters

Page 562, 3rd paragraph	without entering the QS critical	without entering the QS critical section
Page 563, 4th paragraph	`(bimask & bit) != 0`	`(bitmask & bit) != 0`
Page 563, 4th paragraph	`(QS_glbFilter_[5] & 0x40) != ...)`	`((QS_glbFilter_[5] & 0x40) != ...)`
Page 563, last paragraph	records types	record types
Page 565, last paragraph	all local filters is set	all local filters are set
Page 566, 1st paragraph	`QS_BEGIN()`	`QS_BEGIN_NOCRIT()`
Page 566, 2nd paragraph (code snippet)	`#define QS_BEGIN(rec_, obj_) \`	`#define QS_BEGIN_NOCRIT(rec_,`

		`obj_)\`

11.3.6 QS Data Protocol

Page 567, paragraph 3, (item 2)	Following the Fame Sequence Number	Following the Frame Sequence Number
Page 567, paragraph 5, (item 4)	over the frame Sequence Number	over the Frame Sequence Number
Page 567, paragraph 6, (item 5)	HDLC flag	HDLC Flag
Page 568, 3rd paragraph	over the Fame Sequence Number	over the Frame Sequence Number

11.3.7 QS Trace Buffer

Page 569, Section 11.3.7, 2nd paragraph	You can employ just about any repetition physical data link available…	You can employ just about any physical data link available…
Page 569, 4th paragraph	Your can apply	You can apply
Page 570, Listing 11.7 caption	`QS_initBuf(`	`QS_initBuf()`
Page 571, 1st paragraph	options to avid losing	options to avoid losing

11.3.8 Dictionary Trace Records

Page 575, 4th paragraph	`OS_onFlush()`	`QS_onFlush()`

11.3.10 Porting and Configuring QS

Page 580, 2nd paragraph	QK, C/OS-II, and Linux	QK, uC/OS-II, and Linux
Page 580, 3rd paragraph	functions such as `QS_onInit()`	functions such as `QS_onStartup()`
Page 580, Listing 11.11 caption	`qp_port.h`	`qs_port.h`
Page 581, explanation section (5)	`qf_port.h`	`qs_port.h`

11.5 Exporting Trace Data to MATLAB

11.5.3 MATLAB Script qspy.m

Location	Is	Should be
Page 591, Listing 11.14	`% sate entry/exit`	`% state entry/exit`

11.5.4 MATLAB Matrices Generated by QSPY

Page 594, Table 11.4 Header (2nd row)	Timesstamp	Timestamp
Page 595, explanation section (2)	`l_pholo_0_`	`l_philo_0_`

11.6 Adding QS Software Tracing to a QP Application

11.6.3 Generating QS Timestamps with the QS_onGetTime() Callback

Page 601, 5th paragraph	8284 timer/counter	8254 timer/counter
Page 601, 6th paragraph	8284 timer/counter	8254 timer/counter

11.6.4 Generating QS Dictionary Records from Active Objects

Page 605, explanation section (10)	so it does need to have	so it does not need to have

11.6.5 Adding Application-Specific Trace Records

Page 607, explanation section (3)	formatted as 1 using one digit	formatted as using one digit

CHAPTER 12 QP-nano: How Small Can You Go?

Location	Is	Should be
Page 611, 2nd paragraph	[Turely 02]	[Turley 02]

12.2 Implementing "Fly 'n' Shoot" Example with QP-nano

12.2.1 The main() Function

Page 617, 4th paragraph	The order or the active object control blocks	The order of the active object control blocks
Page 618, explanation section (13-15)	function must first explicitly calls	function must first explicitly call

12.2.2 The qpn_port.h Header File

Page 619, explanation section (8)	The `qpn_port.h` must include	The `qpn_port.h` header file must include
Page 619, explanation section (9)	The `qpn_port.h` must include	The `qpn_port.h` header file must include

12.2.3 Signals, Events , and Active Objects in the "Fly 'n' Shoot" Game

Page 620, 3rd paragraph	Listing 12.2(2)	Listing 12.2(1)
Page 621, explanation section (3-5)	Listing 12.1(12)	Listing 12.1(7)

