The embedded software industry is in the midst of a major revolution. Tremendous amount of new development lies ahead. This new software needs an actual architecture that is safer, more extensible, and easier to understand than the usual “shared-state concurrency and blocking” based on a traditional Real-Time Operating System (RTOS).

Quantum Leaps' QP™ real-time embedded frameworks (RTEFs) and the QM™ model-based design (MBD) tool provide such a modern, event-driven architecture based on active objects (actors), hierarchical state machines (UML statecharts), software tracing, unit testing, model-based engineering and automatic code generation.
You can’t just look at the QP™ real-time embedded frameworks and the QM™ model-based design tool as a collection of features, because some of the features will make no sense in isolation. You can only use these powerful tools effectively if you are thinking about the overall architecture and design of your system, not simply coding. And to understand the tools and the underlying concepts that way, you must understand the problems with programming real-time and embedded (RTE) systems in general.

Therefore this presentation starts with discussing problems inherent in RTE systems, why they are problems, and how active object frameworks and hierarchical state machines can help.
Some of the most difficult problems with real-time and embedded (RTE) programming are related to **concurrent code execution** → these problems are usually intermittent, subtle, hard-to-reproduce, hard-to-isolate, hard-to-debug, and hard-to-remove → they pose the highest risk to the project schedule

**#1 Problems due to sharing of resources:**
- Endemic to all **shared-state** systems (main+ISR and RTOS)
- The ripple-effects of preemption in shared-state systems:
  → **Race conditions** → **failure** (if unaddressed)
  → mutual exclusion → **blocking** → missed deadlines

**#2 Problems caused by threads synchronization by blocking:**
- Endemic to most conventional RTOS
  → lack of responsiveness → more threads → more mutual exclusion
  → more blocking … → **architectural decay**

- **No really good options!**
Experts in the field have learned to avoid shared-state concurrency and to avoid blocking to synchronize their threads. Instead, experts apply the following **best practices** of concurrent programming:

1. Keep data and resources **encapsulated** inside threads (“share-nothing” principle) and use **events** to share information

2. Communicate among threads **asynchronously** via **event** objects → Threads run truly independently, without **blocking** on each other

In other words, experts combine multi-threading with **event-driven programming**:
→ Threads are organized as “message pumps” (event queue + event loop)
→ Threads process one event at a time (**Run-to-Completion**, RTC)
→ Threads block only on empty queue and **don't block** anywhere else

Such event-driven, asynchronous, non-blocking, encapsulated threads are called **Active Objects** (a.k.a. **Actors**)

(*) Herb Sutter "Prefer Using Active Objects Instead of Naked Threads"
The Active Object (Actor) pattern inherently supports and automatically enforces the best practices of concurrent programming.

The Active Object pattern is valuable, because it dramatically improves your ability to reason about your thread's code and operation by giving you higher-level abstractions and idioms that raise the semantic level of your program and let you express your intent more directly and safely, thus improving your productivity.

The concept of autonomous software objects communicating by message passing dates back to the 1970s (Carl Hewitt came up with Actors). In the 1990s, methodologies like ROOM adapted actors for real-time computing. More recently, UML has introduced the concept of Active Objects that are essentially synonymous with the ROOM actors. Today, the actor model is all the rage in the enterprise computing. A number of actor programming languages (e.g., Erlang, Scala, D) as well as actor libraries and frameworks (e.g., Akka, Kilim, 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), National Instruments LabVIEW (with LabVIEW Actor Framework), and QP™ frameworks from Quantum Leaps.
Most conventional RTOSes are **not** “event-driven”, because RTOSes are based on **blocking** (while event-driven programming is all about **not**-blocking). Also RTOSes don’t provide the **event abstraction** (event objects or messages carrying event signals and parameters).

→ semaphores or event-flags RTOS primitives are **not** event instances.

