UML 状态图的实用 C/C++设计

嵌入式系统的事件驱动编程技术

第二版

MIRO SAMEK 著
anymcu@gmail.com 译

The real voyage of discovery consists not in seeking new landscapes but in having new eyes.

--Marcel Proust

前言

为了产生一个可用的软件，你不得不和每个修改，每个特征，每个微小的调节做斗争直到别人满意为止。没有捷径可走。有运气的成分，但你的成功不是因为幸运而是你尽力争取到的。

--Dave Winer

许多年以来，我在寻找一本书或者杂志，它介绍如何用主流的编程语言比如 C/C++来进行完全实际可用的现代状态机（UML 1 状态图）编程。我没有找到这种技术。

2002 年，我写了《PSICC：嵌入式系统量子编程》。它作为当时第一本这个领域的书，介绍了长久以来被期待的技术—实现了一个紧凑的，高效率的，高度可移植的 UML 状态机，并对状态的层次式嵌套提供了完全的支持。PSICC 也是第一个为嵌入式系统提供一套完整的，通用的，基于状态机的实时应用程序框架的 C/C++源代码。

据我所知，在嵌入式系统编程方面 PSICC 一直是关于状态图和事件驱动编程技术的最流行的书籍之一。在出版后几年内，PSICC 被翻译成中文，之后一年内又被翻译成韩文。我收到并回答了几千封从世界各地读者的电子邮件，他们在消费，医疗，工业，无线，网络，研究，国防，机器人，汽车空间探索和许多其他的应用中成功的使用了书中的代码。从 2003 年开始，我在美国东西两岸的嵌入式系统会议演讲这个论题。我也开始为许多公司提供顾问服务。这些活动给我提供了许多机会去现场了解工程师是如何在一个广泛的应用领域使用已发行代码的。

你现在在阅读的是 PSICC 的第二版。它是我接收到的大量的反馈和过去 5 年“大规模并行测试”及“现场实战调查”的直接产物。

本书第二版的新增内容

如我在 PSICC 第一版中所承诺的，我持续的改进了代码和优化设计技术。这份完全修订过的第二版采纳了这些先进技术和大量从读者中得到的经验。

全新的代码

首先，本书提供了一套全新的软件“量子平台 QP”，它包含层次式事件处理器 QEP 和实时框架 QF，以及 2 个新的组件。从 6 年前本书第一版发行后 QP 经历了几次重要的改进。PSICC 第一版发行后引进的改进太多不能在前言中一一列举。改进的主要方面包括更高的效率，可测试性，和在不同处理器，编译器和操作系统中更好的移植性。2 个新的 QP 组件是在第十章描叙的轻量级可抢占实时内核 QK 和在第十一章描叙的软件跟踪装置 QS。最后，我十分兴奋能引入一个全新的超轻量级简化版 QP--为从最低端的 8 位直到 16 位 MCU 而量身定做的 QP-NANO。我在第十二章描叙 QP-NANO。

开源和双许可证策略

2004 年,我决定把全部 QP 代码在自由软件基金的 GNU 通用公众许可 GPL 第 2 版条款下发行为开源软件。同时，QP 源代码也可以在独立的传统商业许可下使用。商业许可可以替代 GPL，这对那些需要保护他们基于 QP 的资产状态的用户特别设计的。把开源和使用许可结合起来的策略在逐步流行，被称为双许可制，在附件 A 中有对它的详细解释。

以 C 语言为主要的开发语言

PSICC 第一版中的绝大多数代码实例是以 C++实现的。然而，我在现场发现，很多嵌入式软件开发者都有硬件工作背景（绝大多数是电子工程师）而且他们常常不喜欢使用 C++。

在这版中，我决定交换 C 和 C++的角色。和从前一样，配套网站有 C 和 C++版本的完整源程序。但是现在，书中的绝大多数代码实例使用 C 版本，仅在以 C++和 C 的实现代码之间的差别显著和重要时，才会讨论 C++代码的实例。

由于使用 C 代码，我不再使用在第一版中应用和介绍的 C+面向对象的扩展。代码继续和 C+兼容但是 C+宏已不再使用。

更多的实例

和第一版相比，这版提供了更多的事件驱动系统的实例，这些例子更加完整。我尽力使实例不会琐碎同时也不会把理论细节搞的模糊不清。同时，我所选的实例不需要任何特定方面的知识，从而我不用浪费纸面和你的注意力去解释问题规范。

支持可抢占式多任务调度

QP 一类的事件驱动的底部构造，可以和不同的并发机制一起工作，从简单的“超级循环”到完全可抢占的，基于优先级的多任务调度。QP 的前版本天然的支持简单的非可抢占调度，但是如果需要这个功能的话，它需要一个外部的 RTOS 来提供可抢占的多任务调度。

在第十章，我描叙了这个新的实时内核 QK 组件，它可以提供可确定的，完全可抢占的基于优先级的调度给 QP。QK 是一个很特殊的，非常简单的，运行-到-完成，单堆栈的内核，完美的符合对状态机执行时的运行-到-完成语义的众所周知的设定。

支持全面的测试

建立在并行执行状态机上的运行着的应用程序是个高度结构化的事务，在里面所有重要的系统交互作用都是通过把所有状态机链接起来的事件驱动框架所提供的管道来进行的。通过监视这些“管道”代码，你可以获得对运行系统的深入的了解。事实上，从一个被监视的事件驱动框架上得到的跟踪数据可以告诉你更多关于应用程序的信息，超过其他任何传统的实时操作系统 RTOS，因为现在框架“知道”非常多关于应用程序。

在第十一章，我描叙了新的 QS（spy）组件，它为 QP 事件驱动平台提供了一个丰富的软件跟踪装置。QS 产生的跟踪数据允许你只用花费最少的目标系统资源，不用停止或显著的降低代码执行的情况下，对你在运行中的实时嵌入式系统进行现场分析。和其他工具一起，你可以重新构建你系统中全部主动对象的完整的序列图和详细的，带有时间戳的状态机活动行为。你可以监视所有的事件交换，事件队列，事件池，时间事件（定时器），和可抢占性，上下文改变。你也可以用 QP 添加你自己的装置到应用层代码中。

超轻量级 QP-NANO 版本

使用状态机的事件驱动的解决方法和其他常规的实时内核或实时操作系统相比，更加精简。为了使用在非常小的嵌入式系统，QP 的简化版 QP-NANO，实现了 QP/C 或 QP/C++特性的一个子集。QP-NANO 被特别设计为可在低端 8 和 16 位微控制器，比如 AVR, MSP430,8051,PICMICRO,68HC(S)08,M16C 和其他型号上，使用层次式状态机实现事件驱动编程。典型的 QP-NANO 需要大约 1-2KB 字节 ROM 和每个状态机及字节的 RAM。我在第十二章描叙 QP-NANO。

取消了量子比喻

在 PSICC 第一版中我提供了利用量子机制的比喻来理解事件驱动系统。虽然我仍然相信这个比喻十分精确，而且提供这样一个比喻是极限编程 XP 和敏捷方法的关键方法，但是却它并不吸引读者。

根据读者的反馈，我决定从这版本中移去量子比喻。由于历史的原因，“量子”一词继续在软件中出现，前缀”Q”在代码中用来标识类型和函数名，以便把 QP 和其他代码区分，但是你不需要对这些名字想更多。

使用 QP 所需的条件

本书提供的绝大多数代码使用高度可移植的 C 或 C++，独立于任何特定的 CPU，操作系统或编译器。然而，为了便于讨论，我提供了运行于任何版本 WINDOWS 操作系统下 DOS-窗口上的可执行实例。选择这个古老的 16 位 DOS 作为演示平台的理由是，它允许在一台标准的 X86 电脑上从硬件层开始编程，并直接操作 IO 空间。没有别的其他现代 32 位标准电脑开发环境可以如此容易的做到这点。

使用古老的 DOS 平台的其他优点是可以用到许多成熟的免费工具。由于这点，我使用了古老的 BORLAND C++ 1.01 工具套件编译了所有的实例，这个套件可以从 BORLAND 免费下载。

为了演示使用 QP 在现代嵌入式系统上编程，我也提供了实例，使用从 LUMINARY MICRO 公司的便宜的1基于 ARM CORTEX-M3 的 STELLARIS 芯片的 EKI-LM3S811 评估套件。CORTEX-M3 实例使用和 DOS 平台实例完全相同的源代码，只是 BSP 稍有不同。CORTEX-M3 实例需要 32KB 限制的起步版本的 IAR EWARM 工具套件，它包含在 STELLARIS 套件中，也可以从 IAR 免费下载。

最后，书中的一些实例运行于 LINUX，和任何其他 POSIX 兼容的操作系统比如 BSD,QNX, MAX OS X,或 SOLARIS。你也可以在 CYGWIN 环境下在 WINDOWS 上生成这些 LINUX 实例。

本书的配套网站 WWW.QUANTUM-LEAPS.COM/PSICC2 提供了书中使用工具和其他资源的所有连接。网站也包含了大量的 QP 移植到不同的 CPU,操作系统和编译器的代码的连接。请经常查看这个网站，因为新的移植代码常常增添。

特定的读者

1在本书的写作期间，EKI-LM3S811 的价格是 49 美元 (www.luminarymicro.com)

本书特意为以下对事件驱动编程和现代状态机感兴趣的软件开发者：

1. 嵌入式程序员和顾问会发现一个完整的，立即可以用的事件驱动架构去开发应用系统。本书描叙了状态机编码策略，和同样重要的一个与之配套的执行并发状态机的实时框架。这两个因素是互补的，离开了对方每个因素都不能发挥它的最大潜能。
2. 寻求一个实时内核或实时操作系统的嵌入式系统开发者会发现 QP 事件驱动平台可以做到 RTOS 可以做到的任何事情，而且事实上，QP 包含了一个完全可抢占的实时内核和一个简单的协作式调度器。
3. 无线传感器网络等超低功耗系统的设计者会发现如何把基于事件驱动状态机的解决方案裁剪以适合最小的微控制器。超轻量级的 QP-NANO 版本（第十二章）在仅 1-2KB 字节的 ROM 中融合了一个层次式事件处理器，一个实时框架和一个协作式或完全可抢占式的内核。
4. 对于复杂性的应用，大规模大型并行服务应用的设计者会发现事件驱动的解决方法结合层次式状态机很容易扩展，在管理非常大数目的状态化组件例如客户任务方面，非常理想。可以证明，QP 的嵌入式设计理念对每个组件的时间和空间性能提供了关键的的支持。
5. 开源社区会发现 QP 补充了其他的开源软件，比如 LINUX 或 BSD。QP 到 LINUX（和 POSIX 兼容的操作系统）的移植在第八章描叙。
6. 使用 C 或 C++的图形用户界面开发者和计算机游戏程序员会发现 QP 很优雅的补充了 GUI 库。QP 提供了高层的基于层次式状态机的“屏幕逻辑”，而由 GUI 库处理底层的组件并在屏幕上画图。
7. 系统架构师会发现 QP 是大型自动化设计工具的一个超轻量化的替代。
8. 自动化设计工具的用户会获得对他们使用工具的内部工作方法更深的理解。看到“帽子下面”的内容会帮助他们更加自信的去更有效的使用工具。

由于代码为中心的解决方法，本书主要吸引那些需要创造实际的可工作代码而不是仅仅建模的软件开发者。许多其他关于 UML 的书籍已经在描叙模型驱动分析和设计和其他相关问题比如软件开发过程和建模工具方面，做了很好的工作。

本书不提供另一个 CASE 工具。相反，本书是关于对层次上状态机的实用的，可维护的编码技术和使用一个实时框架把状态机和坚固的事件驱动系统结合起来。

为了从书中有所收获，你必须合理的通晓 C 或 C++语言，并对计算机结果有个大体的理解。我没预设你对 UML 状态机有初步的了解，在第二章我将对底层概念有速成课程的解释。我会在第六章介绍基本的多任务，互斥，和阻塞等实时概念。

本书的网站

本书的网站是 HTTP://WWW.QUANTUM-LEAPS.COM/PSICC2,包含如下信息：

- QP/C,QP/C++和 QP-NANO 的源代码下载
- 本书中描叙的全部 QP 移植代码和实例
- HTML 和 CHM 格式的 QP/C,QP/C++和 QP-NANO 的参考手册
- 本书中使用的编译器和其他工具的下载连接

- 读者反馈和评论精选
- 勘误表

另外，HTTP://WWW.QUANTUM-LEAPS.COM 在 2002 年发行后就一直支持 QP 用户社区。网站提供如下资源：

- 最新的 QP 下载
- QP 移植代码和开发套件
- 程序员手册
- 应用笔记
- 资源和小工具，比如 VISIO 的 UML 图的模板，设计模式，其他有关书和文章的连接等
- 商业许可和技术支持信息
- 技术的顾问和培训
- 新闻和事件
- 论坛
- 时事快报
- 博客
- 其他有关网站的连接
- 其他

最后，QP 也被放在 SOURCEFORGE.NET—世界上最大的开源代码和应用数据库里。QP 项目位于：http://sourceforge.net/projects/qpc/.

致谢

首先，我要感谢我的家庭，他们对这些年来编写软件和本书的两个版本所作出的不可磨灭的支持。
我也要感谢 ELSEVIER 的团队，RACHEL ROUMELIOTIS, HEATHER SCHERER, 和 JOHN(JAY) DONAHUE。

最后，我想感谢所有联系我的软件开发者，他们给我启发性的思考的问题，BUG 报告，无数改善代码和文档的建议。作为一个规则，一个软件系统只有在它在被不同的用户在不同的实际项目中使用和检查后才会变得更好。

MIRO SAMEK
CHAPEL HILL, NORHT CAROLINA
APRIL 2008

介绍

几乎所有计算机系统，特别是嵌入式系统，是事件驱动型的，这意味著它们持续等待某些外部或者内部的事件发生，比如一个时钟 TICK，一个数据包的到来，一个按键被按下，或者一次鼠标的点击。确认事件后这类系统产生相应的反应，去执行相应的计算，去操作硬件或者，去产生“软”事件去触发其他的内部软件组件。（这就是为什么事件驱动型系统也被称作为***反应***系统的原因）。一旦完成了事件处理，软件退回到等待下一个事件发生的状态。

你无疑肯定熟悉基本的顺序控制，使用这种方法，一个程序在它执行路径的不同地方等待事件，它或者主动的轮询事件，或者被动的阻塞于一个旗语或其他的操作系统原语。尽管这种事件驱动系统的编程方法在很多情况下起作用，但是，当系统有许多可能的事件源，而你也不能预测事件的到达时间和次序，而且及时处理事件变得至关重要时，这种方法不能很好的工作。问题在于当连续化程序在等待某类事件时，它没做任何其他工作，也不对其他事件起反应。

明显的，我们需要的是一个程序结构，它可以对不同的可能事件反应，任何事件可以在不能预测的时间以不能预测的次序到达。在嵌入式系统，比如家用电器，手机，工业控制器，医疗设备和其他系统中，这个问题非常普遍。在现代桌面计算机中，这个问题也很突出。比如在使用一个网页浏览器，文字处理器，或者速算表时。绝大多数这些程序有个现代的图形用户界面 GUI，它明显的可以处理许多事件。所有当代的 GUI 系统以及许多嵌入式应用，都采用了一个共同的程序结构，可以优雅的解决需要及时的处理异步事件的难题。这种程序结构一般被称为“事件驱动式编程”。

反向控制

事件驱动式编程方法需要一个完全不同的思考方式，它和传统的连续化编程方法，例如“超级循环”或传统的 RTOS 的任务不同。绝大多数的现代事件驱动式系统根据好莱坞原则被构造，“不要调用我们，我们会调用您”(“Don’t call us, we will call you.)。因此，当它等待一个事件时，这个事件驱动式系统没有控制权，事实上，它甚至没有被激活。仅当一个事件到达了，程序被调用去处理这个事件，然后它又很快的放弃控制权。这种安排允许这个事件驱动式系统同时等待许多事件，结果系统对所有需要处理的事件都能保持反应。

这个方案有三个重要的后果。第一，它意味著一个事件驱动式系统被自然的分解到应用程序，由应用程序处理事件，而它的监督者是事件驱动的平台，由它等待事件并把它们分发给应用程序。第二，控制存在于事件驱动平台的架构中，因此从应用程序的角度看，和传统的顺序式程序相比，控制被反转了。第三，事件驱动的应用程序必须在处理完每个事件后交还控制权，因此和顺序式程序不同的是，运行时上下文和程序计数器不能被保留在基于堆栈的变量中。相反，事件驱动应用程序变成了一个状态机，或实际上一组合作的状态机，并在静态变量里保留从一个事件到下一个事件的上下文。

事件驱动式框架的重要性

控制反转，在所有事件驱动系统中如此典型，因此它使事件驱动下部构造具有了一个应用程序框架的全部特性，而不是一个工具集。当你使用工具集时，比如使用传统的操作系统或 RTOS，你编写应用程序的主

