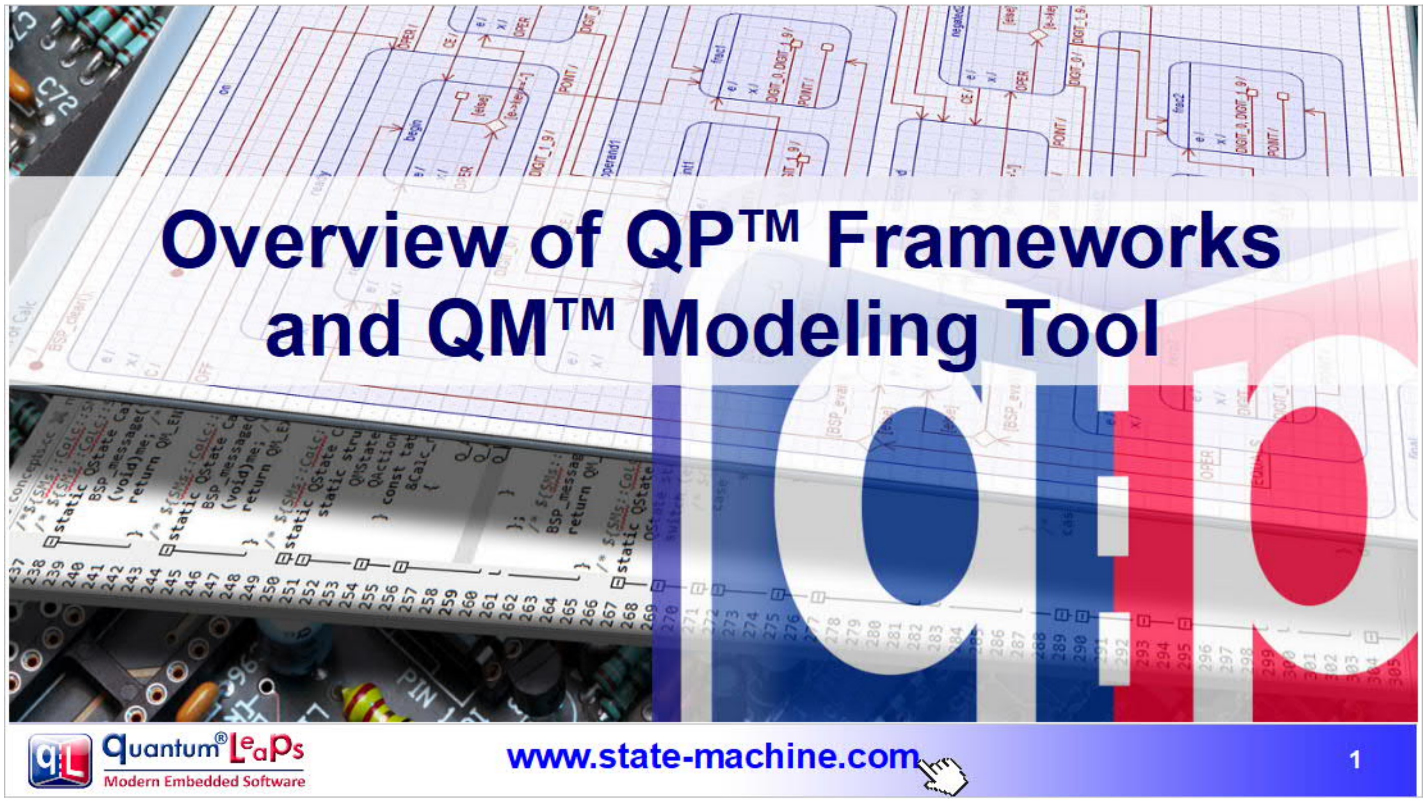


Overview of QP™ Frameworks and QM™ Modeling Tool

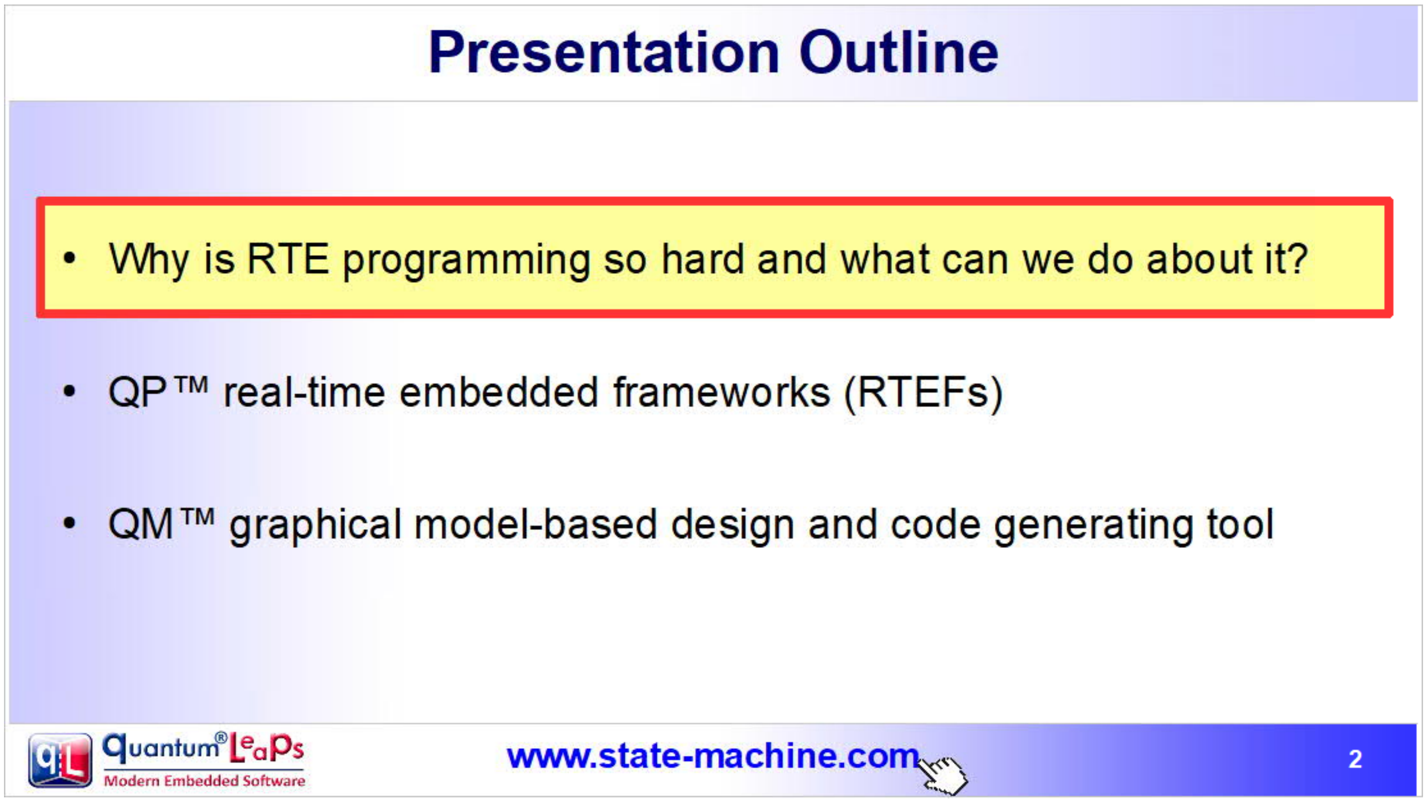
Quantum® LeaPs
Modern Embedded Software

www.state-machine.com

1

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*, truly event-driven and at a higher level of abstraction 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 tool provide such a modern, reusable architecture based on active objects (actors), hierarchical state machines, software tracing, graphical modeling, and automatic code generation.