But still, you **can** manually implement the Active Object pattern on top of a conventional RTOS by self-imposing the following rules and conventions:

- You define your own basic event data type, which carries the event **signal** and can be extended to carry event **parameters**
- Each thread owns an event queue capable of storing your event objects (could be message queue in RTOS)
- The treads communicate **only** by posting events to their queues
  → **asynchronous** communication without blocking
- Each thread is organized as a “message pump” (queue + event loop)
  → thread blocks **only** when its queue is empty, and does **not block** when processing an event
- All data and resources (e.g., peripherals) are bound to threads and can be accessed **only** from the owner thread (**encapsulation for concurrency**)
A framework is a universal, reusable software architecture for development of specific class of software (e.g., real-time embedded control systems).

The most important characteristics of a framework is that code provided by the application developers is being called by the framework, which results in inversion of control compared to using a toolkit such as a conventional RTOS.

For example, when you use an RTOS, you write the main body of each thread and you call the code from the RTOS (such as a semaphore, time delay, etc.) In contrast, when you use a framework, you reuse the whole architecture and write the code that it calls (inversion of control).

The inversion of control is very characteristic to virtually all event-driven systems. It is the main reason for the architectural-reuse and enforcement of the best practices, as opposed to re-inventing them for each project at hand. It also leads to a much higher conceptual integrity of the final product and dramatic improvement of developer's productivity.
Even though a conventional RTOS can be used to implement event-driven Active Objects, you must be very careful not to use most of the RTOS mechanisms, because they block (e.g., semaphores, delays, etc.).

At the same time, a conventional RTOS does not provide much of support for event-driven programming, which you need to create yourself.

This is all because conventional RTOSes are designed for the sequential programming model, where you block and wait in-line for the occurrence of an event. For example, consider the venerable “Blinky” implementation with delay() functions called to wait in-line.

Event-driven programing represents a paradigm shift, where each event (such as timeout event) is processed to completion and the handler returns to the framework, without blocking.
Another big class of problems in programming real-time and embedded (RTE) systems arises from the difficulties in responding to events, which often leads to convoluted program logic (a.k.a. “spaghetti code”):

- The response depends on both: the event type and the **internal state** of the system
- The internal state (history) of the system is represented *ad hoc* as multitude flags and variables
- Convoluted IF-THEN-ELSE-SWITCH logic to test the flags and variables → **spaghetti code** (a.k.a. BBM = Big Ball of Mud)
- Multitude flags and variables → **inconsistencies**
- Multitude of paths through the code → hard to understand code → hard to test with high cyclomatic complexity
- Fragile code → fear of “breaking the logic” → more flags and variables → **architectural decay**
Finite State Machines—the best known “spaghetti reducers”

- “State” captures only the relevant aspects of the system's history and ignores all irrelevant aspects.

For example, a computer keyboard can be in “default” or “caps_locked” state, where it generates lower-case or upper-case characters. Only pressing CAPS_LOCK toggles between these states. Pressing other keys is irrelevant.

State machines are a natural fit for event-driven programming,
- State machine is exactly designed to process an event quickly and return to the caller
- The context of the system between calls is represented by the single state-variable, much more efficient than in sequential programming, where the context is represented by the whole call stack.
State diagrams (statecharts) should not be confused with flowcharts.

The main difference is that state machines need events to perform any actions and possibly change state (execute transitions).

Flowcharts don't need events. They progress from one stage of processing to another upon completion of processing.

Graphically, flowcharts reverse the sense of nodes and arcs in the diagram. In state machines, processing is associated with arcs. In flowchart with nodes.

The main difference boils down to the different programming paradigms represented:
- Statecharts correspond to event-driven programming paradigm
- Flowcharts correspond to the sequential programming paradigm
A lot of state machine examples in various magazines, books, and online pertain to input-driven (a.k.a. polled) state machines, as opposed to truly event-driven state machines introduced earlier.