要部分，当你需要复用时，你调用工具集。当你使用框架时，你复用应用程序的主要部分，而只需编写它调用的代码。

另一个重点是，如果你想把许多事件驱动型状态机组合到一个系统中，事件驱动框架是必需的。要执行并发状态机，不能仅仅是像使用传统的 RTOS 那样只需调用 API。状态机需要一个底层架构（框架）为每一个状态机，事件队列，和基于事件的时间服务提供最小的，运行-到-完成的执行上下文。这点非常关键。状态机不能在“真空”中操作，而且如果没有一个事件驱动框架支持，它就没有实用性。

主动对象计算模型

本书提供了分析事件驱动系统的两个最有效的技术：层次式状态机和事件驱动框架。这两种要素的结合被称为主动对象计算模型。术语“主动对象”是从 UML 而来，表示一个自治的对象，和其他主动对象通过事件进行异步结合。UML 还提议使用状态图的 UML 变体为事件驱动型主动对象的行为建模。

本书中，主动对象使用事件驱动框架 QF 实现，QF 是 QP 事件驱动平台的主要组件。QF 框架有次序的执行主动对象，处理全部线程安全的事件交换细节和主动对象的内部处理过程。QF 保证状态机执行预设的运行-到-完成语义，通过对事件进行排队并依次派发它们（每次一个）给主动对象的内部状态机。

层次式状态机和事件驱动框架结合的概念基础早已有之。事实上，他们已经被广泛的使用了最少 20 年。几乎今天市场上全部成功的设计自动化工具都是基于层次式状态机（状态图）并在内部包含一个类似 QF 事件驱动的实时框架。

以代码为中心的开发方式

本书所预设的实现方法是以代码为中心的，简约的和底层的。这个特证不是贬义的，它只是意味你将会学到在不用巨型工具的情况下如何把层次式状态机和主动对象直接映射成 C 或者 C++代码。问题的关键不是工具—问题是“理解”。

现代化的设计自动化工具确实很强大，但是它们不适合所有人。对有些开发者来说，这些工具太庞大所以被放弃了。本书展示的以代码为中心的实现方法为这些开发者提供了一个重量级工具的轻量级替代品。

然而最重要的是，没有工具能替代对概念的理解。例如，在某个关键的状态转换中确定哪个出口\入口动作以什么序列执行，这些不是你能通过运行设计工具显示的动画能发现的。答案来自于你对底层状态机实现（在第三，四章讨论）的理解。即使你决定以后使用自动化设计工具，即使它使用和本书不同的状态图实现技术，因为你对底层实现机理的理解你将会有更大的信心和更有效率。

尽管现有用户给我许多压力，我坚持保持 QP 事件驱动平台简洁，只直接实现庞大的 UML 规格的必要元素从而支持最精细的设计模式。保持内核实现小而简单有现实的好处。程序员可以很快的学习并部署 QP 而不需在工具和培训上大量投资。他们能很容易的根据实际情况改写和定制框架的源代码，包括资源约束很严的嵌入式系统。他们可以理解并实际使用所有提供的特证。

关注实际的问题

状态机和事件驱动框架不能仅仅被看作是一些特征的集合，因为有些特征被隔离开来后并没有意义。只有你思考设计而不是编码时，你才能有效的使用这些强大的概念。为了从这个角度理解状态机，你必须从事件驱动编程的角度来理解问题。本书讨论事件驱动编程的问题，为什么他们是问题，以及状态机和主动对象计算模式如何帮助解决这个问题。因此，许多章节我从要讨论的编程问题开始。这样，我希望可以使你

前进，每次一小步，直到最后层次式状态机和事件驱动框架成为你的解决问题的自然地思考方法，并放弃那种使用传统的多层嵌套的 IF 和 ELSE 编程方式，或者在传统的 RTOS 中通过旗语和事件标志来传递事件的方法。

面向对象

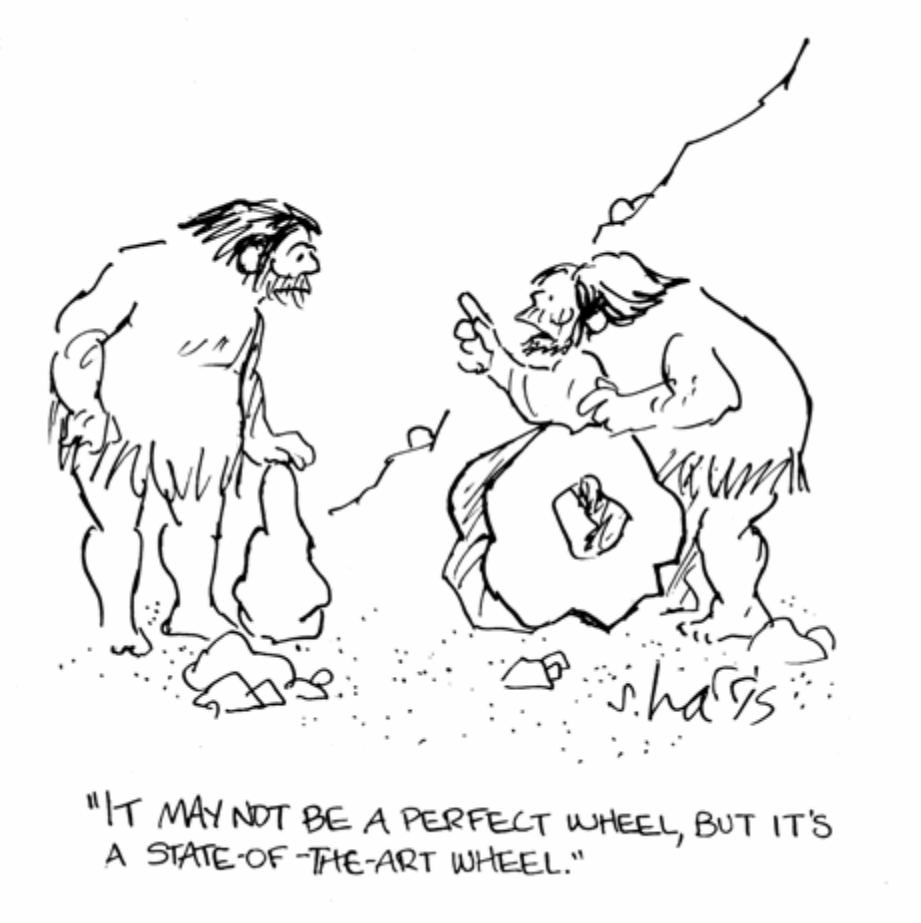
即使我使用 C 作为主要的编程语言，我也彻底地使用面向对象的设计原则。同实际上所有的应用框架一样，QP 使用基本的封装概念（类 class）和单一继承作为主要的定制，特例化和扩展框架到实际应用的实现机制。请不要担心这些概念可能对你太新了，特别是使用 C 语言的读者。在 C 语言的层次，封装和继承是很简单的编码方法，我在第一章会介绍它们。在 C 版本里我明确的避免了多态性，因为在 C 里实现后绑定有些复杂。当然，C++版本直接使用类和继承，QP/C++应用程序可以直接使用多态性。

更加有趣味性

当你开始使用本书描叙的技术时，你的问题将会改变。你不再纠缠于 15 层复杂的 IF-ELSE 语句，你停止担忧旗语，或其他底层的 RTOS 机理。相反，你开始在更高的层次式状态机，事件和主动对象的抽象来思考问题。当你体验了这种飞跃，你将发现，和我已经体验的一样，编程可以更有趣。你再也不会想回到“意大利面条”式的代码或者简陋的 RTOS 里去了。

如何联系作者？

如果你对本书，本书的代码或一般的事件驱动编程方法有任何意见或者问题，我很高兴能听到它们。请给我发电子邮件，miro@quantum-leaps.com



www.CartoonStock.com

第一部分 UML 状态机

状态机是最被熟知的界定和实现必须及时的对外界输入事件反应的事件驱动系统的方法。先进的 UML 状态机展示了当前发展的状态机理论和表示法。

本书第一部分展示了如何在事件驱动应用程序中实际使用 UML 状态机的方法，帮助你写出高效的可维护的软件，有被充分理解的表现，而不是制造有着复杂的 IF-ELSE 的“面条”代码。第一章使用一个工作实例介绍这个方法的概况。第二章介绍状态机的概念和 UML 的表示方法。第三章展示编码状态机的标准方法，第四章介绍一个通用的层次式事件处理器。第五章介绍五状态设计模式的步骤表。你将了解到 UML 状态机，即使你没有复杂的代码合成工具，也是你可以使用的一个强大的设计方法。

第 1 章

UML 状态机和事件驱动式编程技术初探

作为常识，我们采取一种方法并尝试它，如果它失败了，坦率的承认这点再去试别的办法。但最重要的是，去试一些方法。

--Franklin D. Roosevelt

本章展示一个完全使用 UML 状态机和事件驱动的应用项目实例。这个实例应用程序是一个交互式“飞行射击”游戏，我决定尽早放在本书的开始处这样你可以尽快的开始“玩”代码。本章的目的在于用一个实际的实用程序来展示这个方法的基本要素，但不会陷入到细节，规则和例外里面。以此为出发点，我不试着巨细无遗，但是这个实例和本书其他的实例都展示了一个好的设计和推荐的编码风格。我不假设你已经知道很多关于 UML 状态机，UML 表示法或者事件驱动式编程。我将在需要时简要介绍这些概念或者告诉你如何在本书其他地方寻找详细的解释。

“飞行射击”游戏是基于 STELLARIS 的演示版套件中 QUICKSTART 应用软件的源码[Luminary 06]。我尽量使”飞行射击“游戏的表现十分接近原来的 QUICKSTART 应用程序，这样你就可以在相同的问题规范上直接比较事件驱动编程方法和传统的解决方法。

1.1 安装本书代码

本书配套网站 http://www.quantum-leaps.com/psicc2 有自解压文件，包含 QP 事件驱动平台和本书中所有实例的可执行的完整源代码及文档，开发工具，资源和其他。你可以把文件解压到任何目录下。你选择的安装目录被作为 QP 的根目录<qp>。

注：尽管本书中我主要使用 C 语言实现代码，本书网站上也包含等效的 C++版本的实现。C++代码和相应的 C 代码是同样的目录结构，只是它们位于<qp>\qpcpp\...目录下。

对于“飞行射击”游戏实例，附带代码有 2 个版本。我提供了一个标准的基于 WINDOWS 的 DOS 窗口的实现（图 1.1）这样你不需要任何特别的嵌入式系统板就可以玩这个游戏并实验这些代码。

注：我挑选古老的 16 位 DOS 平台的原因是因为它可以使你在一台标准的 PC 上从硬件层开始编程。不用离开你的书桌，你可以使用中断，直接操作 CPU 寄存器，直接存取 I/O 空间。其他现代的 32 位标准 PC 开发平台不容易做到这点。这个独特运行在 DOS 的（或者其他版本 WINDOWS 的 DOS）上 PC 模拟了一台传统的 80X86 硬件。而且，你可以使用免费的成熟的工具比如 BORLAND C/C++ 编译器。

我也为便宜的基于 ARM CORTEX-M3 的 STELLARIS EV-LM3S811 演示版提供了游戏的一个嵌入式版本（图 1.2）。PC 版和 CORTEX-M3 版使用完全相同的应用程序源代码，唯一的差别在于版支持包不用 BSP。

1.2 开始玩吧

下面对“飞行射击”游戏的描叙有 2 个目的,首先是解释如何玩这个游戏,再就是作为问题规范用来在本章后面设计和实现软件.为达成这 2 个目标,我需要详细的解释,所以请耐心的跟我一起.

你在游戏中的目标是带领一艘太空飞船通过一个没有尽头的，布满鱼雷的水平隧道.飞船和隧道或者鱼雷的任何碰撞都会毁灭飞船。你可以使用 PC 键盘上的 UP 或者 DOWN 箭头按键使飞船向上或向下(图 1.1)或者 EV-LM3S811 板上的旋钮。你也可以按 PC 键盘的空格键或者 EV-LM3S811 上的 USER 按钮发射一枚导弹消灭隧道里的地雷。分数根据游戏的时间(速率是每秒 30 点)和炸毁地雷的数量累积的，每次游戏只有一艘飞船。

游戏开始时是在演示模式，隧道以正常的速度从右到左转动，“Press Button”文字在屏幕中间闪烁。你需要产生”fire missile”事件让游戏开始（在 PC 上按空格键，在 EV-LM3S811 板上按 USER 按钮）。

在飞行时你每次只能有一枚导弹，因此当导弹在飞行时你想发射导弹是无效的。导弹击中隧道墙壁没有分数，但是击毁鱼雷可以得分。

游戏中有 2 种不同功能的鱼雷。在原版 LUMINARY “QUICKSTART”应用程序中，2 中鱼雷表现一样，但是我想演示状态机如何能优雅的处理不同表现的鱼雷。

鱼雷一型较小，但是可以被导弹击中它的任何像素点而被击毁。击毁鱼雷一型你获得 25 点。鱼雷二型较大，但是比较狡猾，只有击中它的中心部分而不是触角才能击毁它。当然，飞船对鱼雷的任何一点都是易被攻击的。击毁鱼雷二型你获得 45 点

当你碰到墙壁或者鱼雷而损毁掉飞船时，游戏结束，显示闪烁的“Game Over”文字和你的最后得分。闪烁 5 秒后，”Game Over”屏幕变回演示画面，等待从新开始游戏。

另外，这个应用程序带有一个屏幕保护，因为原 EV-LM3S811 板的 OLED 显示器有和 CRT 类似的烧毁特性。如果在演示模式 20 秒内没有开始游戏屏保就被激活（比如，屏保在游戏过程中不会出现）。屏保是个简单的随机像素类的显示，和 EV-LMS3S811 版的应用程序”Game of Live”算法一样的。我决定简化这个实现，因为混合像素算法不能带给我们别的新的或有趣的表现。在运行了屏保一分钟后，显示器被清空，只有一个随机点出现在屏幕上。这一点和原版”QUICKSTART‘应用程序不一样，原程序清空屏幕并开始显示 USER LED。我改变了这个表现因为 USER LED 有更好的用途（去显示 IDLE 循环的活动状态）。

图 1 运行在 WINDOWS XP 的 DOS 窗口中的“飞行射击”游戏

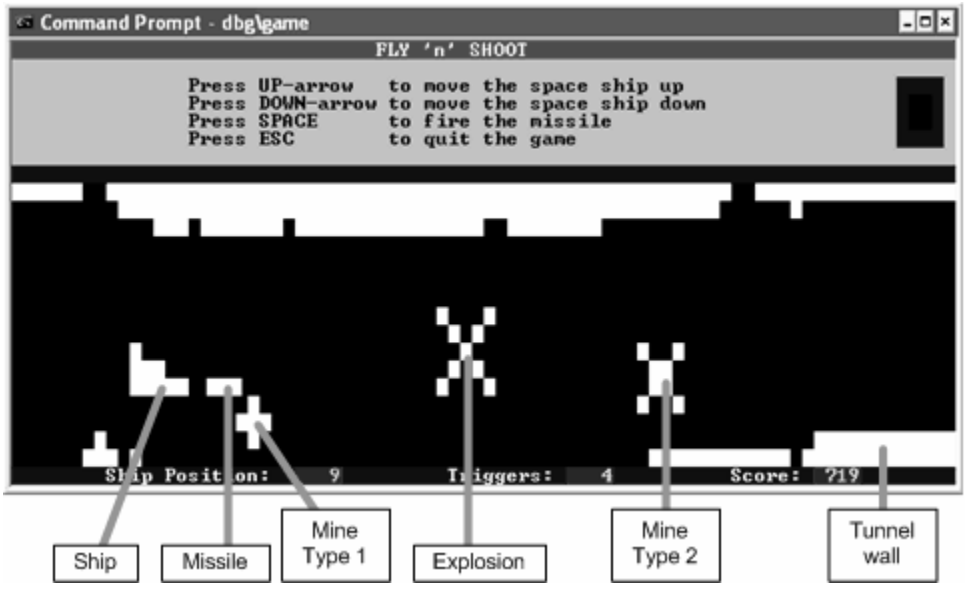
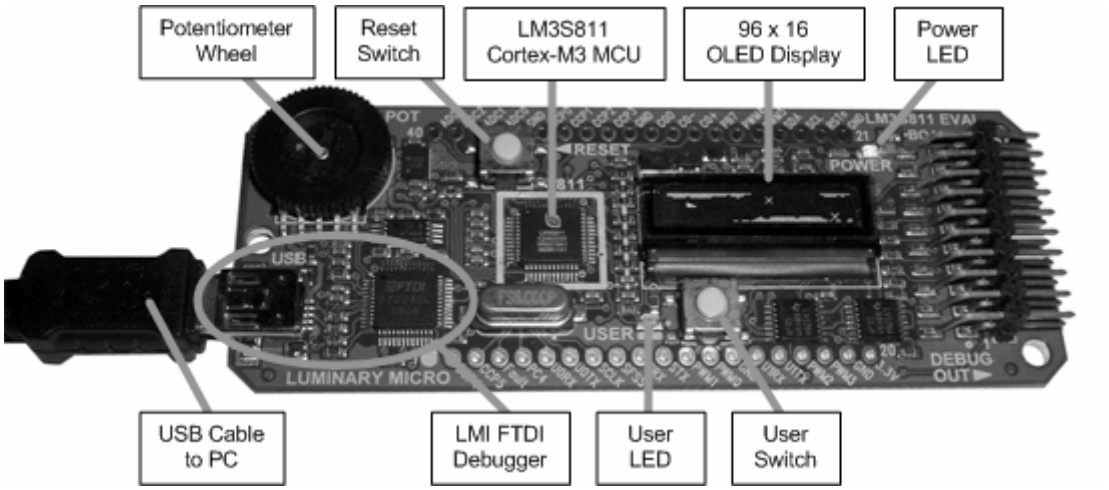