Presentation Outline

- Why is RTE programming so hard and what can we do about it?
- QP™ real-time embedded frameworks (RTEFs)
- QM™ graphical model-based design and code generating tool

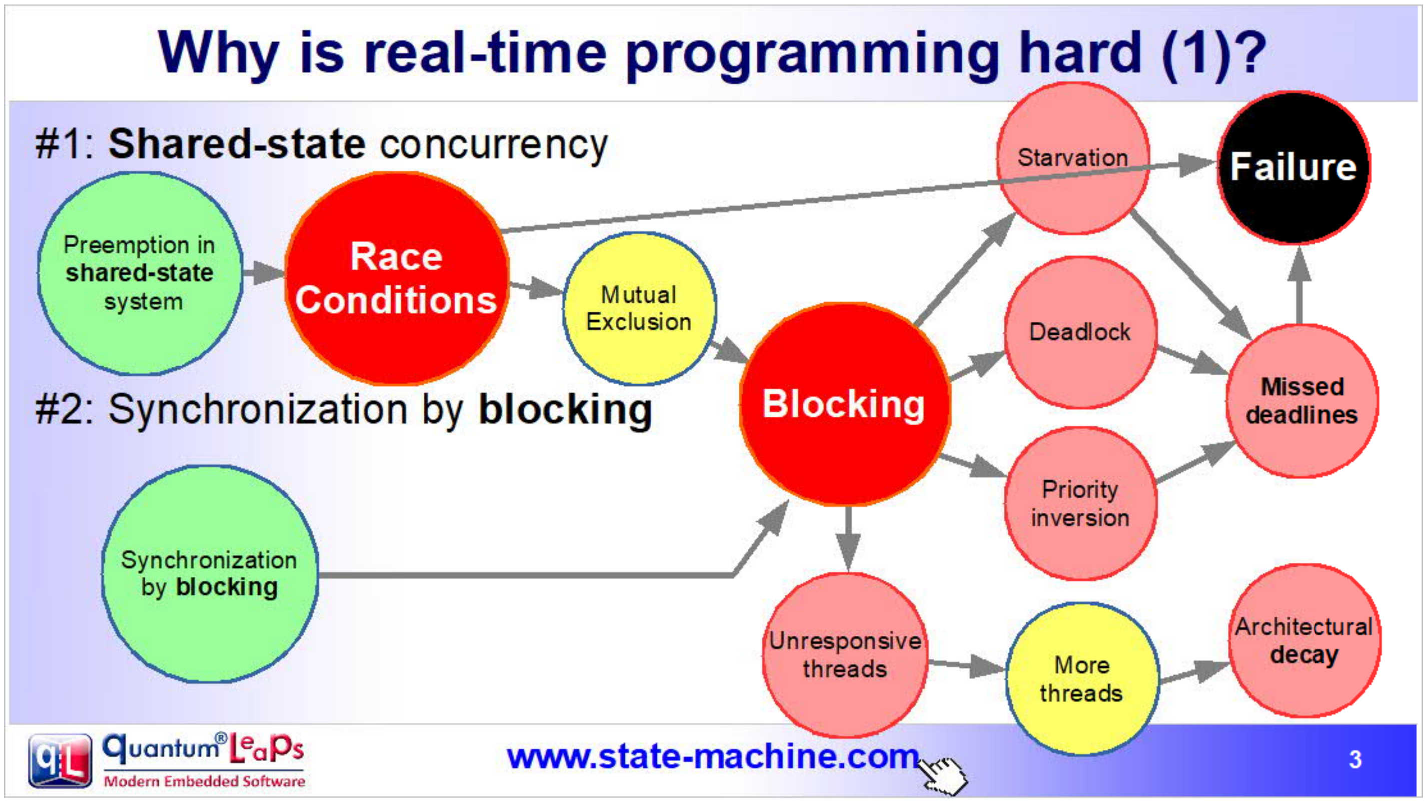
Quantum® LeaPs Modern Embedded Software

www.state-machine.com

2

You can't just look at the QP™ real-time 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 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.



Why is real-time programming hard (1)?

