# Table of Contents

1 Introduction .................................................................................................................................................. 1

2 QP Active Object Frameworks .................................................................................................................. 2
  2.1 Asynchronous Communication ................................................................................................................. 2
  2.2 True Encapsulation for Concurrency ........................................................................................................ 2
  2.3 Run-to-Completion Event Processing ...................................................................................................... 2
  2.4 No Blocking ............................................................................................................................................... 3
  2.5 Super-Fast Preemptive Kernel .................................................................................................................. 3
  2.6 Hierarchical State Machines ................................................................................................................... 3
  2.7 Graphical Modeling and Code Generation Tool ......................................................................................... 4
  2.8 QS/QSPY Software Tracing .................................................................................................................... 4
  2.9 Motor Industry Software Reliability Association (MISRA) Compliance ..................................................... 4
  2.10 Documentation and Support ................................................................................................................ 4

3 Related Documents and References ......................................................................................................... 5

4 Contact Information ..................................................................................................................................... 6

---

## Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.
ABSTRACT:
Active objects (actors) are encapsulated state machines that run in their own thread of execution and process events asynchronously using an event-driven receive loop. They inherently support and automatically enforce the best practices of concurrent programming, such as: keeping the thread's data local and bound to the thread itself, asynchronous inter-thread communication without blocking, and using state machines instead of convoluted IF-ELSE logic (a.k.a. "spaghetti" code). In contrast, raw RTOS-based threading lets you do anything and offers no help or automation for the best practices.

1 Introduction

Active objects (a.k.a. actors) are event-driven, strictly encapsulated software objects endowed with their own threads of control that communicate with one another asynchronously by exchanging events. The UML specification further proposes the UML variant of hierarchical state machines (UML statecharts) with which to model the behavior of event-driven active objects.

Active objects inherently support and automatically enforce the following best practices of concurrent programming:

- Keep all of the thread's data encapsulated and local, bound to the task itself and hidden from the rest of the system.
- Communicate among threads asynchronously via intermediary event objects. Using asynchronous event posting keeps the tasks running truly independently without blocking on each other.
- Threads should spend their lifetime responding to incoming events, so their mainline should consist of an event loop.
- Threads should process events one at a time (run to completion), thus avoiding any concurrency hazards within a thread itself.

Active objects dramatically improve your ability to reason about the concurrent software. In contrast, using raw RTOS tasks directly is trouble for a number of reasons, particularly because raw tasks are undisciplined and offer you no help or automation for the best practices. Active objects provide a more productive architecture, which is safer to use, more efficient, and easier to understand, extend, test, and maintain. As with all good patterns, active objects raise the level of abstraction above the naked threads and let you express your intent more directly thus improving your productivity.

HISTORICAL NOTE: The concept of autonomous software objects communicating by message passing dates back to the 1970s, when Carl Hewitt at MIT developed a notion of an actor. In the 1990s, methodologies like ROOM adapted actors for real-time computing. More recently, UML has introduced the concept of active objects that is essentially synonymous with the ROOM actor. Today, the actor model is all the rage in the enterprise computing, because it can deliver levels of reliability and fault tolerance unachievable really with the “free threading” approach. A number of actor programming languages (e.g., Erlang, Scala, D) as well as actor libraries and frameworks (e.g., Akka, Killim, Jetlang) are in extensive use. In the real-time embedded space, active objects frameworks provide the backbone of various modeling and code generation tools. Examples include: IBM Rational Rhapsody (with OXF/SXF frameworks) and National Instruments LabVIEW (with LabVIEW Actor Framework).
2 QP Active Object Frameworks

Active objects cannot operate in a vacuum and require a software infrastructure (framework) that provides, at a minimum, an execution thread for each active object, queuing of events, and event-based timing services. In the resource-constrained embedded systems, the biggest concern has always been about scalability and efficiency of such frameworks, especially that the frameworks accompanying various modeling tools have traditionally been built on top of a conventional RTOS, which adds memory footprint and CPU overhead to the final solution.