图 2 运行于 STELLARIS EV-LM3S811 演示板上的“飞行射击”游戏



1.2.1 运行游戏的 DOS 版本

使用 C 语言的 DOS 下的“飞行设计”游戏代码位于<qp>\qpc\examples\80x86\dos\tcpp101\l\game\,目录，这里<qp>是你安装随书软件时选择的路径。

编译后的可执行程序也提供了，你可以在任何 WINDOWS 平台的 PC 上双击位于<qp>\qpc\examples\80x86\dos\tcpp101\l\game\dbg\.目录的可执行文件 game.exe。你看到的第一个屏幕是游戏运行于演示模式，“Push Button”文字在屏幕中间闪烁。在屏幕顶部是应用程序可以接受的按键的符号。你需要按空格建开始玩游戏。要退出游戏可以按 ESC 按键。

在 WINDOWS 的窗口模式玩“飞行射击”游戏，和 STELLARIS 版本相比它的动画效果可能有点跳跃。切换到全屏模式可以明显的让游戏运行的更加流畅。按下 ALT 不放再按 ENTER 键可以切换到全屏模式。再使用 ALT-ENTER 组合键可以切换回窗口模式。

如你在图 1.1 所看到的，DOS 版的游戏使用简单的 VGA 文本模式去模拟 EV-LM3S811 演示版的 OLED 显示器。DOS 窗口的屏幕下方被当作一个 80x16 字符的“像素“矩阵，比 OLED 显示器的 96X16 像素稍微小，但是足够玩游戏了。特别的我避免在这个开始的实例里使用任何有趣的图形，因为我有一条更大的“鱼”要烤给你，这比操心无关的复杂的图形编程更重要。

我的主要目标是使你容易的理解事件驱动代码并用它做实验。为此，我选择了 Borland Turbo C++ 1.01 工具集去创建这个实例和本书的的其他几个实例。尽管 Borland Turbo C++ 1.01 是个较老的编译器，它可以演示 C 和 C++版本的所有特征。最后，它可以从 Borland 博物馆
“http://bdn.borland.com/article/0,1410,21751,00.html 免费下载.

这个工具集很容易安装。你从 Borland 下载 Turbo C++ 1.01 安装文件后，你需要把它解压到你的硬盘。然后你运行 INSTALL.EXE 并遵照它的安装提示即可。

注：我强烈推荐你把 Turbo C++1.01 工具套件安装在目录 C:\tools\tcpp101\. 这样，你可以直接使用本书所提供的项目文件和 MAKE 文件。

也许实验“飞行射击“代码的最容易的方法是运行 Turbo C++ IDE (TC.EXE)并打开位于目录 <qp>\qpc\examples\80x86\dos\tcpp101\l\game\的项目文件 GAME-DGB.PRJ.你可以在 IDE 里修改，重新编译，执行和调试。然而，你应该避免在调试器里结束程序，因为这样不会恢复标准的 DOS 下的时刻和键盘中断向量。你应该运行应用程序并用 ESC 按键来退出。

在下一节，我简要的说明如何运行游戏的嵌入式版本。如果你对 CORTEX-M3 版本不感兴趣，可以跳过 1.3 节，后面是应用代码的解释。

1.2.2 运行游戏的 Stellaris 版本

和运行在古老的实模式 80X86 处理器的 DOS 版本的“飞行设计”游戏不同，同样的代码，运行在当代工业最先进的处理器之一 ARM CORTEX-M3 上。

STELLARIS Ev-LM3S811 版本的代码位于
<qp>\qpc\examples\cortex-m3\vanilla\iar\game-ev-lm3s811\目录，<qp>代表你选择安装本书代码的根目录。

STELLARIS 套件的代码已经使用 EM-lm3s811 演示版附带的 32KB 限制版 IAR 嵌入式 ARM 工作平台（IAR EWARM）v5.11 编译.你也可以在在线注册后免费从 IAR 公司网站(www.iar.com)直接下载这个 IDE。

IAR EWARM 的安装使用它附带的安装工具是很直接的.你还需要根据 EV-LM3S811 板的文档说明安装它的硬件调试 USB 驱动.

注：我强烈的推荐你在 c:\tools\iar\arm_ks_5.11 目录安装 IAR EWARM 工具。这样，你可以直接使用本书提供的 workspace 文件和 MAKE 文件。

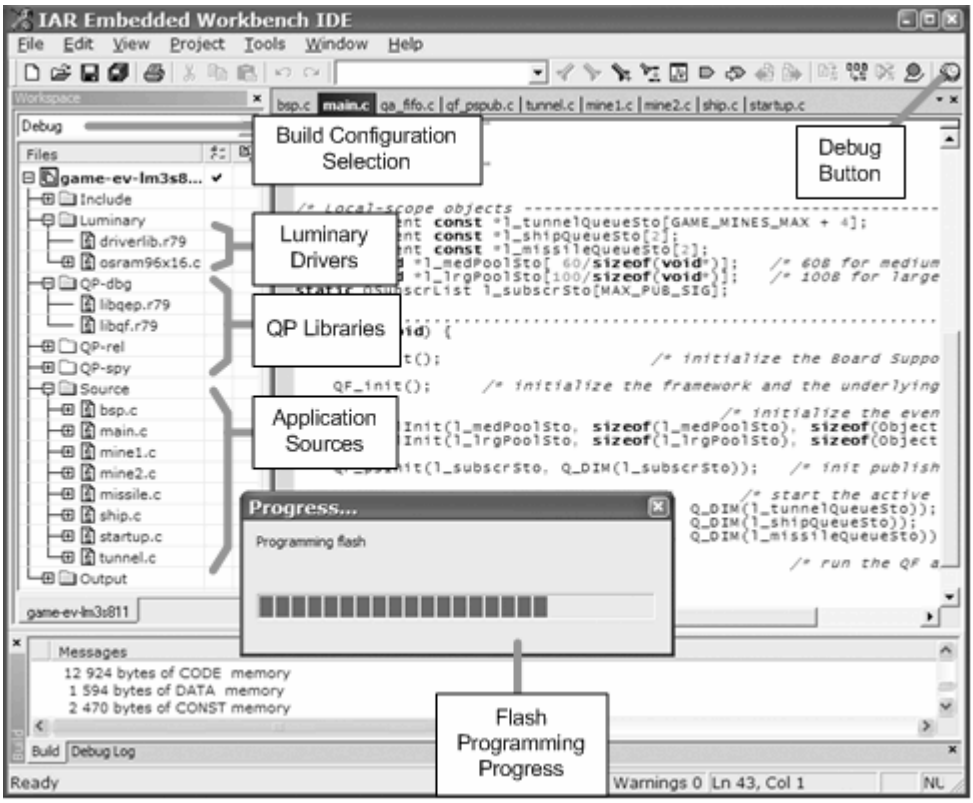
在你对 EV-LM3S811 进行“飞行射击“游戏前，你也许想先玩一会儿板上的原来的”QUICKSTART” 应用程序。

要把“飞行射击”游戏编程到 EV-LM3S811 的 FLASH 里，你首先使用套件提供的 USB 线缆连接 EM-LM3S811 和你的 PC，并确认电源 LED 点亮（图 1.2）。然后，你需要启动 IAR EWARM，打开位于 <qp>\qpc\examples\cortex-m3\vanilla\iar\game-ev-lm3s811\的 game-ev-lm3s811.eww。这时开始，你应该看到和图 1.3 类似的屏幕显示。Game-ev-lm3s811 项目设置为使用 EM-LM3S811 上的 LMI FTDI 集成调试器（图 1.2）。你可以通过 PROJECT->OPTIONS 菜单里的 OPTIONS 对话框验证这个设置。在 OPTIONS 对

话框里你需要选择左面的调试器页面，这里你可以选择 DOWNLOAD 页面来验证装载 FLASH 已被打开。选中的“Useflash loader(s)”意味着 IAR 提供的 flash loader 应用程序会首先被装入 MCU 的 RAM 中，并且这个应用程序会用你的应用程序的映像来对 FLASH 进行编程。

选择 PROJECT->DEBUG 菜单或者单击工具条上的 DEBUG 按钮（图 1.3）开始 FLASH 编程。IAR WORKBENCH 应该显示 FLASH 编程进度条，如图 1.3 所示。但 FLASH 编程结束，IAR 切换到 IAR C-SPY 调试器，程序停止在 main()入口。你可以单击调试器里的”GO“按钮或者关掉调试器，按板上的 RESET 按钮重启演示版，来开始玩游戏。这样，“飞行射击“游戏现在已经被下载到 EV-LM3S811 板，并且以后每次上电它都会自动运行。

图 3 使用 IAR EWARM IDE 装入”飞行射击”游戏到 LM3S811MCU 的 FLASH



IAR 嵌入式工作台使你很容易的实验”飞行射击“游戏的代码。你可以编辑并单击 F7 按钮编译应用程序。唯一的要求是在安装 IAR 工具集后你需要从源代码生成 LUMINARY MICRO 驱动库。在 <IAR-EWARM>\ARM\examples\Luminary\Stellaris\boards\ek-lm3s811 目录装入 ek-lm3s811.eww 工作台文件，<IAR-EWARM>是你安装 IAR 工具集的目录。在 ev-lm3s811.eww 工作台，你选择上方下拉列表的“driverlib – debug”项目，再按 F7 生成库文件。

1.3 main()函数

也许开始解释“飞行射击”游戏代码的最佳地方是位于文件 main.c 里的 main()函数。本章中除非特别说明，你可以浏览 DOS 版或者 EV-LM3S811 版的代码，因为 2 个版本里的应用程序代码是一致的。完整的 main.c 文件请看列表 1.1。

注：为了解释代码，我在感兴趣的代码左边写了在括号里的数字。在后面有对这些标记的解释。有时，为了区分特定段里和解释段里的列表，我使用了完整的包含列表序号和标签的引用。例如：列表 1.1（21）引用列表 1.1 的标签（21）。

代码列表 1 飞行射击游戏的 main.c 文件

```
 (1) #include "qp_port.h"                                    /* the QP port */
 (2) #include "bsp.h"                                  /* Board Support Package */
 (3) #include "game.h"                                      /* this application */

    /* Local-scope objects -----------------------------------------------------*/
 (4) static QEvent const * l_missileQueueSto[2];               /* event queue */
 (5) static QEvent const * l_shipQueueSto[3];                 /* event queue */
 (6) static QEvent const * l_tunnelQueueSto[GAME_MINES_MAX + 5];  /* event queue */
 (7) static ObjectPosEvt   l_smlPoolSto[GAME_MINES_MAX + 8];  /* small-size pool */
 (8) static ObjectImageEvt l_medPoolSto[GAME_MINES_MAX + 8]; /* medium-size pool */
 (9) static QSubscrList    l_subscrSto[MAX_PUB_SIG];       /* publish-subscribe */

    /*.....................................................................*/
    void main(int argc, char *argv[]) {
                          /* explicitly invoke the active objects' ctors... */
(10)    Missile_ctor();
(11)    Ship_ctor();
(12)    Tunnel_ctor();

(13)    BSP_init(argc, argv);         /* initialize the Board Support Package */
(14)    QF_init();    /* initialize the framework and the underlying RT kernel */

                                       /* initialize the event pools... */
(15)    QF_poolInit(l_smlPoolSto, sizeof(l_smlPoolSto), sizeof(l_smlPoolSto[0]));
(16)    QF_poolInit(l_medPoolSto, sizeof(l_medPoolSto), sizeof(l_medPoolSto[0]));

(17)    QF_psInit(l_subscrSto, Q_DIM(l_subscrSto));  /* init publish-subscribe */

                                         /* start the active objects... */
(18)    QActive_start(AO_Missile,/* global pointer to the Missile active object */
                      1,                              /* priority (lowest) */
                      l_missileQueueSto, Q_DIM(l_missileQueueSto), /* evt queue */
                      (void *)0, 0,                    /* no per-thread stack */
                      (QEvent *)0);                /* no initialization event */
(19)    QActive_start(AO_Ship,     /* global pointer to the Ship active object */
                      2,                                    /* priority */
```

```
                    l_shipQueueSto,    Q_DIM(l_shipQueueSto),    /* evt queue */
                    (void *)0, 0,                    /* no per-thread stack */
                    (QEvent *)0);                /* no initialization event */
(20)    QActive_start(AO_Tunnel,  /* global pointer to the Tunnel active object */
                    3,                                      /* priority */
                    l_tunnelQueueSto,  Q_DIM(l_tunnelQueueSto),  /* evt queue */
                    (void *)0, 0,                    /* no per-thread stack */
                    (QEvent *)0);                /* no initialization event */

(21)    QF_run();                                  /* run the QF application */
   }
```

(1) 飞行射击游戏是一个使用 QP 事件驱动平台实现的实例.每个使用 QP 的 C 文件必须包含 qp_port.h 头文件.这个头文件包括了 QP 对指定处理器操作系统和编译器的特别修改,也就是移植.每个 QP 移植位于单独的目录下,编译器通过提供给它的搜索路径找到相应的 qp_port.ht 头文件(典型的是通过-I 编译开关).这样我不需要为不同的处理器或编译器修改源代码.我只需告诉编译器在不同的目录下寻找 qp_port.h 头文件.例如.DOS 版从<qp>\qpc\ports\80x86\dos\tcpp101\l\引入 qp_port.h 头文件,EV-LM3S811 版从<qp>\qpc\ports\cortex-m3\vanilla\iar\引入头文件。

(2) bsp.h 头文件包含了对板支持包的接口，位于应用程序目录下。

(3) game.h 头文件包含了对事件和其他程序的部件共享工具的声明。我将在 1.6 节讨论这个头文件。它位于应用程序目录下。

QP 事件驱动平台是一些部件的集合，如根据 UML 语法执行状态机的 QEP 事件处理器和实现主动对象计算模型的 QF 实时框架。在 QF 的主动对象是封装后的状态机（每个都一个事件队列，一个单独的任务上下文和自己的优先级）它们通过发送和接收事件来异步通讯, QF 处理全部线程安全的事件交换和排队的细节。在一个主动对象内部，事件通过 QEP 事件处理器被依次通过运行-到-完成（RTC）方式处理，意味着在处理下一个事件前当前被处理的事件必须被处理完。

(4-6) 应用程序必须为所有它使用的主动对象的事件队列提供存储空间。这里存储是在编译时通过分配不变的（contst）指向事件的指针数组实现的，因为 QF 事件按队列只拥有指向事件的指针而不是事件本身。事件一些是在 qp_port.h 头文件里声明的 QEvent 结构的实例。每个主动对象能有不同的大小的的事件队列,你可以根据你对应用程序的情况来决定这个大小。我将在第 6 和第 7 章讨论事件队列。

(7-8) 应用程序必须对框架使用的事件池的快速和可决定的事件的动态分配提供存取空间。每个事件池可以提供固定大小的内存块。为了避免为小的事件浪费内存，QF 框架可以管理 3 个不同大小块的事件池（小，中等的和打的事件）。飞行射击应用程序只使用其中 2 种（小的和中等的池）。

QF 框架支持 2 中事件派送机制：简单的直接发送事件给主动对象，和更先进的以便把事件产生者和消费者分开的“发行-订阅“机制。在”发行-订阅“机制里，主动对象通过框架订阅事件。事件生产者发行事件给框架。对每一个发行请求，框架发送事件给所有已订购这个事件的主动对象。一个明显的”发行-订阅“的要求是框架必须存储订阅者的信息。但是它必须能处理对任何给定事件类型的多重订阅者。在第 6 章和第 7 章讨论事件派发机制。

(9) 飞行射击游戏使用 QF 提供的发行-订阅事件派送机制，因此它需要对订阅者列表提供存储空间。订阅者列表记住对每个事件订阅的主动对象。订阅者数据库的大小基于在 game.h 头文件里使用 MAX_PUB_SIG 常数定义发行事件的数量，和 QF 配置变量 QF_MAX_ACTIVE 定义的系统中允许的主动对象的最大数量。

(10-12) 这些函数执行对系统中主动对象的初步初始化。它们扮演静态“构建者”的角色，在 C 语言中你需要明确的调用。(C++在进入 main()函数前隐形的调用这些静态构建者)。

(13) 函数 BSP_init()在 bsp.c 文件中定义，对板初始化。

(14) 函数 QF_init()初始化 QF 组件和底层的 RTOS/内核，如果有使用它们的话。你需要在调用任何 QF 服务前执行 QF_init().