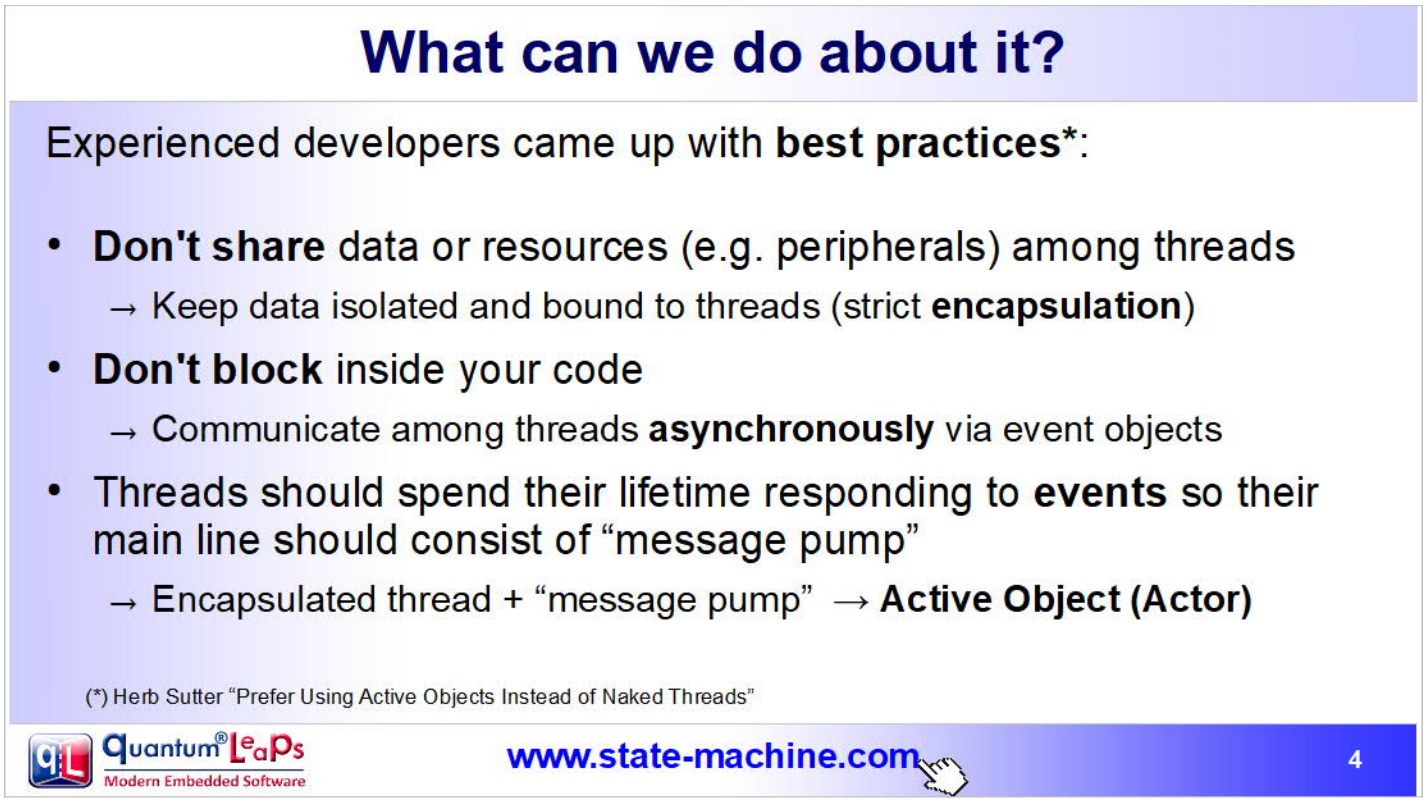
Some of the most difficult problems with Real-Time Embedded (RTE) programming are related to **concurrent code execution**
→ these problems are usually intermittent, subtle, hard-to-reproduce, hard-to-isolate, hard-to-debug, and hard-to-remove
→ they pose the highest risk to the project schedule

#1 **Shared-state** concurrency problems due to preemption:

- Endemic to all **shared-state** systems (main+ISRs 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!**


What can we do about it?

Experienced developers came up with **best practices***:

- **Don't share** data or resources (e.g. peripherals) among threads
  → Keep data isolated and bound to threads (strict **encapsulation**)
- **Don't block** inside your code
  → Communicate among threads **asynchronously** via event objects
- Threads should spend their lifetime responding to **events** so their main line should consist of "message pump"
  → Encapsulated thread + "message pump" → **Active Object (Actor)**

(*) Herb Sutter "Prefer Using Active Objects Instead of Naked Threads"

Quantum® LeaPs Modern Embedded Software www.state-machine.com 4

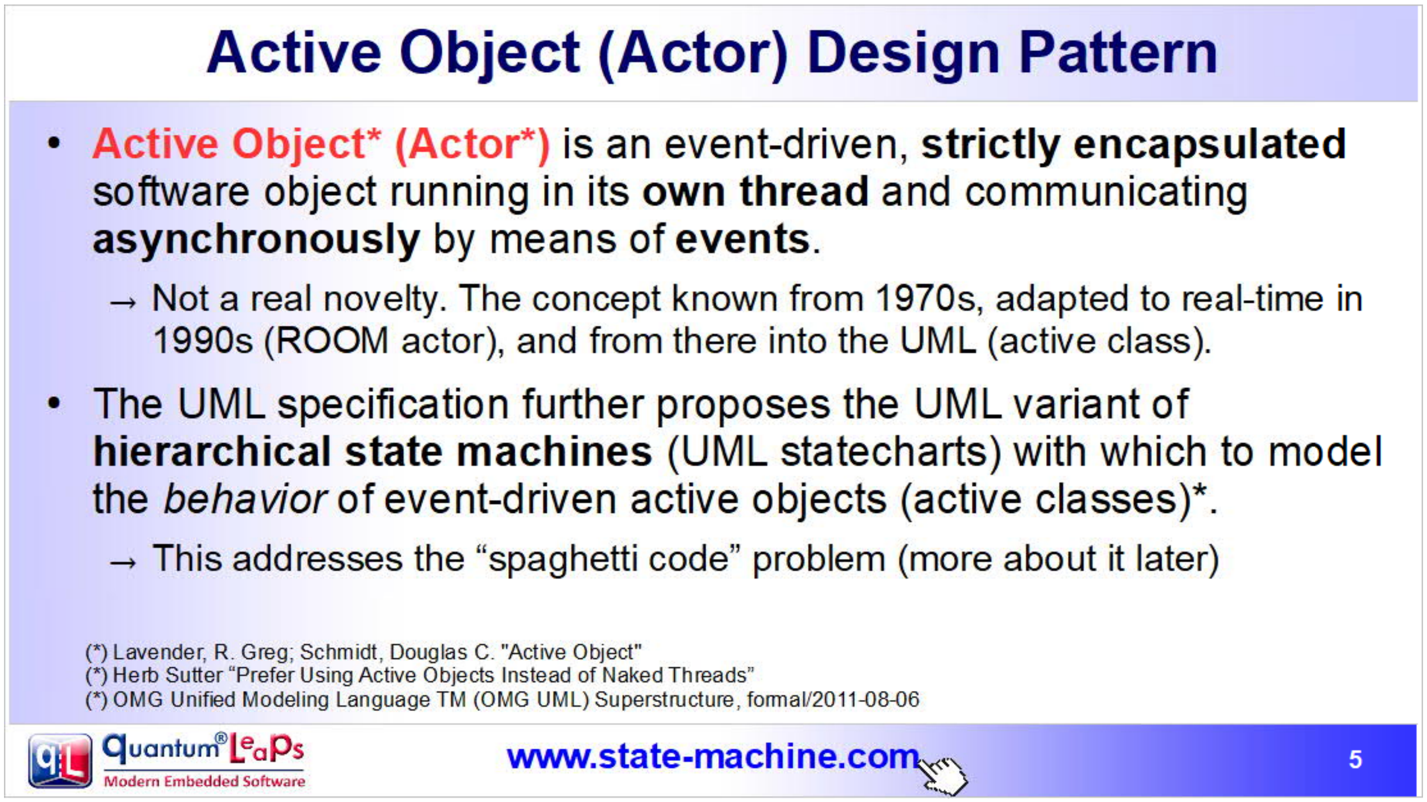
Experts in the field have learned to avoid shared-state concurrency and to avoid blocking to synchronize their threads. Instead, experts apply the following **best practices** of concurrent programming:

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**)



Active Object (Actor) Design Pattern

- **Active Object* (Actor*)** is an event-driven, **strictly encapsulated** software object running in its **own thread** and communicating **asynchronously** by means of **events**.
  - → Not a real novelty. The concept known from 1970s, adapted to real-time in 1990s (ROOM actor), and from there into the UML (active class).
- 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 classes)*.
  - → This addresses the "spaghetti code" problem (more about it later)

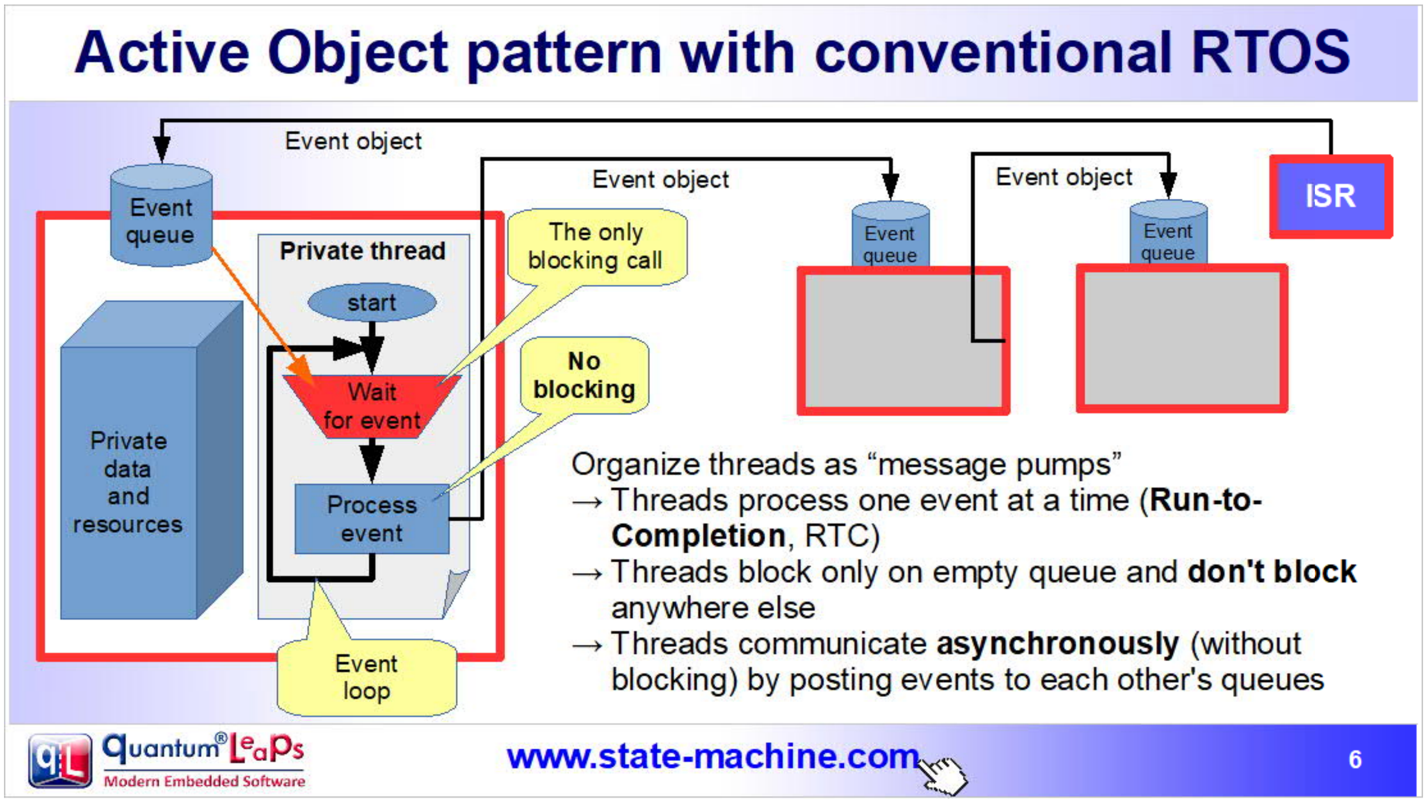
(*) Lavender, R. Greg; Schmidt, Douglas C. "Active Object"
(*) Herb Sutter "Prefer Using Active Objects Instead of Naked Threads"
(*) OMG Unified Modeling Language TM (OMG UML) Superstructure, formal/2011-08-06

Quantum® LeaPs Modern Embedded Software www.state-machine.com 5

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.