Input-driven state machines are NOT driven by events. Instead, an input-driven state machine code is called “as fast as possible”, or “periodically” from while(1) loops to poll for the events. In the code, you can easily recognize such input-driven state machines by the if() statements that check the inputs in each state and only after discovering the right combination of inputs, they execute actions. In the diagrams, you can easily recognize input-driven state machines by the fact that state transitions are NOT labeled by events, but rather by guard conditions. The brackets around those guard conditions, which are required by the UML state machine notation, are often missing, but you typically can recognize that the labels are conditions, especially when you see logic operators, like and/or.

The main problems with input-driven (polled) state machines are that they might miss changes in the inputs (if sampling is too slow) or recognize the changes in different order, depending on the timing (race conditions). They are also wasteful, as they need to run “all the time”. Finally, it is hard to apply hierarchical event processing, because there are no explicit events.
But even the event-driven FSMs have a major shortcoming known as the "state and transition explosion". For example, if you try to represent the behavior of a simple pocket calculator with a traditional FSM, you'll notice that many events (e.g., the Clear or Off button presses) are handled identically in many states. A conventional FSM, has no means of capturing such a commonality and requires repeating the same actions and transitions in many states.

Hierarchical State Machines solve this problem by introducing state nesting with the following semantics: If a system is in the nested state, for example "result" (called the substate), it also (implicitly) is in the surrounding state "on" (called the superstate). This state machine will attempt to handle any event in the context of the substate, which conceptually is at the lower level of the hierarchy. However, if the substate "result" does not prescribe how to handle the event, the event is not quietly discarded as in a traditional "flat" state machine; rather, it is automatically handled at the higher level context of the superstate "on".

State nesting enables substates to reuse the transitions and actions defined already in superstates. The substates need only define the differences from the superstates (programming-by-difference).
This section introduces the QP active object frameworks specifically designed for real-time embedded (RTE) systems, such as single-chip microcontrollers.
QP™ is a family of lightweight real-time embedded frameworks (RTEFs) specifically designed for deeply embedded real-time systems, such as single chip MCUs (8-, 16-, and 32-bit).

The QP family consists of QP/C, QP/C++, and QP-nano frameworks, which are all strictly quality controlled, thoroughly documented, and available in full source code.

The behavior of active objects is specified in QP by means of hierarchical state machines (UML statecharts). The frameworks support manual coding of UML state machines in C or C++ as well as automatic code generation by means of the free QM™ model-based design tool (discussed later).

QP™ frameworks are especially applicable to systems with functional-safety requirements, such as medical devices, defense, aerospace, industrial control, robotics, IoT, transportation, automotive, etc.
QP™ frameworks are developed under the increasingly popular, strictly quality-controlled, **professional open source** business model that combines the best of the **open source** and proprietary software worlds to make open source a **safe choice** for the embedded systems vendors. This includes the accountability for the licensed intellectual property, professional documentation and technical support expected of a traditional software vendor as well as transparent development, availability of source code and active community inherent in open source projects.

QP™ RTEFs address high-reliability applications across a wide variety of markets. In each of these application areas, the modern QP™ software architecture has distinct advantages.
NOTE: All QP™ frameworks are fundamentally object-oriented, which means that the frameworks themselves and your applications derived from the frameworks are fundamentally composed of classes and only classes can have state machines associated with them.

The QP/C and QP/C++ frameworks have very similar features, although QP/C++ supports directly the C++ object model, while QP/C emulates it with design patterns and coding conventions.

The QP-nano framework has significantly reduced feature set, specifically designed for low-end 8-bit CPUs with very limited RAM.

The general guidelines for choosing the QP framework are as follows:

- 8-bit CPU and/or total RAM < 1KB → QP-nano
- 16- or 32-bit CPU and total RAM > 1KB → QP/C or QP/C++
In the resource-constrained embedded systems, the biggest concern has always been about the size and efficiency of Active Object (Actor) 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.

However, it turns out that a real-time embedded framework (RTEF) can be actually smaller than a traditional RTOS. This is possible, because active objects don't need to block internally, so most blocking mechanisms (e.g., semaphores) of a conventional RTOS are not needed.