(15-16) 函数 QF_poolInit()初始化事件池。函数的参数是指向事件池存储空间的指针，存储空间的大小和池的块大小。你可以最多 3 次调用这个函数去初始化至多 3 个事件池。调用 QF_poolInit()的次序必须按块的大小递增的次序。例如，小事件池块必须在中事件池块之前被初始化。

(17) 函数 QF_poolInit()初始化 QF 的发行-订阅事件派发机制。函数的参数是指向订阅者列表数组的指针和这个数组的维数。

帮助宏 Q_DIM(a)通过 sizeof(a)/sizeof(a[0])计算出一维数组 a[]的大小, 这是个编译时常数。这个宏简化了代码因为它允许我不用去写很多#define 常数，否则我需要对不同的数组提供维数。我能在定义一个数组，这个唯一需要我提供维数的时候，硬编码数组的维数。然后，当我需要在代码中使用这个维度时用宏 Q_DIM()。

(18-20) 函数 QActive_start()告诉 QF 框架开始管理一个主动对象作为应用程序的一部分。这个函数使用如下参数：指向主动对象结构的指针，主动对象的优先级，指向它的事件队列的指针，队列的维度（长度），和三个我将在第 7 章解释的其他参数，因为它们和现在讨论的问题无关。QF 里的主动对象优先级是从 1 开始递增到并包括 QF_MAX_ACTIVE，较高的优先级数字表示主动对象有较高的急迫性。常数 QF_MAX_ACTIVE 在 QF 移植头文件 qf_port.h 中定义，当前版本中它不能超过 63。

我喜欢把每个主动对象的代码和数据严格的封装在它自己的 C 文件中。例如，主动对象 Ship 的全部代码和数据都放在文件 ship.c 里，和外部的界面是函数 Ship_ctor()和指针 AO_Ship.

(21) 从这里开始，你提供了框架它管理你的应用程序所需的全部存储空间和信息。最后你必须要做的就是调用函数 QF_run()把控制权交给框架。

在调用 QF_run()后，框架获得完全的控制权。框架通过调用你的代码而不是其他方法来运行应用程序。在 DOS 版本的飞行射击游戏里，你可以按 ESC 按键来结束程序，这里 QF_run()退回到 DOS 而不是 main().在嵌入式系统里，比如在 stellaris 演示版里，QF_run()一直运行或者直到电源关闭为止。

注：为了最好的平台间可移植性，源代码使用了 UNIX 风格的行尾结束规范（行只使用 LF 结束, 0xA 字符）。这个规范看来对全部 C/C++编译器和交叉编译器,包括古老的 DOS 时代的工具.相反, DOS/WINDWOS 结尾规范(行使用 CR LF 结束。0xD, 0xA 字符对), 已知会在 UNIX 类的平台特别是在多行预处理器宏上造成问题,.

1.4 “飞行射击“游戏的设计

为了更深入的解释飞行射击应用程序，我需要进入到设计层面。这里我需要解释应用程序是如何被分解为各个主动对象，以及它们如何交换事件从而实现飞行射击游戏的功能。

一般的，把问题分解成主动对象并不简单。通常在任何分解中。你的目的是达成在各个主动对象间尽可能松散的耦合（理想情况是它们之间没有任何共享的资源），而且你需要尽力减少它们之间通讯的频度和需要交换的事件的大小。

在飞行射击游戏实例里，我需要首先确认所有拥有反应行为的对象（比如，拥有一个状态机）。我应用最简单的确认对象的面向对象技术,也就是挑选问题规范里被频繁使用的名词。从 1.2 节里，我去人了 Ship, Missile, Mines 和 Tunnel。然而，不是系统中的每一个状态机都需要是一个主动对象（并拥有单独的任务上下文，一个事件队列和唯一的优先级），并且，当需要性能或空间上需要时，融合它们是一个有效的选择。作为这个方法的一个举例，我把 Mines 融合到 Tunnel 主动对象中，从而我把 Mines 作为主动对象 Tunnel 里面独立的状态机组件。这样我使用了在第 5 章里描叙的“正交组件”设计模式。

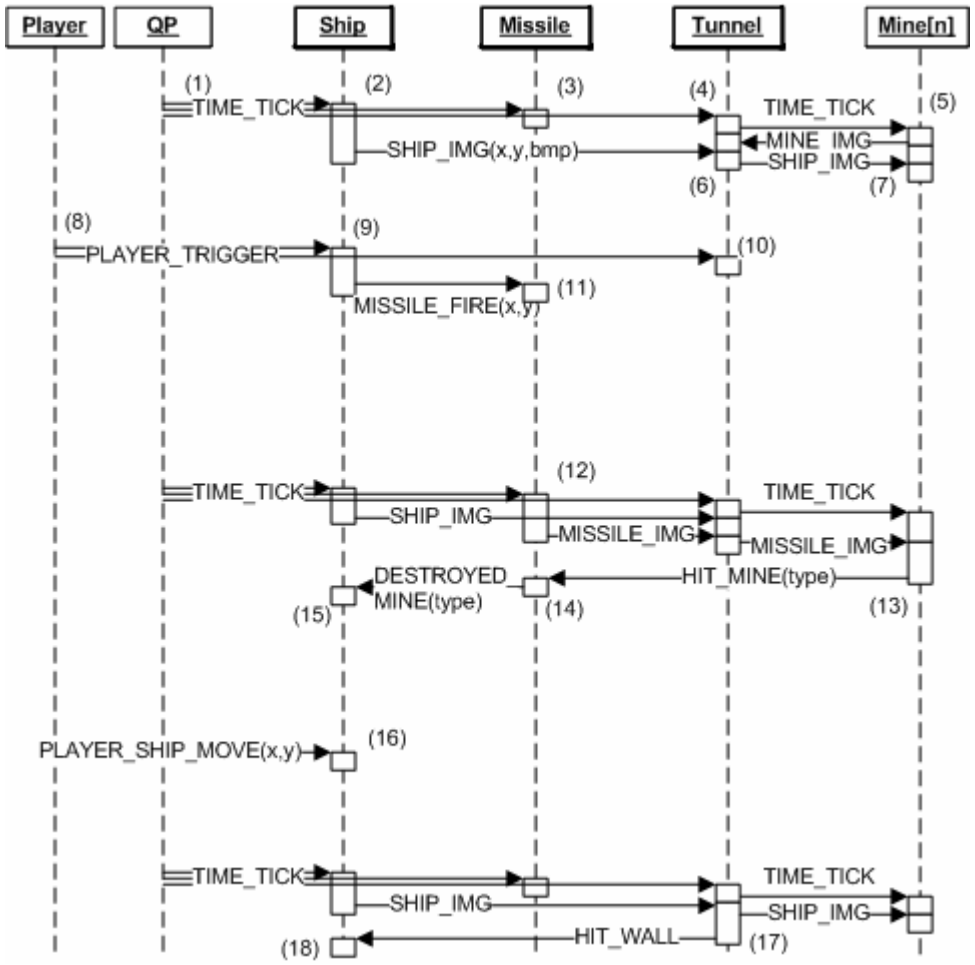
事件驱动应用程序射击的下一步是给每一个被确认的主动对象分配责任和资源。避免共享资源的通用的设计策略是吧每一个资源封装在一个指定的主动对象里面，并让这个主动对象为应用程序的其他部分管理资这个资源。这样，程序通过事件来共享指定的主动对象而不是直接共享资源。

所以，作为例子，我决定让主动对象 Tunnel 来管理显示器。其他的主动对象和状态机组件，比如 Ship, Missile 和 Mines，不直接在屏幕上绘图，而是给对象 Tunnel 发送事件，请求在显示器指定的坐标点(x,y)描绘 Ship,Missile 或者 Mine 的位图。

在理解了对主动对象的责任和分配的资源后，我可以进一步规划对象之间不同的事件交换场景。也许这个阶段最好的思考过程的指导是在图 1.4 里描叙的 UML 顺序图.这份顺序图展示了在飞行射击游戏（主要的用例，如果你愿意）里最常见的事件交换场景。在顺序图后免得解释部分阐明了一些有趣的地方。

注：类似图 1.4 的 UML 顺序图有 2 个维度。水平方向排列的方格表示这个场景里不同的对象，黑实线的方格表示是主动对象。通常在 UML 里，对象名使用下划线。从方格往下的垂直点划线表明时间的方向。事件用水平箭头表示，从发送对象开始，到接收对象结束。围绕实例线的可选的细框表示控制的焦点。

图 4 飞行射击游戏的顺序图



注：为了解释图例，我在图中感兴趣的地方放入带括号的数字记号。然后我使用这些标记在随后解释部分引用。为了在特定的文字段引用特定图的指定元素，我使用全引用。比如，图 1.4(12)表示应用在图 1.4 里的元素（12）。

(1) TIME_TICK 是游戏中最重要的事件。这个事件由 QF 框架从系统事件节拍中断生成, 速率是每秒 30 次从而可以流畅的驱动显示器上的动画。因为 TIME_TICK 事件对应用程序中全部对象都有用，它被框架发行给所有的主动对象。(QF 里的发行-订阅事件派送在第 6 章描叙.)

(2) 在收到 TIME_TICK 事件后，对象 Ship 朝前一步移动它的位置，并发送事件 SHIP_IMG(x, y, bmp) 给对象 Tunnel。事件 SHIP_IMG 有参数 x 和 y, 是 Ship 在显示器上的坐标，还有需要在这个坐标画出的位图序号 bmp。

(3) 对象 Missile 还没有飞行，因此这个时刻它简单的忽略事件 TIME_TICK.

(4) 对象 Tunnel 执行对应事件 TIME_TICK 最繁重的工作。首先，Tunnel 根据当前的帧缓冲区重新画全部显示器。这个动作每秒执行 30 次从而提供显示动画的效果。然后，Tunnel 清空帧缓冲区，并开始为下帧显示时刻而填充帧缓冲区。Tunnel 向前一步移动隧道墙壁，然后把墙壁复制到帧缓冲区。Tunnel 也派送事件 TIME_TICK 给所有它的 Mine 状态机组件。

(5) 每个 Mine 向前移动一步它的位置并发行事件 MINE_IMG(x, y, bmp)给 Tunnel 在当前的帧缓存中的位置(x,y)画出合适的地雷位图。一型地雷送出位图序号 MINE1_BMP,二型地雷送出位图序号 MINE2_BMP。

(6) 对象 Tunnel 在从 Ship 收到事件 SHIP_IMG(x, y, bmp)后，它在帧缓存里画出飞船的位图并检查飞船位图和隧道墙壁是否有碰撞。Tunnel 再发布原事件 SHIP_IMG(x, y, bmp)给全部活动的 Mine.

(7) 每个 Mine 确定 Ship 是否和它有碰撞。

(8) 当用户可靠的按下按钮（按钮已被消抖）时产生事件 PLAYER_TRIGGER。这个事件被 QF 框架发行并派送给对象 Ship 和对象 Tunnel，这 2 个对象都订阅了事件 PLAYER_TRIGGER.

(9) 对象 Ship 产生事件 MISSILE_FIRE(x, y)给对象 Missile.这个事件的参数是 Ship 当前的座标(x, y), 它也是 Missile 的出发点座标。

(10) Tunnel 接收被发行的 PLAYER_TRIGGER 事件因为根据情况这个输入它需要启动游戏或者结束屏保模式。

(11) Missile 对事件 MISSILE_FIRE(x, y)的反应是开始飞行，它根据 Ship 发行给它的事件参数(x, y)设置自己的初始位置。

(12) 从这时开始，当 Missile 在飞行时，事件 TIME_TICK 到达。Missile 发送事件 MISSILE_IMG(x, y, bmp)给 Table.

(13) Table 在当前帧缓存里画出 Missile 的位图，并派送事件 MISSILE_IMG(x, y, bmp)给所有的 Mine，让它们测试自己和 Missile 有否碰撞。这中确定和 Mine 的类型有关。在这个场景里，一个特定的对象 Mine[n]检测到一次击中，并发行事件 HIT_MINE(score)给 Missile. Mine 把击毁这个特定类型鱼雷获得的分数作为这个事件的参数。

(14) Missile 处理事件 HIT_MINE(score),它立刻可以预备发射，并且它让 Mine 执行爆炸。因为我决定让 Ship 负责维护分数，Missile 也产生事件 DESTROYED_MINE(score)给 Ship,报告击毁 Mine 获得的分数。

(15) 在收到事件 DESTROYED_MINE(score)后，Ship 根据 Missile 的报告更新分数。

(16) 对象 Ship 处理事件 PLAYER_SHIP_MOVE(x, y), 根据事件的参数更新它自己的位置。

(17) 当对象 Tunnel 处理事件 SHIP_IMG(x, y, bmp_id)时，它检测到在 Ship 和隧道墙壁之间有碰撞。这种情况下，它发送事件 HIT_WALL 给 Ship。

(18) Ship 对事件 HIT_WALL 反应，它转换到“爆炸”状态。

尽管图 1.4 的顺序图展示了飞行设计游戏里很少的几个场景，我希望这个解说能给你一个程序是如何工作的总体印象。更重要的是，你应该开始掌握一些一般的如何使用主动对象和事件来设计一个事件驱动系统的思考方法。

1.5 “飞行和射击”游戏中的主动对象

我希望对图 1.4 顺序图的分析能清楚的说明，一个主动对象执行的行为依靠于它接收到的事件和它内部的状态。例如，主动对象 Missile 处理事件 TIME_TICK 的方式，当它在飞行时(图 1.4(12)和它没有飞行时（图 1.4(3)，是完全不同的。

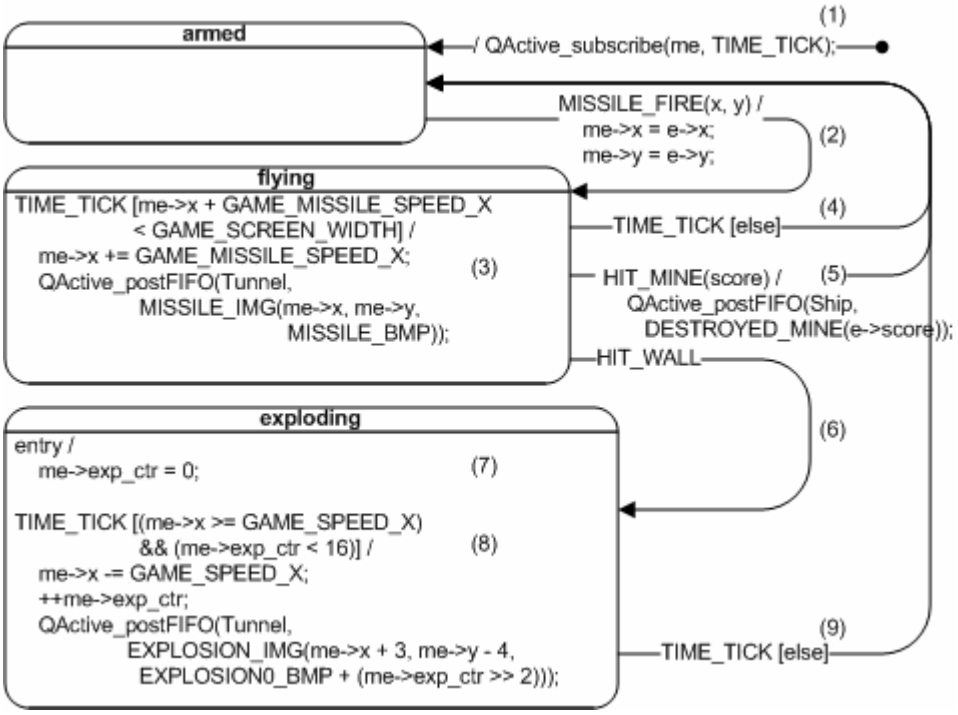
已知的最佳的处理这类模式行为的机制是通过状态机，因为状态机使表现完全的依靠于事件和对象的状态。在第 2 章，我会更加具体的介绍 UML 状态机概念。本节里，我对飞行射击游戏里和每个对象结合的状态机做个简要的解说。

1.5.1 Missile 主动对象

我从图 1.5 的 Missile 状态机开始，因为它是最简单的。图片后面的解说部分展示了有趣的要点。

注：类似图 1.5 的 UML 状态图保留传统状态转移图的一般外形，状态用节点表示，转移用弧线连接在节点间。在 UML 里，状态节点用圆角方框表示。状态名用粗体字在状态框顶部的名字格里。可选的，在名字下面，一个状态可以有内部转移格，用一条水平线和名字格分开。内部转移格包含入口动作（在保留符号 entry 后的动作）退出动作（在保留符号 exit 后的动作）和其他内部转移（比如在图 1.5(3)里被 TIME_TICK 触发的那些动作）。状态转移用从源状态边界指向目的状态边界的箭头表示。最低要求，一个转移必须用触发事件来标注。触发可以有可选的事件参数，一个警戒条件和一个动作列表。