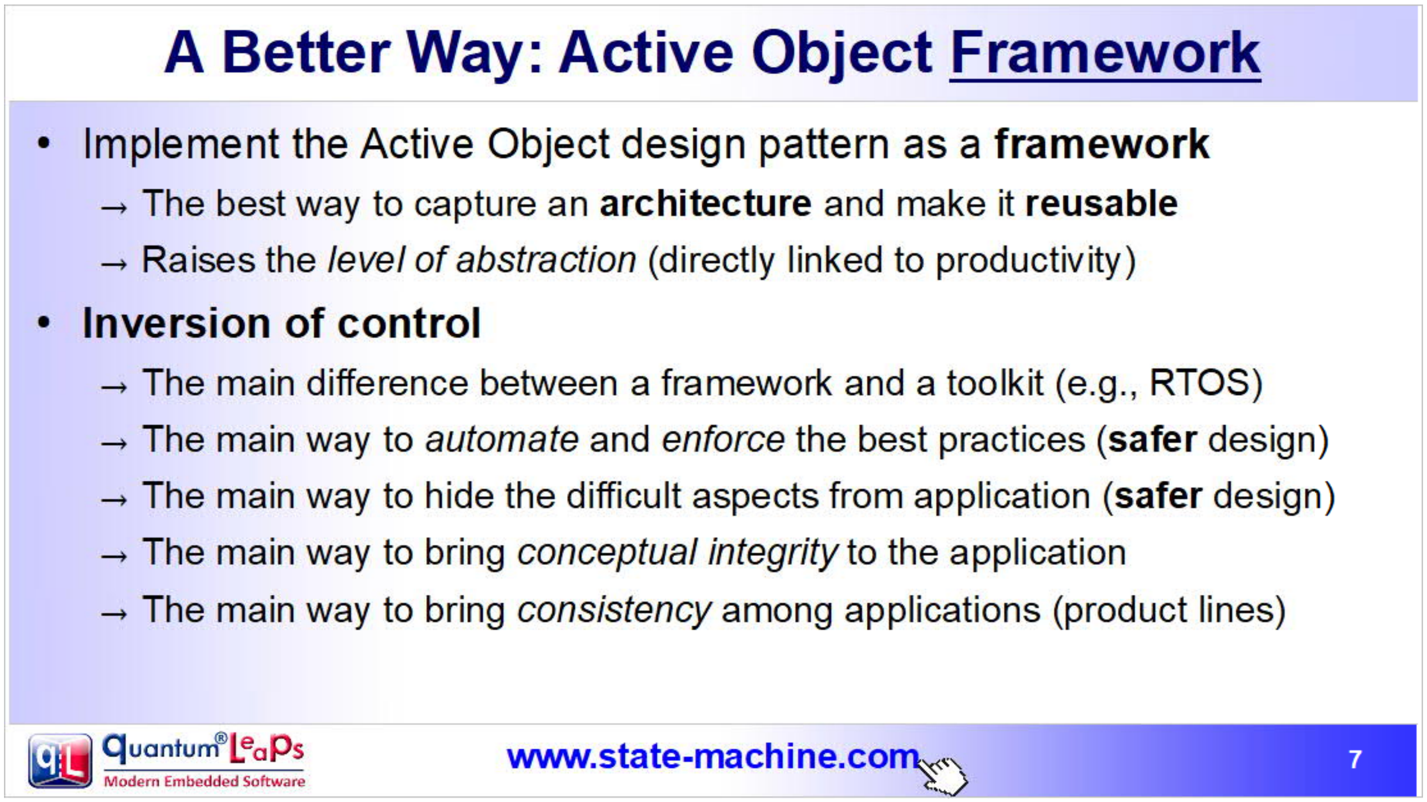
Active Object pattern with conventional RTOS

Organize threads as “message pumps”
→ Threads process one event at a time (**Run-to-Completion**, RTC)
→ Threads block only on empty queue and **don't block** anywhere else
→ Threads communicate **asynchronously** (without blocking) by posting events to each other's queues

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 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 Better Way: Active Object Framework

- Implement the Active Object design pattern as a **framework**
  - → The best way to capture an **architecture** and make it **reusable**
  - → Raises the *level of abstraction* (directly linked to productivity)
- **Inversion of control**
  - → The main difference between a framework and a toolkit (e.g., RTOS)
  - → The main way to *automate* and *enforce* the best practices (**safer** design)
  - → The main way to hide the difficult aspects from application (**safer** design)
  - → The main way to bring *conceptual integrity* to the application
  - → The main way to bring *consistency* among applications (product lines)

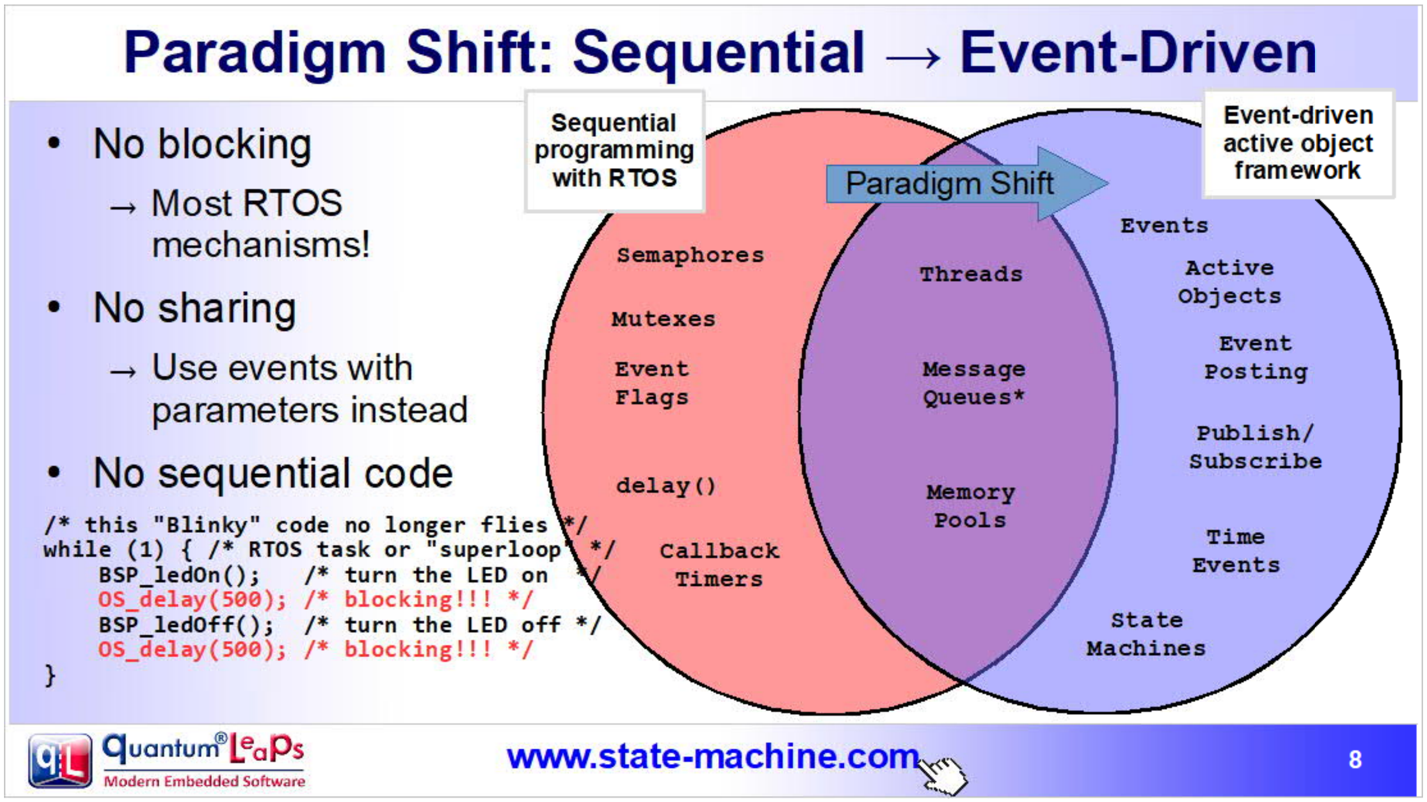
Quantum® LeaPs Modern Embedded Software www.state-machine.com 7