The QP frameworks have been designed for efficiency and minimal footprint from the ground up and do not need an RTOS in the stand-alone configuration. In fact, when compared to conventional RTOSes shown on the left, QP frameworks provide the smallest footprint especially in RAM (data space), but also in ROM (code space). This is possible, because active objects don't need to block, so most blocking mechanisms (e.g., semaphores) of a conventional RTOS are not needed.

All these characteristics make event-driven active objects a perfect fit for single-chip microcontrollers (MCUs). Not only you get the productivity boost by working at a higher level of abstraction than raw RTOS tasks, but you get it at a lower resource utilization and better power efficiency, because event-driven systems use the CPU only when processing events and otherwise can put the MCU in a low-power sleep mode.

2.1 Asynchronous Communication

Each active object has its own event queue and receives all events exclusively through this queue. Events are delivered asynchronously, meaning that an event producer merely posts an event to the event queue of the recipient active object but doesn't wait in line for the actual processing of the event. The event processing occurs always in the thread context of the recipient active object. The QP framework is responsible for delivering and queuing the events in a thread-safe and deterministic manner.

2.2 True Encapsulation for Concurrency

In a sense active objects are the most stringent form of object-oriented programming (OOP), because the asynchronous communication enables active objects to be truly encapsulated. In contrast, the traditional OOP encapsulation, as provided by C++, C# or Java, does not really encapsulate anything in terms of concurrency. Any operation on an object runs in the caller's thread and the attributes of the object are subject to the same race conditions as global data, not encapsulated at all. To become thread-safe, operations need to be explicitly protected by a mutual exclusion mechanism, such as a mutex or a monitor, but this reduces parallelism dramatically, causes contention, and is a natural enemy of scalability.

In contrast, all private attributes of an active object are truly encapsulated without any mutual exclusion mechanism, because they can be only accessed from the active object's own thread. Note that this encapsulation for concurrency is not a programming language feature, so it is no more difficult to achieve in C as in C++, but it requires a programming discipline to avoid sharing resources (shared-nothing principle). However, the event-based communication helps immensely, because instead of sharing a resource, a dedicated active object can become the manager of the resource and the rest of the system can access the resource only via events posted to this manager active object.

2.3 Run-to-Completion Event Processing

Each active object handles events in run-to-completion (RTC) fashion, which also is exactly the semantics universally assumed by all state machine formalisms, including UML statecharts. RTC simply means that an active object handles one event at a time, that is, the active object must complete the processing of an event before it can start processing of the next event from its queue.

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

2.4 No Blocking

Most conventional RTOS kernels manage the tasks and all inter-task communication based on blocking, such as waiting on a semaphore. However, blocking is problematic, because while a task is blocked waiting for one type of event, the task is not doing any other work and is not responsive to other events. Such a task cannot be easily extended to handle new events.

In contrast, event-driven active objects don't need to block, because in event-driven systems the control is inverted compared to traditional RTOS tasks. Instead of blocking to wait for an event, an active object simply finishes its RTC step and returns to the framework to be activated when the next event arrives. This arrangement allows active objects to remain responsive to events of all types, which is central to the unprecedented flexibility and extensibility of active object systems.

The QP frameworks provide all mechanisms you might need for non-blocking operation. For example, instead of delaying an active object with a blocking delay() call, you can use a time event to arrange activation in the specific time in the future.

2.5 Super-Fast Preemptive Kernel

While the active object model can work with a traditional blocking RTOS, it can also work with a much simpler non-blocking, run-to-completion kernel (see also basic tasks in OSEK/VDX). The QP frameworks provide such a super-simple and super-fast kernel called QK, which provides fully preemptive multitasking using a single stack for all active object threads.

The fixed-priority, preemptive QK kernel meets all the assumptions of the Rate Monotonic Analysis (RMA) to ensure schedulability of active object's threads. In fact, the non-blocking execution model makes the RMA method much simpler to apply to a system of active objects than to a set of RTOS tasks.