图 5 Missile 状态机图



(1) 从实心黑点的状态转移被称为初始转移。这种转移表明状态机被建立后的第一个活动状态。初始转移可以有相应的动作，根据 UML 标注法，在/后列出。在本例中，Missile 状态机开始于 ARM 状态，相应的动作是订阅事件 TIME_TICK。订阅一个事件意味着每次当这个事件被发行给框架时，框架把它发行给 Missile 主动对象。

(2) 标注了事件 MISSILE_FIRE(x, y)的箭头说明一个状态转移，也就是说，从 armed 状态到 flying 状态。当用户触发 Missile(参考图 1.4 的顺序图)时，对象 Ship 产生事件 MISSILE_FIRE(x, y)。在 MISSILE_FIRE 事件里，Ship 通过参数(x,y)提供给 Missile 初始的座标。

注：UML 没有特意规定动作的标注。实际应用中，动作常常用要对这个状态机进行编程的语言书写。我将用 C 语言编写本书中所有的状态机。而且，在用 C 表达时，我通过前缀“me->”来应引用和状态机对象有关的数据成员，用前缀”e->”来表示事件变量。例如，动作“me->x = e->x;”表示用事件变量 x 给主动对象 Missile 的内部数据成员 x 赋值。

(3) 在状态名下方的事件名 TIME_TICK 标识一个内部转移。内部转移是没有状态改变的简单的事件反应。内部转移，和正常的转移一样，可以拥有一个守卫条件，用方括号围住。守卫条件是一个在运行时计算的布尔量。如果守卫条件为真，转移就发生。否则，转移不会发生，在/后面的动作也不会执行。在我们的例子里，守卫条件检查 Missile 的速度影响下它的 x 座标是否还在显示器的可视范围内。如果在，动作就执行。这些动作包括步进增加 Missile 位置并发送 MISSILE_IMG 事件，使用当前 Missile 的位置和 MISSILE_BMP 位图序号作为参数，给主动对象 Tunnel。直接发送事件给一个主动对象是通过 QF 函数 QActive_postFIFO()完成的，这个函数将在 7 章讨论。

(4) 相同的事件 TIME_TICK,带有[else]守卫条件，表明一个规范的状态转移，它的守卫条件是和其他在相同状态并发的 TIME_TICK 事件的补充。这个例子里，TIME_TICK 转移到 armed 状态只有在对象 Missile 飞出显示器外面才发生。

(5) 事件 HIT_MINE(score)触发了另一个到 armed 状态的转移。和这个转移相连的动作发送事件 DESTROYED_MIND,参数是 e->score，报告摧毁地雷给对象 Ship。

(6) 事件 HIT_WALL 触发了一个到 exploding 状态的转移，目的是在显示器上显示爆炸位图。

(7) 标签 entry 表明在进入 exploding 状态时入口动作必须无条件执行。这个动作是清除对象 Missile 的爆炸计数器(me->exp_ctr)成员。

(8) TIME_TICK 内部转移的守卫条件是爆炸不会移出到屏幕外面，而且爆炸计算器低于 16.相应的动作是产生爆炸的位置，并发送事件 EXPLOSION_IMG 给主动对象 Tunnel。请注意当爆炸计数器增大时，爆炸位图相应的改变。

(9) 带有守卫条件的 TIME_TICK 规则转移使状态回到 armed 状态。这个转移在爆炸动画完成后发生。

1.5.2 Ship 主动对象

主动对象 Ship 的状态机如图 1.6 所示。这个状态机引入了层次状态机的概念。状态嵌套的威力在于它被设计于用来排除可能会发生的重复。

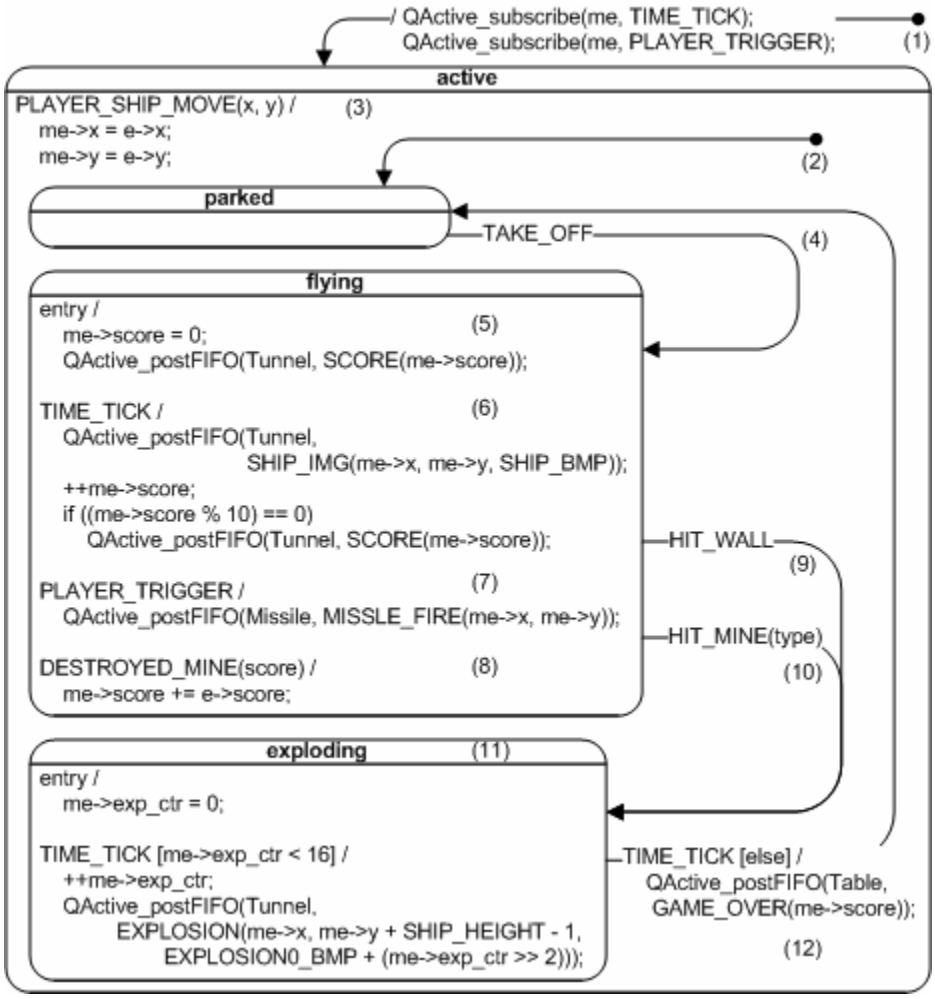
主动对象 Ship 的一个主要责任是维护 Ship 的当前位置数据。在原 EV-LM3S811 板，这个位置用电位器转轮确定（图 1.2）。只要转轮的位置改变，就会产生事件 PLAYER_SHIP_MOVE(x, y)。对象 Ship 必须一直跟踪轮的位置，意味着 Ship 状态机的所有状态都必须处理 PLAYER_SHIP_MOVE(x,y)。

在传统的有限状态机 FSM 方法里，你需要在 Ship 的每个状态根据事件 PLAYER_SHIP_MOVE(x,y)来重复更新它的位置。这种重复使状态机变得很庞大，而且更重要的是，在图和代码里需要同时维护的地方也相应的增多。这种重复和 DRY(Don’t Repeat Yourself, 不要重复你自己)原理相抵触，这个原理对灵活的和可维护的代码是很非常重要的。

层次状态嵌套解决了这个问题。考虑在图 1.6 里包含所有其他状态的 active 状态。高层的 active 状态被称为超级状态，抽象定义是状态机不能直接在这个状态里，只能在它里面嵌套的某一个状态里。UML 和状态嵌套有关的语法说明任何事件先被当前正在活动的子状态处理。如果这个子状态不能处理这个事件，状态机尝试在较高一层超级状态的上下文处理这个事件。当然，UML 里的状态嵌套不限于仅一层，而且，这个处理事件的简单规则可以递归的应用于任何多层的嵌套。

对图 1.6 展示的 Ship 状态机来说，假设当状态机在 parked 状态时刻事件 PLAYER_SHIP_MOVE(x, y)到达。Parked 状态不处理事件 PLAYER_SHIP_MOVE(x, y). 在传统的有限状态机里这可能就这样结束了—事件 PLAYER_SHIP_MOVE(x, y);被默默的丢弃。然而，图 1.6 里的状态机有另外一层 active 超状态。根据状态嵌套的语法，较高层的超状态会处理事件 PLAYER_SHIP_MOVE(x, y),这正是我们需要的。完全同样方法也应用于超状态 active 的所有子状态，如 flying 或 exploding，因为这些子状态都不处理事件 PLAYER_SHIP_MOVE(x, y). 相反，超状态 active 在同一个地方而不用重复的处理这个事件。

图 6 Ship 状态机图



(1) 在初始转移时，状态机 Ship 进入到超状态 active 并订阅事件 TIME_TICK 和 PLAYER_TRIGGER.

(2) 在嵌套状态的每一层的超状态都可以有一个私有的初始转移，用来指定直接进入这个超状态后的活动子状态。这里，active 状态的初始转移指定子状态 parked 作为初始活动子状态。

(3) 超状态 active 把处理事件 PLAYER_SHIP_MOVE(x, y)作为一个内部转移，这里它根据事件参数 e->x 和 e->y 相应的更新内部数据成员 me->x 和 me->y.

(4) 事件 TAKE_OFF 触发到 flying 的转移。当玩家开始游戏时（见 1.2 节对游戏的描叙），对象 Tunnel 产生这个事件。

(5) flying 的入口动作包括清除数据成员 me->score，并发送事件 SCORE, 用 me->score 作参数，给主动对象 Tunnel.

(6) TIME_TICK 内部转移会发送事件 SHIP_IMG，使用当前 Ship 的位置和 SHIP_BMP 位图序号作参数，给主动对象 Tunnel。另外，因为在这个事件片继续存活，分数也相应的被增加。最后，当分数要‘进位’（可被 10 整除）时，z 这个消息也被发送给主动对象 Tunnel。这样发送事件 SCORE 的目的是减少通讯的所需的带宽,因为主动对象 Tunnel 只需要在游戏时告诉用户一个近似的分数。

(7) 内部转移 PLAYER_TRIGGER 会发送参数是当前 Ship 位置的事件 MISSILE_FIRE 给主动对象。参数(me->x, me->y)从 Ship 给 Missile 提供初始的位置值。

(8) 内部转移 DESTORYED_MINE(score)更新由 Ship 维护的分数。在这一刻，分数没有被发送给 Table，因为下一个 TIME_TICK 将会送出“凑整”的分数，已经足够给玩家近似的分数了。

(9) 事件 HIT_WALL 触发到 exploding 的转移。

(10) 事件 HIT_MINE(type)也触发到 exploding 的转移。

(11) Ship 状态机的 exploding 状态和 Missile 的 exploding 状态十分相似。（见图 1.5（7-9））。

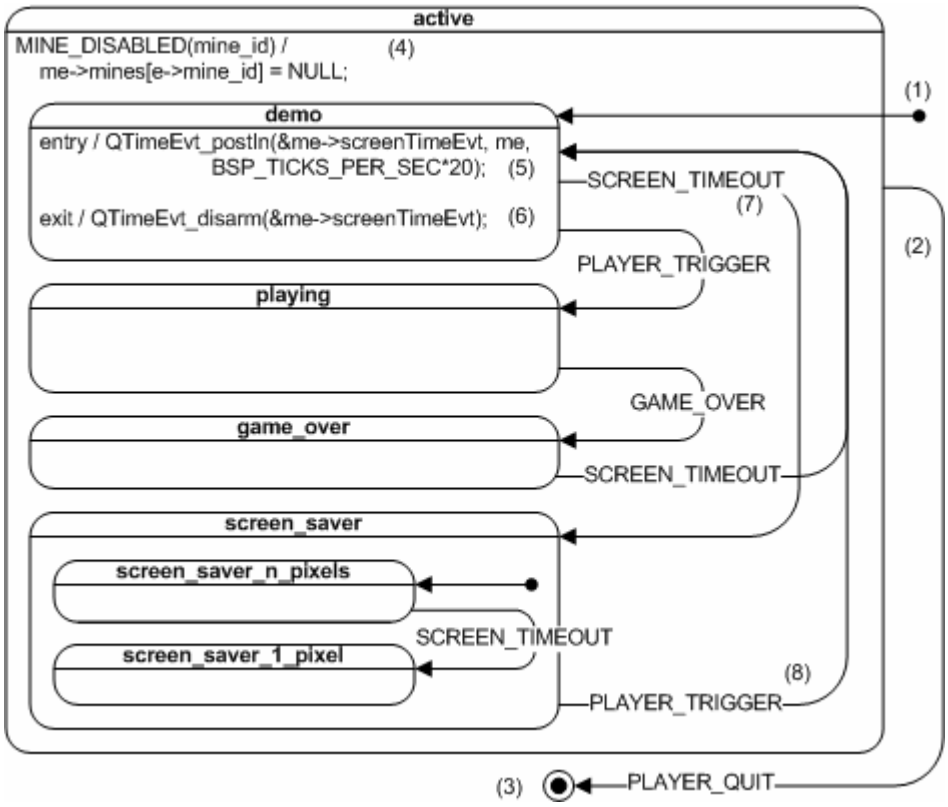
(12) 转移 TIME_TICK[else]在 Ship 完成爆炸后发生。在这个转移里，对象 Ship 发送事件 GAME_OVER(me->score)给主动对象 Tunnel,结束游戏，并显示最后得分给玩家。

1.5.3 Tunnel 主动对象

如图 1.7 所示，主动对象 Tunnel 拥有最复杂的状态机。和前面的状态图不同，图 1.7 仅显示了高层的状态并忽略了许多细节，如绝大多数入口/出口动作，内部转移，警卫条件，或在转移是的动作。这类“缩小”视图在 UML 是合法的。因为 UML 允许你选择你想包含在图里的细节的层次。

Tunnel 状态机比图 1.6 的 Ship 状态机更加彻底的使用了层次式状态。在图 1.7 后的解说说明了状态嵌套的新的用法和在其他状态图里还没解释过的新的元素。

图 7 Tunnel 状态机图



(1) 一个初始转移能到达层次式状态的任何层的子状态，而不仅仅是次一级。这里，最顶层的初始转移向下 2 层到达子状态 demo.

(2) 超状态 active 处理事件 PLAYER_QUIT，转移到最终状态（见元素（3）解释）。请注意 PLAYE_QUIT 转移适用于超状态 active 的所有直接或间接嵌套的子状态。因为一个状态转移总是包含状态的所有退出动作，高层 PLAYER_QUIT 转移保证对当前状态上下文饿正确的清理，而不轮在 PLAYER_QUIT 事件到达是哪个子状态是活动态。

(3) 最终状态在 UML 记号里是牛眼符号，典型的用法是指示状态机对象的解构。这里，PLAYER_QUIT 事件表明了游戏的结束。

(4) 事件 MINE_DISABLED(mine_id)在状态 active 高层被处理，意味着这个内部转移适用嵌套在超状态 active 的所有子状态机。（请看下一节的对象 Mine 讨论）。

(5) 到状态 demo 的入口动作在 20 秒内启动屏幕 time 事件（计时器）me->screenTimerEvt。时间事件有应用程序分配，但是有 QF 框架管理。QF 提供函数去装备一个时间事情，如为一次性超时的 QTimeEvt-postIn()，和周期性时间事件的 QTimeEvt_postEvery(). 装备一个时间事件有效的告诉 QF 框架，例如“20 秒内提醒我”。QF 然后在请求的时钟时刻后发送时间事件(本例里是 me->screenTimeEvt)给主动对象。第 6 章和第 7 章谈论时间事件的细节。

(6) 从 demo 状态的退出动作取消 me->screenTimeEvt 事件事件。当一个状态可以由和时间事件不同的另外的事件，比如 PLAYER_TRIGGER 转移，退出时，这个清理是必须的。

(7) 到 screen saverSCREEN_TIMEOUT 转移由 me->screenTimerEvt 时间事件的到期而被触发。信号 SCREEN_TIMEROUT 在初始化时被赋予这个时间事件并不能再被修改。

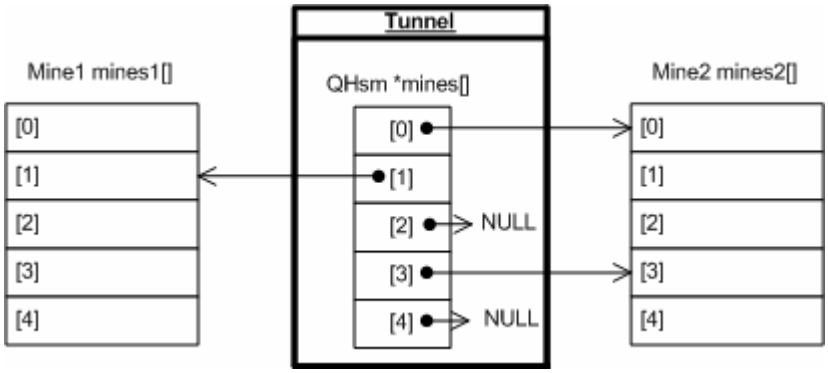
(8) 由 PLAYER_TRIGGER 触发的转移一致的适用与超状态的 2 个子状态。