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*.



Paradigm Shift: Sequential → Event-Driven

- No blocking
  → Most RTOS mechanisms!
- No sharing
  → Use events with parameters instead
- No sequential code

```
/* this "Blinky" code no longer flies */
while (1) { /* RTOS task or "superloop" */
    BSP_ledOn();    /* turn the LED on  */
    OS_delay(500); /* blocking!!! */
    BSP_ledOff();   /* turn the LED off */
    OS_delay(500); /* blocking!!! */
}
```

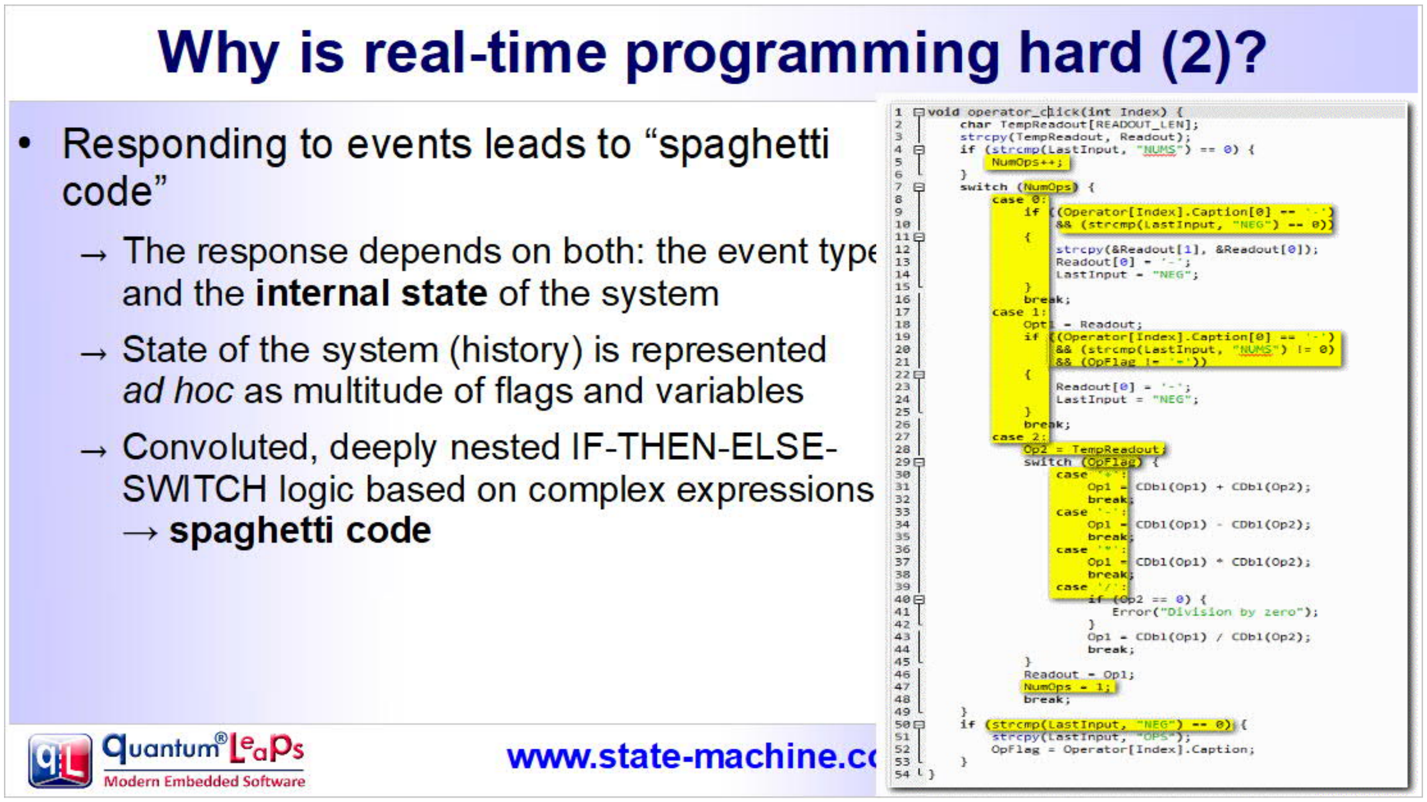
Quantum® Leaps Modern Embedded Software   www.state-machine.com   8

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**.