For example, the diagram shows the RAM/ROM sizes of the QP/C, QP/C++, and QP-nano RTEFs versus a number of conventional (RT)OSes. The diagram shows the total system size as opposed to just the RTOS/OS footprints. As you can see, when compared to conventional RTOSes, QP™ frameworks require significantly less RAM (the most precious resource in single-chip MCUs).

All these characteristics make event-driven RTEF is 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.
The QP™ RTEFs have a layered structure:
- The Target hardware sits at the bottom.
- The Board Support Package (BSP) above it provides access to the board-specific features, such as the peripherals.
- The real-time kernel (QV, QK, QXK, or a conventional 3rd-party RTOS) provides the foundation for multitasking, such as task scheduling, context-switching, and inter-task communication.
- The event-driven framework (QF) supplies the event-driven infrastructure for executing active objects and ensuring thread-safe event-driven exchanges among them.
- The event-processor (QEP) implements the hierarchical state machine semantics (based on UML statecharts). The top layer is the application-level code consisting of loosely-coupled active objects.
- QS is software tracing system that enables developers to monitor live event-driven QP™ applications with minimal target system resources and without stopping or significantly slowing down the code. QS is an ideal tool for testing, troubleshooting, and optimizing QP™ applications. QS can even be used to support acceptance testing in product manufacturing.
The package and class structure reflects the layered architecture. The QP framework provides a few base classes to be subclassed and specialized in the applications. This is a very common approach characteristic of most frameworks. The framework also uses the underlying kernel or RTOS for basic multitasking and event-queuing services.

The most important base classes provided by the framework are:
- **QHsm** base class for deriving application-specific HSMs.
- **QActive** base class for deriving application-specific Active Objects
- **QEvt** base class for deriving application-specific events with parameters or to be used directly for events without parameters.
- **QTimeEvt** class to be used “as is” for time events or to be further subclassed into application-specific time events.
The QEP event processor provides efficient implementation for hierarchical state machines and enables developers to code hierarchical state machines in an intuitive, straightforward way, where each state machine elements maps to code precisely, unambiguously, and exactly once (traceability between code and design).

QEP supports the following state machine concepts:
- Hierarchical state nesting
- Entry/exit actions in states
- Regular transitions
- Internal transitions
- Nested initial transitions
- Guard conditions on all transition types
The main job of an RTEF is to provide execution environment (thread) to each active object and to provide thread-safe, asynchronous mechanisms to exchange events.

The QF real-time framework serves as a “software bus” to connect active objects. The framework supports direct event posting as well as publish-subscribe event exchange.
The QP/C and QP/C++ frameworks support “zero copy” event delivery for exchanging events with arbitrary parameters, which works as follows:

1. Application allocates an event from one of the fixed-size event-pools
2. Application posts or publishes the event to active objects
3. AOs process the event, whereas they can re-post or re-publish it
4. QP automatically detects if the event is still in use and recycles it if isn't
5. QP supports immutable “static events” as an optimization. Such static events don't need to be dynamically allocated and recycled.
The QP™ framework provides an assortment of real-time kernels that the developers can choose to execute their active objects.

The simplest and most efficient is the cooperative QV (“Vanilla”) kernel, which operates as follows:

The kernel runs in a single main loop, which constantly polls the event queues of all active objects. The kernel always selects the highest-priority, not-empty event queue. Every event is always processed to completion in the main loop. If any new events are produced during the RTC step (e.g., by ISRs or by actions in the currently running active object) they are just queued, but the current RTC step is not preempted. The kernel very easily detects a situation where all event queues are empty, in which case it invokes the idle callback, where the application can put the CPU into a low-level sleep mode (power-efficient kernel).