1.5.4 Mine 组件

Mine 也被建模成层次式状态机，但是不是主动对象。相反，地雷是主动对象 Tunnel 的组件，共享它的事件队列和优先级。主动对象 Tunnel 和 Mine 的同步通讯是通过函数 QHsm_dispatch()给它们派送事件。Mine 和 Tunnel 还有其他所有主动对象的异步通讯是通过函数 QActive_postFIFO()发送事件给它们的事件队列。

注：主动对象异步交换事件，意味着事件的发送者仅发送事件给接收者的事件队列而不用等待事件处理过程结束。同步事件处理对应着一个函数调用（如 QHsm_dispatch()）,它在调用者的执行线程里处理事件。

图 8 Tunnel 主动对象管理 2 类地雷



如图 1.8 所示，Tunnel 维护数据成员 mines[],这是一个指向层次式状态机(QHsm*)的指针数组。数组的每一个指针能指向一个 Mine1 对象或者一个 Mine2 对象，如果单元没有使用，它是空指针。请注意，Tunnel 仅‘知道’Mine 是一个通用的状态机(指向在 QP 定义的 QHsm 结构)。Tunnel 一致的派送事件给 Mine,而不去区分 Mine 的类型的不同。还有，每个 Mine 状态机用它自己的方式来处理这些事件。例如，Mine 类型 2 检查和 Missile 碰撞与和 Ship 的碰撞不同，Mine 类型 1 在处理这 2 类碰撞时是一样的。

注：最后一点很有趣。发送同样的事件给不同的 Mine 对象造成不同的表现，比如相对于不同 Mine 的类型反应不同，在 OOP 里这叫多态性。我将在第 3 章里讨论这点。

每个 Mine 对象都是自治的。Mine 维护它自己的位置并负责通知 Tunnel 对象它自己何时爆炸或何时移出显示画面。这个信息对对象 Tunnel 是非常有用的，它能以此跟踪未用的地雷。

图 1.9 显示了 Mine2 状态机的层次状态图。Mine1 和它很相似，除了它使用同样的位图来测试和 Missile 和 Ship 的碰撞

1.6 “飞行和射击”游戏中的事件

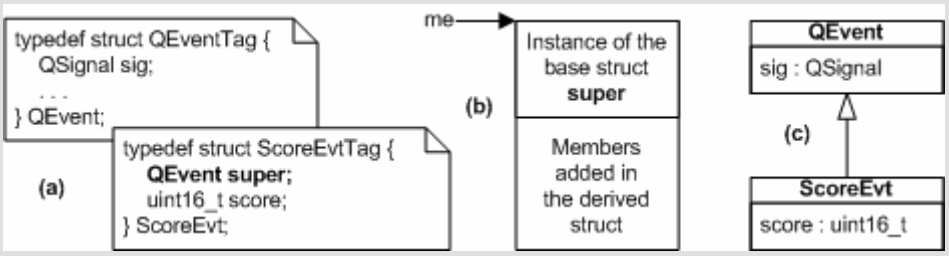
飞行射击中关键的事件已在序列图 1.4 中标示出来.其他的事件在状态机的设计阶段确定。任何情况下，你应该注意到事件实际包含 2 个部分。事件里被称为信号的部分表达已发生事件的类型（什么发生了）。例如，TIME_TICK 信号表达一个时刻的到来，而 PLAYER_SHIP_MOVE 信号表达玩家想移动飞船。一个事件也能通过事件参数的形式来包含已发生事件的量化信息。例如，PLAYER_SHIP_MOVE 信号携带这变量(x, y)，包含着把 Ship 移动到何处的量化信息。

在 QP 里，事件被表示为使用框架提供的 QEvent 结构的实例。具体来说，QEvent 结构包含成员 sig, 表示事件的信号。事件参数通过继承过程来添加。如下文“C 语言里的单继承”所描叙。

C 语言里的单继承

继承是基于已有结构派生新的结构，从而复用和组织代码的一种能力。你可以非常简单的在 C 里实现单继承，只要字面上把基结构作为派生结构的第一个成员即可。例如，图 1.10（a）显示了通过把 QEvent 实例作为 ScoreEvt 的第一个成员嵌入，从而从基本结构 QEvent 派生出结构 ScoreEvt。为了使这个方法更加突出，我总是把基本结构成员命名为 super.

图表 1 (a) C 里的结构派生，(b)内存安排 (c)UML 类图



如图 1.10(b)所示，这种结构的嵌套总是把数据成员 super 放在派生结构的每一个实例的开始处，这是被 C 标准保证的。具体来说，WG14/N1124 的 6.7.2.1.13 小节说：“…指向一个结构对象的指针，在适当的转换后，指向它的最初的成员。在一个结构对象中可能有未命名的填充部分，但不是放在开始处” [ISO/IEC 9899:TC2]. 这种内存安排使你可以把一个指向派生的 ScoreEvt 结构的指针当作一个指向 QEvent 基本结构的指针一样处理. 这些都是合法,可移植的，而且是 C 标准所保证的.因此，你总是可以安全的传递一个指向 ScoreEvt 的指针给任何需要一个指向 QEvent 的指针的函数。（严格来讲，在 C 里你应该显式的转换这个指针。在 OOP 里，这种转换被陈称为向上转换，而且它总是安全的。）结果，所以为 QEvent 结构设计的函数自动的适用于 ScoreEvt 结构和其他从 QEvent 派生的结构。图 1.10(c)的 UML 类图, 说明了 ScoreEvt 结构和 QEvent 结构的继承关系。

QP 广泛的使用单继承，不仅用来派生带有变量的事件，也用来派生状态机和主动对象。当然，QP 的 C++版本使用固有的 C++类派生而不是“结构的派生”。你将在本章后面和全书看到更多的继承实例。

因为事件是在大多数应用组件间共享的，在如列表 1.2 所示的一个单独的头文件声明它们会带来便利。紧跟在列表后的解释部分揭示了一些有趣的要点。

代码列表 2 在文件 game.h 里定义的信号，事件结构和主动对象接口

```
(1) enum GameSignals {                          /* signals used in the game */
(2)     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 ... */
```

```
(3)    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 */
       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 ... */
(4)    MAX_SIG                       /* the last signal (keep always last) */
   };

(5) typedef struct ObjectPosEvtTag {
(6)    QEvent super;                          /* extend the QEvent class */
(7)    uint8_t x;                        /* the x-position of the object */
(8)    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 */
 (9) extern QActive * const AO_Tunnel;
(10) extern QActive * const AO_Ship;
(11) extern QActive * const AO_Missile;

    /* active objects' "constructors" */
(12) void Tunnel_ctor(void);
(13) void Ship_ctor(void);
(14) void Missile_ctor(void);
```

(1) 在 QP 里，事件的信号是简单的枚举常数。把所有的信号放在一个枚举里特别便于避免无意间重叠不同信号之间的数值。

(2) 应用程序层信号不是从零开始，而是有常数为 Q_USER_SIG 的偏移。这是因为 QP 保留了最低的几个信号给内部使用，并提供常数 Q_USER_SIG 作为偏移值，用户信号从它开始枚举。另外请注意，按照惯例，我在所有的信号后加上后缀_SIG，这样我可以很容易的把信号和其他常量分开。在状态图里我去掉了后缀_SIG 以减少杂乱。

(3) 常数 MAX_PUB_SIG 限定了需要发行的信号。发行-订阅事件递送机制消耗一些 RAM，它的大小和被发行信号数量成比例。我通过提供被发行信号的较小的数值给 QP(MAX_PUB_SIG)，而不是程序里被使用的全部信号的最大值。

(4) 最后一个枚举 MAX_SIG 标示程序里被使用的全部信号的最大值。

(5) 事件结构 ObjectPosEvt 定义了一个事件的’类‘，在事件的参数里携带着对象在显示屏的位置。

(6) 结构 ObjectPosEvt 从基本结构 QEvent 派生，解释请参考侧栏“C 语言里的单继承“。

(7-8) 结构 ObjectPosEvt 增加了参数 x 和 y,它们是对象在显示屏上的座标。

注：在全书里我使用以下标准的整数类型(WG14/N843 C99 标准，7.18.1.1 节)[ISO/IEC 9899:TC2]:

exact size	**unsigned**	**signed**
8-bits	uint8_t	int8_t
16-bits	uint16_t	int16_t
32-bits	uint32_t	int32_t

如果你的（非标准）编译器没有提供<stdint.h>头文件，你也能 typedef 标准的 C 语言数据类型如 signed/unsigned char, short, int 和 long，从而得到这些实际宽度整数类型。

(9-11) 这些全局指针代表在应用程序里的主动对象，并被用来直接发送事件给主动对象。因为这些指针可以在编译时被初始化，我喜欢把它们申明为 const，这样它们能被放在 ROM 里。这些主动对象指针是“不透明的”，因为他们不能存取整个主动对象，只能

1.6.1 事件的生成，发布和出版

QF 框架支持 2 种类型的事件交换。

1. 比较简单的机制，通过函数 QActive_postFIFO()和 QAcive_postLIFO()直接进行事件发布，一个事件的产生者直接把这个事件发布到作为这个事件消费者的主动对象的事件队列。
2. 更加灵活的发行-订阅事件传递机制，通过函数 QF_publish()和 QActive_subscribe()支持。事件的产生者把事件发行给框架，框架然后传递所有订阅了这些事件的主动对象。

在 QF 里，系统的任何部分都能够产生事件，而不仅局限于主动对象。例如，中断服务程序(ISR)或设备驱动程序也能产生事件。另一方面，只有主动对象能消费事件，因为只有主动对象才拥有事件队列。

注：QF 也提供了“原生”的线程安全事件对列(QEQueue 队列)。，它也能消费事件。这些“原生”的队列不能阻塞，而且它们被特意用来派送事件给 ISR 或者设备驱动程序。详细内容请参考第 7 章。

QF 的事件管理的最重要的特征是，框架只传递指向事件的指针，而不是事件本身。QF 从不复制事件（零复制策略），即使在发行事件从而常常需要把同样的事件发送到多个订阅者时也是如此。实际的事件实例是在编译时静态分配的固定事件，或者是在运行时从一个框架管理的事件池里分配的动态事件。代码列表 1.3 提供了在 Stellaris 版本(文件 <qp>\qpc\examples\cortex-m3\vanilla\iar\game-ev-lm3s811\bsp.c). 的“飞行射击”游戏里，发行静态事件和从 ISR 里发送动态事件的实例。在下一节 1.7.3 你将看到在状态及代码里从主动对象里发送事件的其他例子。

代码列表 3 Stellaris 版本的游戏代码从 ISR 中产生，发送和发行事件

```
(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 */
 (6)    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
        */
 (7)    adcLPS += (((int)tmp - (int)adcLPS + 4) >> 3);      /* Low-Pass-Filter */
```

```
        /* compute the next position of the wheel */
 (8)     tmp = (((1 << 10) - adcLPS)*(BSP_SCREEN_HEIGHT - 2)) >> 10;

        if (tmp != wheel) {                /* did the wheel position change? */
 (9)        ObjectPosEvt *ope = Q_NEW(ObjectPosEvt, PLAYER_SHIP_MOVE_SIG);
(10)        ope->x = (uint8_t)GAME_SHIP_X;             /* x-position is fixed */
(11)        ope->y = (uint8_t)tmp;
(12)        QActive_postFIFO(AO_ship, (QEvent *)ope);   /* post to the Ship AO */
           wheel = tmp;              /* save the last position of the wheel */
        }
        . . .
    }
```

(1) 在 Stellaris 版里，函数 ISR_SysTick()为 Cortex-M3 系统节拍计时器产生的系统时钟节拍 ISR 提供服务。

(2) TIME_TICK 事件从不改变，因此它可以被一次性静态分配。这个事件被声明为 const,意味着它可以被放在 ROM 区。这个事件的初始花列表包含信号 TIME_TICK_SIG，附带一个 0.这个 0 告诉 QF 框架这个事件是静态的并且不用被回收到某个事件池里。

(3) ISR 调用框架函数 QF_publish(), 这个函数把指向 tickEvt 事件的指针传递给所有订阅者。

(4) ISR 调用函数 QF_tick(), 在这个函数里框架管理有准备的时间事件。

(5) 函数 ISR_ADC()提供 ADC 转换服务。它最后派送 Ship 的位置。

(6) 这个 ISR 从 ADC 读出数据。

(7-8) 一个低通滤波器被用于读出的原始 ADC 数据，从而计算出电位器转盘的位置。

(9) QF 宏 Q_NEW(ObjectPostEvt, PLAYER_SHIP_MOVE_SIG)动态的从由 QF 框架管理的事件池内分配一个 ObjectPostEvt 事件的实例。这个宏也把信号 PLAYER_SHIP_MOVE_SIG 和分配的事件联合起来。宏 Q_NEW()返回一个指向这个刚分配事件的指针。

注：事件 PLAYER_SHIP_MOVE(x, y)是一个带有可变参数的事件的例子。一般的，这类事件不能被静态的分配（如表(2)里的 TIME_TICK），因为它能在下次 ISR 运行时异步的改变。系统里的一些主动对象可能继续通过指针引用事件，因此事件不能变化。QF 的动态事件分配解决了所有这些并发性问题，因为每一次都有一个新的事件被分配。QF 然后在确定所有要访问这个事件的主动对象结束使用这个事件后，回收这个动态的事件。

(10-11) 事件参数 x,y 被赋值。

(12) 动态事件被从 Ship 主动对象直接发布。

1.7 对层次状态机编码

和广泛传播的误解相反，你不需要大型设计自动化工具去把层次式状态图(UML 状态图)转换成有效的高度可维护的 C 或者 C++代码。本节说明如何利用 QF 实时框架和作为 QF 事件驱动平台一部分的 QEP 层次处

理器，从图 1.6 来对 Ship 状态机进行手工编码。一旦你知道如何编码这个状态机，你就知道如何对所有状态机编码。

Ship 状态机的源代码可在 DOS 版或者 Stellaris 的飞行射击游戏的文件 ship.c 找到。我对这个文件的解释分为 3 步。

1.7.1 第一步：定义 Ship 的结构

在第一步你需要定义 Ship 的数据结构。和事件的例子一样，你使用继承从框架的数据结构 QActive 派生出 Ship 的数据结构。（参考注释“C 语言的单继承”）。建立这样的继承关系令 Ship 结构连接到 QF 框架。

QActive 基本结构的主要任务是储存状态机的当前活动状态的信息，还有 Ship 主动对象的事件队列和优先级。事实上，QActive 本身是从一个比较简单的 QEP 结构 QHsm 派生的，这个 QHsm 表示一个层次状态机的当前活动状态。在这些信息的上层，几乎每一个状态机都必须储存其他的“扩展状态”信息。例如，Ship 对象负责维护 Ship 的位置和在游戏里累积的分数。你通过在基本结构成员 super 后加入其他数据成员，如代码列表 1.4 所示，来提供这些附加的信息。