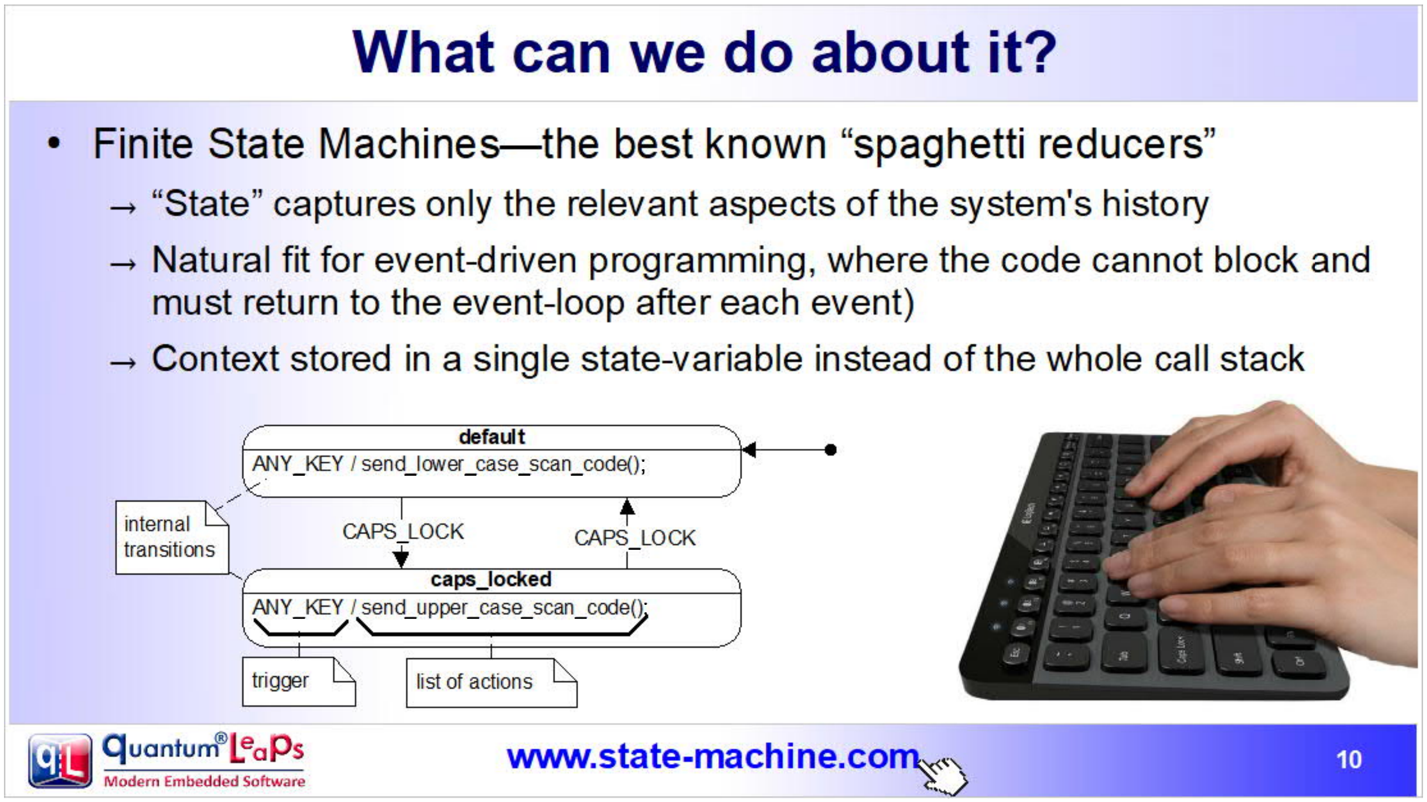
Why is real-time programming hard (2)?

- Responding to events leads to “spaghetti code”
  - → The response depends on both: the event type and the **internal state** of the system
  - → State of the system (history) is represented *ad hoc* as multitude of flags and variables
  - → Convoluted, deeply nested IF-THEN-ELSE-SWITCH logic based on complex expressions → **spaghetti code**

```
void operator_click(int Index) {
    char TempReadout[READOUT_LEN];
    strcpy(TempReadout, Readout);
    if (strcmp(LastInput, "NUMS") == 0) {
        NumOps++;
    }
    switch (NumOps) {
        case 0:
            if ((Operator[Index].Caption[0] == '-')
              && (strcmp(LastInput, "NEG") == 0))
            {
              strcpy(&Readout[1], &Readout[0]);
              Readout[0] = '-';
              LastInput = "NEG";
            }
            break;
        case 1:
            Op1 = Readout;
            if ((Operator[Index].Caption[0] == '-')
              && (strcmp(LastInput, "NUMS") != 0)
              && (OpFlag != '='))
            {
              Readout[0] = '-';
              LastInput = "NEG";
            }
            break;
        case 2:
            Op2 = TempReadout;
            switch (OpFlag) {
              case '+':
                  Op1 = CDbl(Op1) + CDbl(Op2);
                  break;
              case '-':
                  Op1 = CDbl(Op1) - CDbl(Op2);
                  break;
              case '*':
                  Op1 = CDbl(Op1) * CDbl(Op2);
                  break;
              case '/':
                  if (Op2 == 0) {
                     Error("Division by zero");
                  }
                  Op1 = CDbl(Op1) / CDbl(Op2);
                  break;
            }
            Readout = Op1;
            NumOps = 1;
            break;
    }
    if (strcmp(LastInput, "NEG") == 0) {
        strcpy(LastInput, "OPS");
        OpFlag = Operator[Index].Caption;
    }
}
```

Another big class of problems in programming Real-Time Embedded (RTE) 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-SWICH 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**



What can we do about it?