The task-level response of this kernel is the longest RTC step in the whole system, but without blocking the RTC steps are naturally very short. Therefore the QV kernel is adequate to many systems, including safety-critical systems.
The QP™ frameworks also contain a very efficient, preemptive, priority-based, run-to-completion kernel called QK. This kernel does not allow threads to block in the middle of run-to-completion step, but allows them to preempt each other (such threads are classified as “basic threads” in the OSEK/VDX terminology). The non-blocking limitation is irrelevant for event-driven active objects, where blocking is not needed anyway.

The threads in the QK kernel operate a lot like interrupts with a prioritized interrupt controller, except that the priority management happens in software (with up to 64 priority levels). The limitation of not-blocking allows the QK kernel to nest all threads on the single stack, the same way as all prioritized interrupts nest on the same stack. This use of the natural stack protocol of the CPU makes the QK kernel very efficient and requires much less stack space than traditional blocking kernels.

Still, the QK kernel meets all the requirements of the Rate Monotonic Analysis and can be used to in hard real-time systems.
Finally, the QP/C and QP/C++ frameworks contain a traditional preemptive, blocking kernel called QXK. QXK allows threads to block anywhere in the code, so it works just like most traditional blocking kernel.

The main purpose of QXK is to allow sequential code (middleware or legacy code) to coexist with event-driven active objects, without a need for any 3rd-party RTOS kernel.

QXK provides typical blocking mechanisms, such as semaphores, mutexes, and time delays. Such primitives are typically expected by various middleware libraries (TCP/IP stacks, File systems, USB libraries, etc.)

The main advantage of QXK is that is integrates very tightly with QP™ and reuses most of the common facilities, such as event queues. QXK is currently available for all ARM Cortex-M cores (M0/M0+/M3/M4/M4F/M7).
Testing, debugging, and fine-tuning of embedded software often takes more calendar time than design and coding combined. The biggest problem is the limited visibility into the deeply embedded system.

Software tracing is a method for obtaining diagnostic information in a live environment without the need to stop or significantly slow-down the code.

Software tracing is especially effective in event-driven systems, where all important system interactions funnel through the active object framework and the state machine event processor.

QP/C and QP/C++ frameworks contain QS/QSPY software tracing system that is an ideal tool for testing, troubleshooting, and optimizing QP™ applications. QS can even be used to support acceptance testing in product manufacturing.
QUTest™ (pronounced 'cutest') is a unit testing harness (a.k.a. unit testing framework), which is specifically designed for deeply embedded systems, but also supports unit testing of embedded code on host computers ("dual targeting"). QUTest™ is the fundamental tooling for Test-Driven Development (TDD) of QP/C/C++ applications, which is a highly recommended best-practice.

In a nutshell, working with QUTest™ is similar to "debugging with printf", where you instrument the code with the printf statements (or sprintf or similar). You then run the code with a controlled set of inputs, and examine the produced output from the printfs to determine whether the code under test operates correctly. The main differences from using printfs are: (1) that the much more efficient QP/Spy is used instead and (2) that both generating the inputs and the checking of the test outputs are automated.

QUTest™ separates the execution of the CUT (Code Under Test) from checking of the "test assertions". The embedded target is concerned only with running a test fixture that exercises the CUT and produces QP/Spy™ trace, but it does not check the "test assertions". Checking the "test assertions" against the expectations is performed on the Host computer by means of test scripts (Python and Tcl are supported).
QSpyView™ is a tool for remote monitoring and control of embedded devices. QSpyView™ leverages the QP/Spy software tracing system to provide:

- Customizable (scripted) remote User Interface
- Graphically display information about the life, running Target
- Dynamically interact with the running Target
- Remotely reset of the Target
Design by Contract™ (DbC) is a philosophy that views a software system as a set of components whose collaboration is based on precisely defined specifications of mutual obligations — the contracts. The central idea of this method is to inherently embed the contracts in the code and validate them automatically at runtime.

In C and C++, the most important aspects of DbC (the contracts) can be implemented with assertions. Assertions are increasingly popular among the developers of mission-critical software. For example, NASA requires certain density of assertions in such software.