12.2.4 Implementing the Ship Active Object in QP-nano

Page 626, explanation section (15)	overflow the dynamic range	overflow the range

12.2.5 Time Events in QP-nano

Page 626 & 627: Listing 12.5, three occurrences	`return (QState)0;`	`return Q_HANDLED();`
Page 627: Listing 12.5, next to last line	`return (QState)&Tunnel_active;`	`return Q_SUPER(&Tunnel_active);`

12.2.7 Building the "Fly 'n' Shoot" QP-nano Application

Location	Is	Should be
Page 630, last paragraph	C:\software\qpn	<qp>\qpn

12.3 QP-nano Structure

Page 631, 1st paragraph	derivation of concrete active objects	derivation of concrete active object classes
Page 632, Figure 12.3, in the "QHsm" class	QState Handler	QStateHandler
Page 632, last paragraph	Every QP-application	Every QP-nano application

12.3.1 QP-nano Source Code, Examples, and Documentation

Page 633, Listing 12.7	Platform-specific QP examples	Platform-specific QP-nano examples
Page 633, Listing 12.7	+-main.c -	+-main.c - main() entry point
Page 634, Listing 12.7	- QP-nano Reference Manual"	- "QP-nano Reference Manual"

12.3.4 Active Objects in QP-nano

Page 640, 3rd paragraph	deriving application-specific active objects	deriving application-specific active object classes

12.4 Event Queues in QP-nano

12.4.2 Posting Events from the Task Level (QActive_post())

Page 648, (just above (9))	Such as global variable	Such a global variable

12.4.3 Posting Events from the ISR Level (QActive_postISR())

Page 650, explanation section (3)	The advance policy	The advanced policy

12.5 The Cooperative "Vanilla" Kernel QP-nano

Location	Is	Should be
Page 652, section (2)	log2(bmask)	log2(bitmask)
Page 652, section (3)	`QF_readSet_`	`QF_readySet_`

12.6 The Preemptive Run-to-Completion QK-nano Kernel

12.6.1 QK-nano Interface qkn.h

Page 657, explanation section (6)	The `QK_SCHEDULE_()` encapsulates	The `QK_SCHEDULE_()` macro encapsulates

12.6.2 Starting Active Objects and the QK-nano Idle Loop

Page 659, explanation section (6)	All active objects in the application are initialized, exactly the same way as in 12.16(6--11).	All active objects in the application are initialized, the same way as in 12.16(6--11).

12.6.3 The QK-nano Scheduler

Page 661, Listing 12.19, the comment between (12) and (13)	set cb and a again	set cb and act again

12.7 The PELICAN Crossing Example

12.7.1 PELICAN Crossing State Machine

Page 669, section (3)	sperstate	superstate

12.7.2 The Pedestrian Active Object

Page 671, 5th paragraph	`PED_WAITING`	`PEDS_WAITING`
Page 671, last paragraph	`PED_WAITING`	`PEDS_WAITING`

12.7.3 QP-nano Memory Usage

Location	Is	Should be
Page 675, NOTE	When you apply low-power mode **is** MSP430	When you apply low-power mode in the MSP430

APPENDIX B Guide to Notation

B.1 Class Diagrams

Page 686, 2nd paragraph	Figure B.1C	Figure B.1(C)

Bibliography

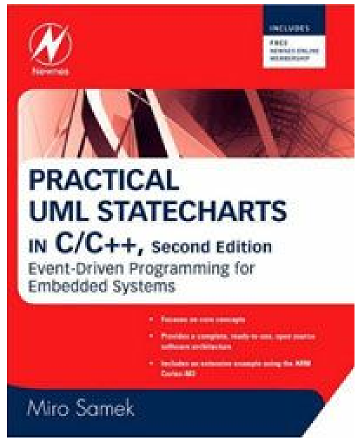
Page 693, 5th entry	[Butenhof 97] ... [Butenhof 97] ...	[Butenhof 97] … (unintended repetition)
Page 695, two occurrences on the same line	Kerninghan	Kernighan
Page 695		[Meyer 97b] Bertrand Meyer. Letters from readers (response to the article "Put it in the contract: The lessons of Ariane" by Jena-Marc J\'ez\'equel, and Bertrand Meyer). IEEE Computer, 30(2):8--9, 11, 1997.
Page 696	Rambaugh, James	Rumbaugh, James

Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008