代码列表 4 在 ship.c 里派生 Ship 结构

```
(1) #include "qp_port.h"                                    /* the QP port */
 (2) #include "bsp.h"                                  /* Board Support Package */
 (3) #include "game.h"                                     /* this application */

    /* local objects -----------------------------------------------------------*/
 (4) typedef struct ShipTag {
 (5)    QActive super;                    /* derive from the QActive struct */
 (6)    uint8_t x;        /* x-coordinate of the Ship position on the display */
 (7)    uint8_t y;        /* y-coordinate of the Ship position on the display */
 (8)    uint8_t exp_ctr;     /* explosion counter, used to animate explosions */
 (9)    uint16_t score;                       /* running score of the game */
(10) } Ship;                       /* the typedef-ed name for the Ship struct */

                                          /* state handler functions... */
(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 */

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

(1) 每一个使用 QP 平台的应用层 C 文件必须包含 qp_port.h 头文件。
(2) bsp.h 头文件包含到板级支持包的接口。
(3) game.h 头文件包含事件和其他在应用程序的组件间共享的功能的声明（见清单 1.2）。
(4) 这个结构定义了 Ship 主动对象。

注：我喜欢使主动对象，和甚至所有状态机对象（比如 Mines）被严格的封装。所以，我不在头文件里放入状态机的结构定义，而是在实现文件比如 ship.c 里定义它们。这样，我能确保 Ship 结构的内部数据成员不被应用程序的其他部分知道。

---

(5) Ship 主动对象结构从框架的 QActive 结构派生，如在“C 语言里的单一继承”所叙。
(6-7) 数据成员 x 和 y 代表 Ship 在显示屏的位置。
(8) 成员 exp_ctr 被用来记录爆炸动画的进度（请看在图 1.6 里 Ship 状态图的“exploding”状态）。
(9) score 成员储存游戏里累积的分数。
(10) 我使用 typedef 来定义 struct ShipTag 得到较短的名字 Ship。
(11-14) 这 4 个函数被称为状态处理函数，因为它们和图 1.6Ship 状态机的状态一一对应。例如函数 Ship_active()代表 activ 状态。QEP 事件处理器调用这些状态处理函数从而实现 UML 的状态机执行语义。所有的状态处理函数有一样的特征。一个状态处理函数使用状态机指针和事件指针作为参数量，并把操作的状态返回给事件处理器，例如事件是否被处理了或没有被处理。返回 QState 类型在头文件<qp>\qpc\¬include\¬qep.h.被定义为 uint8_t。

注：我使用一个简单的命名协议来加强结构和用来操作这些结构的函数之间的联系。首先，我使用 typedef 后的结构名和操作名结合起来（如 Ship_active）.第二，我总是把指向这个结构的指针作为相关函数的第一个参数，并且这个参数总是被命名为 me（如，Ship_active(Ship *me, …)）

---

(15) 除了状态处理函数外，没个状态机必须声明初始化伪状态，QEP 使用它执行最顶层的初始化转移(见图 1.6(1)).初始伪状态处理函数和一般的状态处理函数是一样的写法。
(16) 在这里我静态分配储存给 Ship 主动对象。请注意对象 l_ship 被定义为 static,因此它只能在文件 ship.c 内部被存取。
(17) 这里我定义并初始化全局指针 AO_Ship 指向 Ship 主动对象(见清单 1.2（10）).这个指针是“不透明的”，因为它把 Ship 对象当作一般的 QActive 基本结构对待，而不是具体的 Ship 结构。一个“不透明的”指针的威力在于它允许我完全隐藏 Ship 结构的定义并使它不能被程序的其他部分存取。当然，其他的程序组件能通过 QActive_postFIFO(QActive *me, QEvent const *e)函数直接发布事件给它从而存取 Ship 对象

1.7.2 第二步：初始化状态机

状态机的初始化被分为以下 2 步，以便增加灵活性，更好的控制初始化时间线：

3. 状态机“构造函数”；
4. 最顶层初始化转移。

状态机“构造函数”，如 Ship_ctor(), 特意不去执行在初始伪状态里定义的最顶层初始转移，因为在那时刻可能缺少一些重要的对象，并且硬件也许没有被适当的初始化。相反，状态机“构造函数”仅仅

把状态机放到初始伪装态。然后，用户代码必须明确的触发最顶层初始化转移，这在函数 QActive_start()函数内部发生(见清单 1.1(18-20)). 清单 1.5 展示了 Ship 主动对象的实例化（“构造”函数）和初始化（初始伪状态）。

代码列表 5 ship.c 里 Ship 主动对象的实例化和初始化

```
(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) 全局函数 Ship_ctor()原型在 game.h，并在 main()开始处被调用。

(2) “me“指针指向静态分配的 Ship 对象（见清单 1.4(16)）.

(3) 每一个派生的结构负责初始化从基本结构继承过来的部分。“构造函数“QActive_ctor()把状态机放到初始伪状态&Ship_initial(见清单 1.4(15)).

(4-5) Ship 的位置被初始化。

(6) 函数 Ship_initial()定义 Ship 状态机的最顶层初始化转移（见图 1.6(1)）.

(7-8) Ship 主动对象订阅信号 TIME_TICK_SIG 和 PLAYER_TRIGGER_SIG，如图 1.3（1）的状态图所示。

(9) 通过调用 QP 宏 Q_TRAN()指定初始状态”active”。

注：宏 Q_TRAN()必须一直在 return 语句后。

1.7.3 第三步：定义状态处理函数

在最后一步，你每次把一个状态作为一个状态处理函数实现，从而对 Ship 状态机实际编码。为了决定什么元素属于哪个给定的状态处理函数，你检查着图里每个状态的边界（图 1.6）。你需要实现所有从边界开始的转移，任何在状态里定义的入口和出口动作，还有在状态里列出的所有内部转移。另外，如果有一个初始转移直接嵌入进这个状态，你同样需要实现它。

以图 1.6 里的状态”flying”为例。这个状态有一个入口动作和 2 个在它边界开始的转移：HIT_WALL 和 HIT_MINE(type), 和 3 个内部转移 TIME_TICK, PLAYER_TRIGGER,和 DESTOYED_MIND(score).状态”flying”嵌套在超状态”active”里。

清单 1.6 展示了在图 1.6 的 Ship 状态机的 2 个状态处理函数。这些状态处理函数分别对应于”active”状态和”flying”状态。紧随其后的解释部分说明了一些重要的实现技巧。

代码列表 6 ship.c 里"active"和"flying"的状态处理函数

```
(1) QState Ship_active(Ship *me, QEvent const *e) {
 (2)    switch (e->sig) {
 (3)       case Q_INIT_SIG: {                 /* nested initial transition */
 (4)          /* any actions associated with the initial transition */
 (5)          return Q_TRAN(&Ship_parked);
          }
 (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;

              me->score = 0;                          /* reset the score */
(13)          sev = Q_NEW(ScoreEvt, SCORE_SIG);
(14)          sev->score = me->score;
(15)          QActive_postFIFO(AO_Tunnel, (QEvent *)sev);
(16)          return Q_HANDLED();
          }
          case TIME_TICK_SIG: {
             /* tell the Tunnel to draw the Ship and test for hits */
             ObjectImageEvt *oie = Q_NEW(ObjectImageEvt, SHIP_IMG_SIG);
             oie->x   = me->x;
             oie->y   = me->y;
             oie->bmp = SHIP_BMP;
             QActive_postFIFO(AO_Tunnel, (QEvent *)oie);

             ++me->score;  /* increment the score for surviving another tick */

             if ((me->score % 10) == 0) {         /* is the score "round"? */
                ScoreEvt *sev = Q_NEW(ScoreEvt, SCORE_SIG);
                sev->score = me->score;
                QActive_postFIFO(AO_Tunnel, (QEvent *)sev);
```

```
            }
            return Q_HANDLED();
        }
        case PLAYER_TRIGGER_SIG: {                /* trigger the Missile */
            ObjectPosEvt *ope = Q_NEW(ObjectPosEvt, MISSILE_FIRE_SIG);
            ope->x = me->x;
            ope->y = me->y + SHIP_HEIGHT - 1;
            QActive_postFIFO(AO_Missile, (QEvent *)ope);
            return Q_HANDLED();
        }
        case DESTROYED_MINE_SIG: {
            me->score += ((ScoreEvt const *)e)->score;
            /* the score will be sent to the Tunnel by the next TIME_TICK */
            return Q_HANDLED();
        }
(17)     case HIT_WALL_SIG:
(18)     case HIT_MINE_SIG: {
(19)         /* any actions associated with the transition */
(20)         return Q_TRAN(&Ship_exploding);
        }
    }
(21) return Q_SUPER(&Ship_active);              /* return the superstate */
}
```

(1) 每个状态处理函数必须有相同的特征。也就是说，它必须带有 2 个参数：状态机指针 me 和指向 QEvent 的指针。在事件指针声明*前的关键字 const 意味着这个指针指向的 event 不能在状态处理函数内部被改变（即，这个事件是只读的）。状态处理函数必须返回 QState，它把事件处理的情况传回给 QEP 事件处理器。

(2) 典型的，每一个状态处理函数由 switch 语句构成，用来处理事件 e->sig 的信号部分。

(3) 这条代码属于嵌套的初始转移图 1.6(2). QEP 提供一个保留的信号 Q_INIT_SIG，框架在需要某个状态处理函数执行初始转移时会传递给这个函数。

(4) 你能加入和这个初始转移有关的任何动作。(本例里没有。)

(5) 你使用宏 Q_TRAN()指定目标子状态。这个宏必须在 return 语句后，状态处理函数通过它告诉 QEP 事件处理器转移已经发生了。

注：初始转移必须以某个给定状态的直接的或嵌套的子状态作为目的。初始转移不能以同级状态或上级状态为目标，否则会形成在 UML 里所说的“畸形”状态机。

---

(6) 这条代码处理图 1.6(3)的初始转移 PLAYER_SHIP_MOVE_SIG(x, y).

(7-8) 你通过状态处理函数的 me 参数来存取 Ship 状态机的数据成员。通过 e 参数存取事件参数。你需要把事件指针从通用 QEvent 基本结构进行强制类型转换到特定的需要 PLAYER_SHIP_MOVE_SIG 的事件结构上，本例里这个结构是 ObjectPosEvt。

注：事件信号和事件结构（事件变量）的关系在事件发生时被建立起来。所有这个事件的接收者必须知道这个联系从而把它类型转换到正确的事件结构。

(9) 你使用”return Q_HANDLED()”结束 case 语句，这通知 QEP 事件处理器这个事件已经被处理了（但是没有转移发生）。

(10) 从一个状态处理函数最后的 renturn 语句通过 QEP 宏 Q_SUPER()指定它的超状态。这个从状态处理函数最后的那条 return 语句是改变某个给定状态的嵌套层次的唯一的维护点。在图 1.6 的 active 状态没有显式的超状态，意味着它隐式的嵌套于 top 状态。‘top’状态是 UML 的概念，表明在一个层次式状态机里的最后的那个状态。QPE 提供的’top’状态是状态处理函数 QHsm_top(), 因此 Ship_active()状态处理函数返回指针&Qhsm_top.

注：在 C 和 C++里，一个函数指针 QHsm_top()可以写成 QHsm_top 或&QHsm_top. 尽管 QHsm_top 表达更加简洁，我更喜欢使用表达式&QHsm_top，以表明这里我确实需要一个函数指针&QHsm_top.

(11) 本行代码处理进到状态 flying 的入口动作（图 1.6(5)）.QEP 提供了一个保留的信号 Q_ENTRY_SIG，当某个状态处理函数想执行入口动作时，由框架传送给它。

(12) 到 flying 的入口动作发送 SCORE 事件到 Tunnel 主动对象（图 1.6(5)）本行代码定义了一个指向事件结构 ScoreEvt 的临时指针。

(13) QF 宏 Q_NEW(ScoreEvt, SCORE_SIG)从一个由 QF 管理的事件池里动态分配了 ScoreEvt 的一个实例。这个宏把 SCORE_SIG 和被分配的事件联合起来。宏 Q_NEW()返回一个指向被分配的事件的指针。

(14) ScoreEvt 的 score 参数根据状态机成员 me->score 设定。

(15) 事件 SCORE(me->score)通过 QP 函数 QActive_postFIFO()被直接发送给主动对象 Tunnel。这个函数的参数是接收者主动对象（这里是 AO_Tunnel）,和指向这个事件的指针（这里是临时指针 sev）。

(16) 你使用 return Q_HANDLED()结束 case 语句，这会通知 QEP 入口动作已经被处理了。

(17-18) 这 2 条代码处理从 flying 到 exploding 的状态转移（图 1.6(9,10)）.

(19) 你可以加入和这个转移有关的任何动作。（本例这里没有别的动作）。

(20) 你使用宏 Q_TRAN()指定转移的目的。

(21) 一个状态处理函数的最后一个 return 指定了这个状态的超状态。图 1.6 的状态 flying 嵌套在状态 active 里，因此状态处理函数 Ship_flying()返回指针&Ship_active.

当实现状态处理函数时你需要注意在这里由 QEP 事件处理器而不是你的代码管理。QEP 将根据不同的原因调用某个状态处理函数：层次事件处理，执行入口和出口动作，触发初始转移，或引出某给定状态处理函数的超状态。因此，你不能假设某个状态处理函数仅仅用来处理在那些在 case 语句后的信号。你必须消除任何在 switch 语句的代码，特别是那些有副作用的代码。

1.8 执行模式

如你在程序列表 1.1(21)看到的，main()函数最后通过调用 QF_run()把控制交给了事件驱动框架。本节里我简要的解释 QF 如何分配 CUP 时间给系统里不同的任务，以及你在选择执行模式时拥有的选项。

1.8.1 简单的非抢占式 Vanilla 调度器

飞行射击实例程序使用了最简单的 QF 配置，这种配置里 QF 直接运行于目标处理器上，而不用任何底层的操作系统或者内核2。我把这种 QF 配置称做 plain vanilla 或 vanilla。

QF 包含一个简单的非抢占式 vanilla 内核，它使用一个无限循环（和超级循环类似），每次执行一个主动对象。在每个事件以运行-到-完成(RTC)方式被处理后，Vanilla 内核开始选择下一个准备处理下个事件的主动对象。Vanilla 调度器是合作式的。这意味着所有的主动对象合作分享一个单独的 CPU 并在每一个 RTC 步骤后把控制转移给别的对象。内核是非抢占式的，意味着每一个主动对象必须在任何其他主动对象能开始处理别的事件时处理完一个事件。

中断服务程序 ISR 能在任何时候抢占一个主动对象的执行，但是由于 vanilla 内核的简单化性质，每个 ISR 精确的返回到抢占入口点。如果 ISR 发送或发行一个事件给任何主动对象，对这个事件的处理只有在当前的 RTC 步骤完成后才会开始。一个最高优先级主动对象的事件能被延迟的最大时间被称为任务级响应。当使用非抢占式 vanilla 内核时，任务级响应等于系统中所有主动对象的最长的 RTC 步骤。请注意 vanilla 的任务级响应还是比传统的“超级循环“（即 main+ISR）架构要好很多。我在下面的 1.9 节对此有更多的解释，那里我会比较事件驱动式飞行射击实例和传统的 Quickstart 应用程序。

简单的 vanilla 内核的任务级响应适合非常多的应用程序，因为状态机本身处理事件非常快不需要为事件而忙等待。（一个状态机简单的运行-到-完成然后保持静止直到下个事件到达）。还请注意你也可以通过把较长的 RTC 步骤分解成较短的 RTC 从而使任务级响应如你所需的快。（例如使用在第 5 章描叙的 Reminder 状态模式）。

1.8.2 QK 抢占式内核

某些情况下，把较长的 RTC 步骤分解成足够短的片段也许很困难，从而非抢占式 vanilla 内核的任务级响应会太长了。比如一个 GPS 接收机系统。这个接收机在一个定点 CUP 上执行大量的浮点运算，计算 GPS 的位置。同时，GPS 接收机必须跟踪 GPS 卫星的信号，这牵涉到在小于毫秒级间隔内的闭环控制回路。很明显我们不容易把位置计算分解成足够短的 RTC 步骤从而允许可靠的信号跟踪。

但是状态机执行的 RTC 语义不意味着一个状态机在 RTC 步骤时一定要垄断 CUP。一个抢占是内核可以在一个较长的 RTC 步骤中间进行一个上下文切换，让一个较高优先级的主动对象开始运行。只要这些主动对象没有共享资源，它们可以并发的运行，并且独立的完成它们各自的 RTC 步骤。

QP 事件驱动平台包含了一个微小的，完全可抢占式，基于优先级的实时内核组件，即 QK，它被明确的设计成使用 RTC 方式处理事件。配置 QP 使用抢占是 QK 内核很容易，但是使用任何完全可抢占式内核时，你必须十分注意在主动对象之间共享的任何资源3。飞行射击实例被特意射击成避免在主动对象之间共享任何资源，因此这个应用程序代码可以不许改动而运行在 QK 抢占式内核上，或任何其他的抢占式内核或实时操作系统。附随代码包含了使用 QK 的飞行射击游戏实例，在<qp>\qpc\examples\80x86**qk**\-tcpp101\l\game\目录。你可以在任何标准的 Windows 个人电脑上的 DOS 控制台运行它。

2 80x86 版的飞行射击游戏运行于 DOS 上面，但是 DOS 并不提供任何的多任务支持。

3 QK 提供了一个互斥功能以便于互斥的存取共享的资源。QK 互斥体使用优先级-天花板协议避免优先级反转。细节请参考第 10 章。

1.8.3 传统的操作系统/实时操作系统

QP 也可以和传统的操作系统(OS)，如 Windows 或 Linux 或几乎任何实时操作系统(RTOS)一起工作，从而使用现有的设备驱动，通讯堆栈，和其他的中间件。