In the context of active object frameworks, such as QP™, DbC provides an excellent methodology for implementing a very robust, redundancy layer for monitoring error-free operation. Due to inversion of control so typical in all event-driven systems, an active object framework controls many more aspects of the application than a traditional (Real-Time) Operating System. Such a framework is in a much better position to ensure that the application is performing correctly, rather than the application to check error codes or catch exceptions originating from the framework.
This section introduces the QM graphical modeling tool for the QP frameworks.
QM™ (QP™ Modeler) is a freeware, graphical modeling tool for designing and implementing real-time embedded applications based on the QP™ frameworks and hierarchical state machines (UML statecharts). QM™ is available for Windows 64-bit, Linux 64-bit, and Mac OS X 64-bit.

QM™ and QP™ beautifully complement each other:
- QM™ provides a diagram editor for building models of the QP™ applications, to take advantage of the very expressive visual representation of HSMs as state diagrams (UML statecharts)
- QP™ frameworks provide an excellent target for code generation

The main goals of the QM™ modeling tool are:
- to help you break down your software into active objects;
- to help you graphically design the hierarchical state machines associated with these active objects, and
- to automatically generate code that is of production-quality and is fully traceable from your design.
Compared to most other "high ceremony" modeling tools on the market today, QM™ is much simpler, code-centric, and relatively low-level.

This characterization is not pejorative. It simply means that QM™ maps the design unambiguously and directly to C or C++ code, without intermediate layers of "Platform-Independent Models" (PIMs), "Platform-Specific Models" (PSMs), complex "Model-Transformations", or "Action Languages". All actions executed by state machines are entered into the model directly in C or C++.
As most modeling tools, QM™ allows you to capture the logical structure of your application in terms of packages, classes, and state machines. The tool provides several views of the abstract model, such as the hierarchical tree-like Model Explorer, the Diagrams, and Property Sheets associated with the selected model element.

A lot of thought went into drawing hierarchical state diagrams in QM™. In this respect, the tool is innovative and might work differently than other graphical state machine tools on the market. For example, QM does not use "pseudostates", such as the initial pseudostate or choice point. Instead QM uses higher-level primitives of initial-transition and choice-segment, respectively. This simplifies state diagramming immensely, because you don't need to separately position pseudostates and then connect them. Also, QM introduces a new notation for internal transitions, which allows actual drawing of internal transitions (in standard UML notation internal transitions are just text in the state body). This notation enables you to attach internal transitions and/or regular state transitions to a single choice point—something that comes up very often in practice and was never addressed well in the standard UML.
QM™ is a unique model-based design tool on the market that allows you to capture the **physical design** of your code as an integral part of the model, whereas "physical design" is the partitioning of the code into directories and files, such as header files (.h) and implementation files (.c or .cpp files).

This unique approach gives you the ultimate flexibility in the source code structure and mitigates the needs to make manual changes to the generated code in order to combine it with hand-written code or existing 3rd-party code.

Also, QM™ provides mechanisms to quickly go back and forth between the model and the generated code so that any changes can be conveniently applied directly to the model rather than the code.
QM™ provides a mechanism to extend the tool with external commands, which can be executed *directly* from QM™.

Examples of external tools include make to perform a software build, lint to statically check your code, your own unit tests that run on the host computer, or a command-line tool to download the code to the embedded target, directly from QM™ with just one key press. The output generated by the external tool will appear in the QM's Log Console.
Welcome to the 21st Century!

- Experts avoid shared-state concurrency and blocking
- Experts use the event-driven Active Object design pattern
- Experts use hierarchical state machines instead of “spaghetti code”
- Event-driven active objects and state machines require a paradigm shift from sequential to event-driven programming
- QP™ real-time embedded frameworks (RTEFs) provide a lightweight, reusable architecture based on the AO pattern and hierarchical state machines for deeply embedded systems, such as single-chip MCUs
- QM™ model-based design tool eliminates manual coding of HSMs