2.6 Hierarchical State Machines

The behavior of each active object in QP is specified by means of a hierarchical state machine (UML statechart), which is the most effective and elegant technique of decomposing event-driven behavior. The most important innovation of UML state machines over classical FSMs is the hierarchical state nesting. The value of state nesting lies in avoiding repetitions, which are inevitable in the traditional "flat" FSM formalism and are the main reason for the "state-transition explosion" in FSMs. The semantics of state nesting allow substates to define only the differences of behavior from the superstates, thus promoting sharing and reusing behavior.

NOTE: The hallmark of the QP implementation technique is traceability, which is direct, precise, and unambiguous mapping of every state machine element to human-readable C or C++ code. Preserving the traceability from requirements through design to code is essential for mission-critical systems, such as medical devices or avionic systems.
2.7 **Graphical Modeling and Code Generation Tool**

The QP state machine frameworks make also excellent targets for automatic code generation, which is provided by a graphical modeling tool called QM (QP Modeler). QM is a freeware, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP active object frameworks. QM is available for 64-bit Windows, 64-bit Linux, and Mac OS X.

QM provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM eliminates coding errors by automatic generation of compact, production-quality, MISRA-compliant C or C++ code that is 100% traceable from your design.

2.8 **QS/QSPY Software Tracing**

QS™ (QP Spy) is a real-time tracing instrumentation built into the QP/C and QP/C++ active object frameworks and available also to the applications. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS event logging is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression. QS can be configured to send the real-time data out of the serial or Ethernet port of the target device, or even write the data to a file.

2.9 **Motor Industry Software Reliability Association (MISRA) Compliance**

The QP/C and QP-nano frameworks comply with most of the MISRA-C:2004 rules while the QP/C++ framework complies with most of the MISRA-C++:2008 rules. All deviations are carefully limited into very specific contexts and are documented with the MISRA Compliance Matrices. The QP frameworks go even beyond MISRA, by complying with the strict type checking of PC-Lint and a very consistent, documented Quantum Leaps Coding Standard.

All QP framework types come with extensive support for automatic rule checking by means of PC-Lint, which is designed not just for proving compliance of the QP framework code, but more importantly, to aid in checking compliance of the application-level code. Any organization engaged in designing safety-related embedded software could benefit from the unprecedented quality infrastructure built around the QP frameworks.

(NOTE: MISRA and MISRA C are registered trademarks of MIRA Ltd, held on behalf of the MISRA Consortium.)

2.10 **Documentation and Support**

The QP family of active object frameworks comes with the most thorough documentation in the industry. In fact, we have literally written books on active object frameworks and state machines for embedded systems.

The latest book, Practical UML Statecharts in C/C++, 2nd Edition: Event-Driven Programming for Embedded Systems by Miro Samek, is the most popular text on UML statecharts, event-driven programming, and active objects for embedded systems. This ultimate resource describes all the related concepts and provides a very detailed design study of the QP frameworks.

The book is augmented by the extensive library of Application Notes, Articles and a popular “State Space” blog. Additionally, QP Development Kits (QDKs) for specific boards and compilers are accompanied by detailed User Manuals.
## 3 Related Documents and References

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
</table>
| “Practical UML Statecharts in C/C++, Second Edition” [PSiCC2], Miro Samek, Newnes, 2008 | Available from most online book retailers, such as Amazon.com.  
  See also: [http://www.statemachine.com/psicc2.htm](http://www.statemachine.com/psicc2.htm) |
| Free QM graphical modeling and code generation tool, Quantum Leaps, 2011 | [http://www.statemachine.com/qm](http://www.statemachine.com/qm) |
  [http://www.statemachine.com/resources/samek06.pdf](http://www.statemachine.com/resources/samek06.pdf) |
4 Contact Information

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

WEB: http://www.state-machine.com
Support: sourceforge.net/projects/qpc/forums/forum/668726