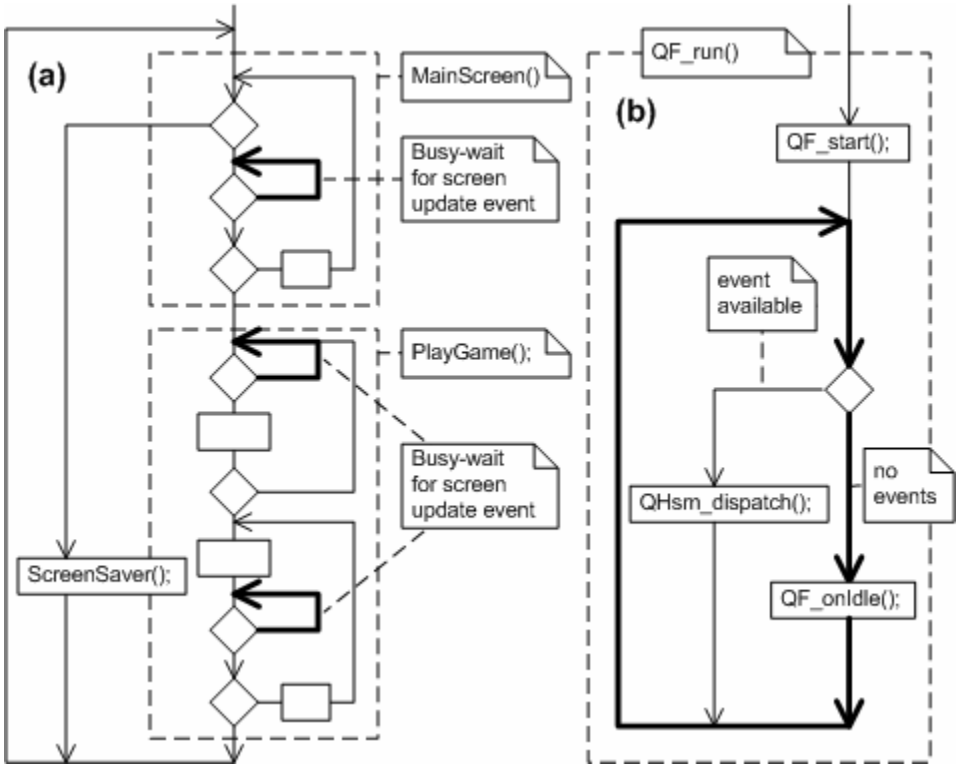
QP 包含一个平台抽象层(PAL),这使移植 QP 到几乎任何操作系统变得很容易。被仔细设计的 PAL 允许通过复用任何预备的中断管理，消息队列和内存划分等等这些功能，从而和底层 OS/RTOS 的紧密的集成。在第 8 章我会讨论如何移植 QP。

1.9 和传统方法的比较

飞行射击游戏的运行被特意和在 Luminary Micro Stellaris EV-LM3S811 演示版上提供源代码的 Quickstart 应用程序保持一致。[Luminary 06]。这节我想比较由 Quickstart 应用程序为代表的传统的方法和以飞行射击游戏为代表的基于状态机的解决方法。

图 1.11(a)展示了 Quickstart 的流程图，图 1.11(b)展示了运行在合作式 vanilla 内核上的飞行射击游戏的流程图。在最高层次，这写流程图是类似的，他们 2 者都包含一个围绕整个处理过程的无限循环。但是这 2 者主循环的内部结构是非常不同的。如在流程图里粗线所标识的，Quickstart 程序把绝大多数时间用于紧密的用于忙等待特定事件的‘事件循环‘内，如屏幕更新事件。相反，飞行射击程序把绝大多数事件用在主循环内。QP 框架分派任何可用的事件到适当的处理这个事件的状态机，并很快的返回主循环，而不是在那里等待事件。

图 9Quickstart 应用程序(a) 和飞行射击(b)的控制流,.粗线表示在代码里最经常执行的路径



Quickstart 程序和飞行射击实例相比有更多的循环式控制流程，因为传统的解决办法是完全针对手头的问题，而状态机方案是通用性的。Quickstart 应用程序的结构和传统的顺序式程序非常相似，它尽量从头到尾保持控制。它不时的停住，忙等待某个事件，代码基本上没有准备好去处理任何它在等待的事件以外的别的事件。这些都是由于设计的不灵活性导致。增加新的事件是困难的，因为需要更改的代码的整个结构被设计为只接收很特定的事件，而且需要被剧烈的改变从而容纳新的事件。同样，当忙等待屏幕更新事件（等效为飞行射击游戏里的 TIME_TICK 事件）时，程序不会对任何其他的时间产生反应。任务级响应难以被确定，一般的它和事件的类型有关。通过硬编码确定的等待时序适合现有的事件但是也许不会适合新的事件。

相反，飞行射击游戏有更简单的控制流程，纯粹的事件驱动型，完全通用（见图 1.11(b)）。每个主动对象的上下文表示为一个状态机的当前状态，而不是某处的代码。这样，根据当前上下文在代码某处进行“事件循环”变得没必要了。相反，状态机把上下文当做一个很小的数据成员（见第 3 章状态变量）的方法，是很有效率的。在处理完每个事件后，状态机能返回为被设计成能处理全部事件的公共事件循环。状态机自然的挑选每一个事件并移动到下一个状态，如果必要的话。在这种设计里增加新的事件很容易，因为状态机在任何时候可以响应任何事件。一个事件驱动的基于状态机的应用程序，和传统的程序相比，更加灵活，更有弹性。

注：通用事件循环也能很容易的探测到没有有效事件的情况，这种情况下，QP 框架调用 QF_onIdle()函数（见图 1.11(b)）。这个回调函数被设计为由应用程序定制它的功能，所以是一个理想的使 CPU 进入低功耗睡眠模式以节省能量的地方。相反，传统的方法不会提供任何单一的地点以便转移到低功耗睡眠模式，因此很不容易实现一个完全低功耗的设计。

1.10 小结

如果你以前没接触过事件驱动编程，飞行射击游戏的内部结构对你肯定代表一个很大的转变。事实上，我希望它可以改变你的想法，否则我不能确信你真的领会到一个事件驱动程序和传统的顺序式代码相比完全的控制反转。这种控制的反转，也被成为“好莱坞原则”(不要调用我们，我们将调用您)，使很多新用户感到困惑，他们常常感到“难以致信”，“落后”，或“古怪”。

飞行射击游戏不是一个大的应用程序，但它肯定不是微不足道的。你不必担心第一次阅读它时你是否能完全理解它。在接下来的章节里，我会提供对状态机设计和编码技巧更深入的研究。在第 2 部分，我会讨论 QF 实时框架的特征，实现和移植。

本章我主要的目的是向你介绍事件驱动范例和现代状态机，让你确信这些强大的概念不是特别难以用 C 或 C++直接实现。实际上，我希望你注意到编写飞行射击游戏的代码完全并不是很大份量的任务。你需要知道的只是一些为状态机编码的固定规则，并熟悉一些实现动作所需要的框架服务。

当编码基本上不成问题时，大部分编程工作就花在设计应用程序上了。从这点来说，我希望飞行射击实例帮你对这个方法如何工作有个大概的理解。在事件驱动模型里，程序结构被大体的分为 2 组：事件和状态机组件（主动对象）。事件代表某些感兴趣的某事的发生。状态机根据事件的性质和主动对象的状态对这个事件的反应进行编码。事件常常从你程序的外部产生，比如时钟滴答或飞行射击游戏里的按键被按下，事件也可以被程序从内部自己产生。例如 Mine 组件在它们检测到和 Missile 或 Shipp 碰撞时，产生通知事件。

一个事件驱动程序运行时，不停的查看可能的事件，当一个事件被探测到，分派这个事件到合适的状态机组件（见图 1.11(b)）.为了让这个方法可以工作，必须连续频繁的查看事件。这意味者状态机必须

很快的运行，这样程序能返回来查看事件。为了符合这个要求，状态机不能进入到需要忙等待一段较长或不确定时间的状态。这种状态最普遍的例子就是在一个状态处理函数内的 while 循环，推出循环的条件，比如按键输入，不受程序的控制。这类程序结构，一个无限循环，被称为“阻塞“代码4，你可以在 Quickstart 应用程序里看到这些例子（见图 1.11(a)）.为了让事件驱动编程模式能工作，你必须只编写“非阻塞”代码[Carryer 05]。

最后，飞行射击实例演示了使用了 QF 事件驱动平台，它是一些用来建立事件驱动应用程序的组件的集合。QF 实时框架组件用“好莱坞原则“而不是别的方法去调用应用程序代码。这种安排在事件驱动系统里非常典型，和 QF 类似的应用程序框架是基本上现在市场上每一个设计自动化工具的核心部分。

在飞行射击游戏里运转的 QF 框架使用它最简单的配置，这样 QF 可以直接运行在目标处理器上而不使用任何操作系统。QF 也可以被配置从而使用它内置的抢占式实时内核 QK（见第 10 章），或者被很容易的移植到几乎任何传统的 OS 或 RTOS（见第 8 章）。事实上，你可以把 QF 框架本身看作是一个高级的事件驱动实时操作系统。

4 在多任务操作系统的环境下，“阻塞“代码对应着流程在等待一个旗语，事件标志，消息邮箱，或其他操作系统原语。

第 2 章

UML 状态机速成

一个我们可能使用帮助的地方是用于优化 IF-THEN-ELSE 结构。绝大多数程序开始时相当好的结构。当发现了一个缺陷以及新的特征需要增加时，IF 和 ELSE 被陆续加入，直到没有人能知道数据是如何流过一个函数的。漂亮的打印代码有点帮助，但是不能减少嵌套达 15 层的 IF 语句的复杂性。

--Jack Ganssle, "Break Points", ESP Magazine, January 1991

2.1 事件-动作范例的简化

2.2 基本的状态机概念

2.2.1 状态

2.2.2 状态图

2.2.3 状态图和流程图的比较

2.2.4 扩展状态机

2.2.5 警戒条件

2.2.6 事件

2.2.7 动作和转移

2.2.8 运行-到-完成执行模式

2.3 UML 对传统 FSM 方法的扩展

2.3.1 反应性系统里的行为重用

2.3.2 层次式嵌套状态

第 3 章

标准状态机的实现方法

专家就是在一个很小的领域犯了所有可能犯的错误的那个人。
--Niels Bohr

第 4 章

层次式事件处理器的实现

...增加一个特征的成本不仅是需要编写它的时间。成本还包括所增加的对未来扩展性的障碍...诀窍在于挑选不会彼此反对的特征。

---John Carmack

4.1 QEP 事件处理器的关键特征

4.2 QEP 的结构

4.2.1 QEP 源代码的组织

4.3 事件

4.3.1 事件信号(QSignal)

4.3.2 C 语言实现的 QEvent

4.3.3 C++语言实现的 QEvent

4.4 层次式状态处理函数

4.4.1 标识超状态(宏 Q_SUMPER())

4.4.2 C 语言实现的层次式状态处理函数

4.4.3 C++语言实现的层次式状态处理函数

4.5 层次式状态机的类

4.5.1 C 语言实现的层次式状态机（QHsm 结构）

4.5.2 C++语言实现的层次式状态机（QHsm 类）

第 5 章

状态模式

科学就是成功方法的集合。
--Paul Valery

第二部分 实时框架

第 6 章

实时框架的概念

“不要调用我们，我们将调用您”（好莱坞原则）
--Richard E. Sweet, The Mesa Programming Environment 1985

6.1 控制的反转

6.2 CPU 管理

6.2.1 传统的顺序式系统

6.2.2 传统的多任务系统

6.2.3 传统的事件驱动型系统

6.3 主动对象计算模式

6.3.1 系统结构

6.3.2 异步通讯

6.3.3 运行-到-完成

6.3.4 封装

6.3.5 对状态机的支持

6.3.6 传统的可抢占式内核/RTOS

6.3.7 合作式 Vanilla 内核

6.3.8 可抢占式 RTC 内核

6.8 基于框架的软件追踪

6.9 小结

第 7 章

实时框架的实现

让我们改变我们对程序结构的传统态度。不要想象我们主要的任务是指导一台电脑去做什么，让我们集中精力向人类解释，我们希望电脑做什么。
--Donald E. Knuth

7.1 QF 实时框架的关键特征

7.1.1 源代码

7.1.2 可移植性

7.1.3 可裁剪性

7.1.4 对现代状态机的支持

7.1.5 直接事件发送和发布-订阅式事件派发

7.1.6 零复制的事件内存管理

7.1.7 开放式序号的时间事件

7.1.8 原生的事件队列

7.1.9 原生的内存池

7.1.10 内置 Vanilla 调度器

7.1.11 和 QK 可抢占式内核的紧密集成

7.1.12 低功耗架构

7.1.13 基于断言的错误处理

7.1.14 内置软件追踪测试设备

第 8 章

移植和配置 QF

作为一个规则，软件系统只有当它们在实际应用中被使用过并经过了多次失败以后，才会正常工作。

-David Parns

8.1 QP 平台抽象层

8.1.1 生成 QP 应用程序

8.1.2 生成 QP 库

8.1.3 目录和文件

8.1.4 头文件 qep_port.h

8.1.5 头文件 qf_port.h

和平台相关 QActive 数据成员的类型
为 QActive 派生的基类
在应用系统中主动对象的最大数目
在 QF 框架里不同的对象规模
QF 的临界区机制
在 QF 移植里使用的包含文件
仅在 QF 内而不在应用程序里使用的接口
主动对象事件队列的操作
QF 事件迟的操作

8.1.6 源代码 qf_port.c

8.1.7 头文件 qp_port.h

8.1.8 和平台相关的 QF 回调函数

8.1.9 系统时钟节拍（调用 QF_tick()）

8.1.10 生成 QF 库

8.2 移植合作式 Vanilla 内核

8.2.1 头文件 qep_port.h

8.2.2 头文件 qf_port.h

8.2.3 系统时钟节拍（QF_tick()）

8.2.4 空闲处理（QF_onIdel()）

8.3 QF 移植到 uc/os-II (常规 RTOS)

8.3.1 头文件 qep_port.h

8.3.2 头文件 qf_port.h

8.3.3 源代码 qf_port.c

8.3.4 生成 uc/os-II 移植

8.3.5 系统时钟节拍（QF_tick()）

8.3.6 空闲处理

8.4 QF 移植到 Linux（常规 POSIX 兼容的操作系统）

8.4.1 头文件 qep_port.h

8.4.2 头文件 qf_port.h

8.4.3 源代码 qf_port.c

8.5 小结

第 9 章

开发 QP 应用程序

例子不是影响他人的主要素材，它是唯一的素材。
--Albert Schweitzer

9.4 调整事件队列和事件池

9.4.1 调整事件队列

9.4.2 调整事件池

9.4.3 系统集成

9.5 小结

第 10 章

可抢占式‘运行-到-完成’内核

简单是效率的灵魂
--R.Austin Freeman (in The Eye of Osiris)

第 11 章

事件驱动式系统的软件追踪

在计算机的历史里一次耗时出乎意料短的调试过程从来没有存在过。
--Steven Levy

11.1 软件追踪的概念

11.2 Quantum Spy 软件追踪系统

11.2.1 一个软件追踪会话的实例

11.2.2 具有人类可读性的追踪输出

11.3 QS 目标组件

11.3.1 QS 源代码组件

11.3.2 QS 的平台无关性头文件 qs.h 和 qs_dummy.h

11.3.3 QS 的临界段

11.3.4 QS 记录的一般结构

11.3.5 QS 的过滤器

全局开/关过滤器
本地过滤器

11.3.6 QS 数据协议

透明
大小端

11.3.7 QS 追踪缓存区

初始化 QS 追踪缓存区 QS_initBuf()
面向字节的接口：QS_getByte()

面向块的接口：QS_getBlock()

11.3.8 字典追踪记录

对象字典
功能字典
信号字典

11.3.9 应用程序相关的 QS 追踪记录

11.3.10移植和配置 QS

11.4 QSPY 主机应用程序

11.4.1 安装 QSPY

11.4.2 从源代码生成 QSPY

使用 Visual C++ 2006 为 Windows 生成 QSPY
使用 MinGW 为 Windows 生成 QSPY
为 Linux 生成 QSPY

11.4.3 使用 QSPY

11.5 向 MATLAB 输出追踪数据

11.5.1 使用 MATLAB 分析追踪数据

11.5.2 MATLAB 输出文件

11.5.3 MATLAB 脚本 qspy.m

11.5.4 由 qspy.m 产生的

11.6 向 QP 应用程序添加 QS 软件追踪

11.6.1 初始化 QS 并安排过滤器

11.6.2 定义平台相关的 QS 回调函数

11.6.3 使用回调函数 QS_onGetTime()产生 QS 时间戳

11.6.4 从主动对象产生 QS 字典

11.6.5 添加应用程序相关的追踪记录

11.6.6 QSPY 参考手册

11.7 小结

第 12 章

QP-nano：你能变多小？

地球上所有的生物都是昆虫。。。
--Scientific American, 2001-七月刊

12.1 QP-nano 的关键特征

12.2 使用 QP-nono 实现飞行射击游戏实例

12.2.1 main()函数

12.2.2 头文件 qpn_port.h

12.2.3 在飞行射击游戏程序里的信号，事件和主动对象

12.2.4 在 QP-nano 里实现 Ship 主动对象

12.2.5 QP-nano 的时间事件

12.2.6 飞行射击游戏程序在 QP-nano 里的版支持包

12.2.7 生成 QP-nano 上的飞行射击游戏程序

12.3 QP-nano 的结构

12.3.1 QP-nano 的源代码，实例和文档

12.3.2 QP-nano 的临界区

任务级中断上锁
ISR 级中断上锁

12.3.3 QP-nano 里的状态机

12.3.4 QP-nano 里的主动对象

附录 A QP 和 QP-nano 的许可证政策

附录 B 标识法

参考文献

索引