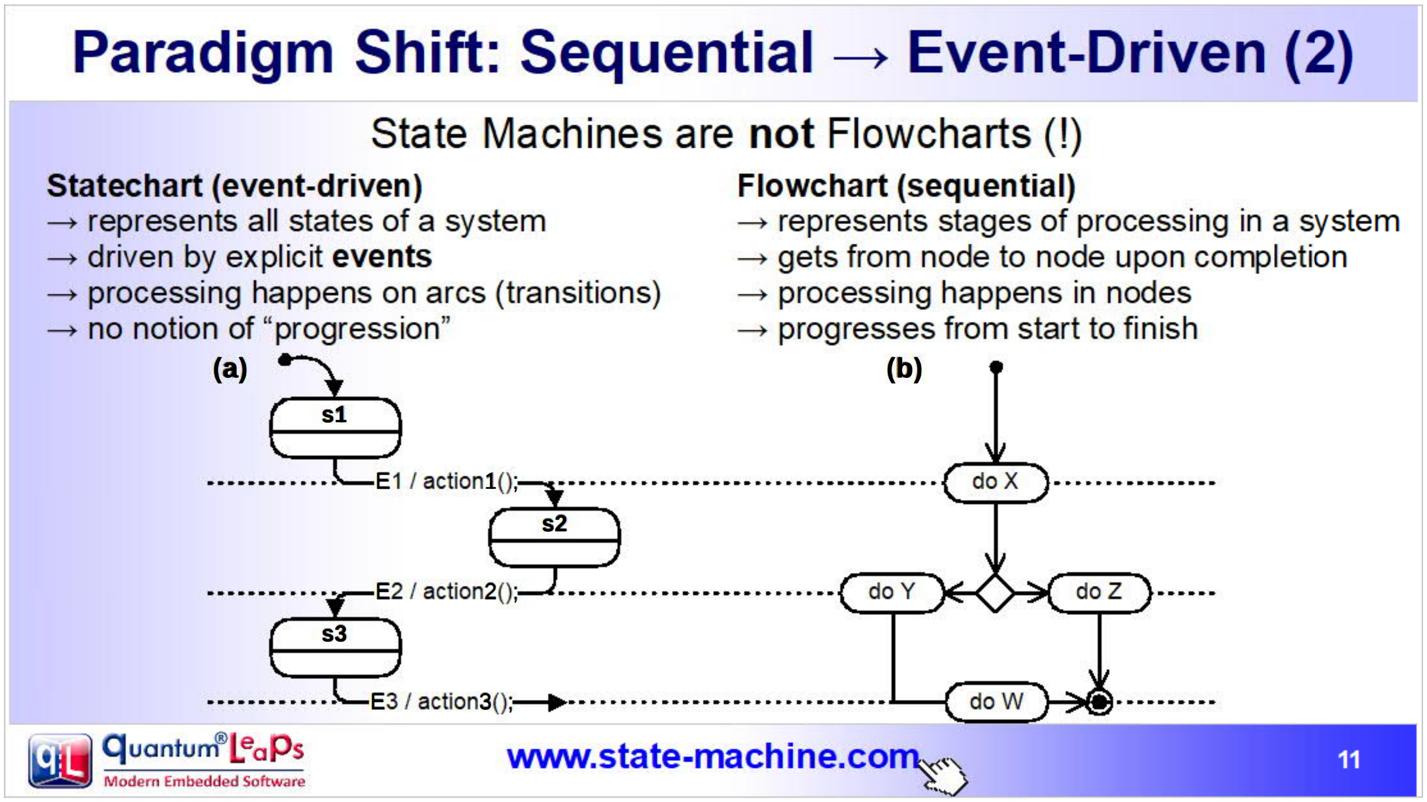
- Finite State Machines—the best known "spaghetti reducers"
  - → "State" captures only the relevant aspects of the system's history
  - → Natural fit for event-driven programming, where the code cannot block and must return to the event-loop after each event)
  - → Context stored in a single state-variable instead of the whole call stack

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*.



Paradigm Shift: Sequential → Event-Driven (2)

State Machines are **not** Flowcharts (!)

Statechart (event-driven)
→ represents all states of a system
→ driven by explicit **events**
→ processing happens on arcs (transitions)
→ no notion of "progression"

Flowchart (sequential)
→ represents stages of processing in a system
→ gets from node to node upon completion
→ processing happens in nodes
→ progresses from start to finish

www.state-machine.com

11

State diagrams (statecharts) should **not** be confused with flowcharts

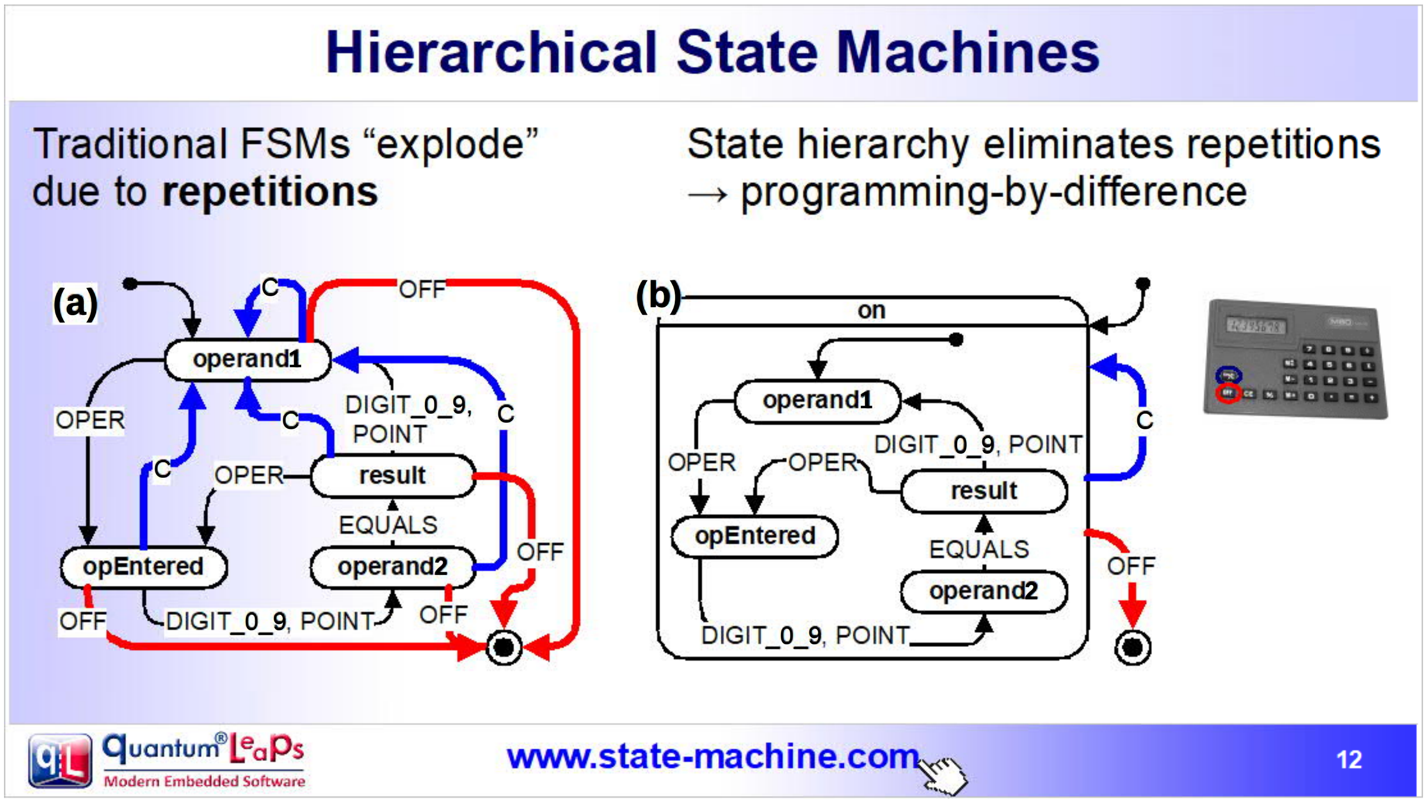
The main difference is that state machines need **events** to perform any actions and possibly change state (execute transitions).

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



Hierarchical State Machines

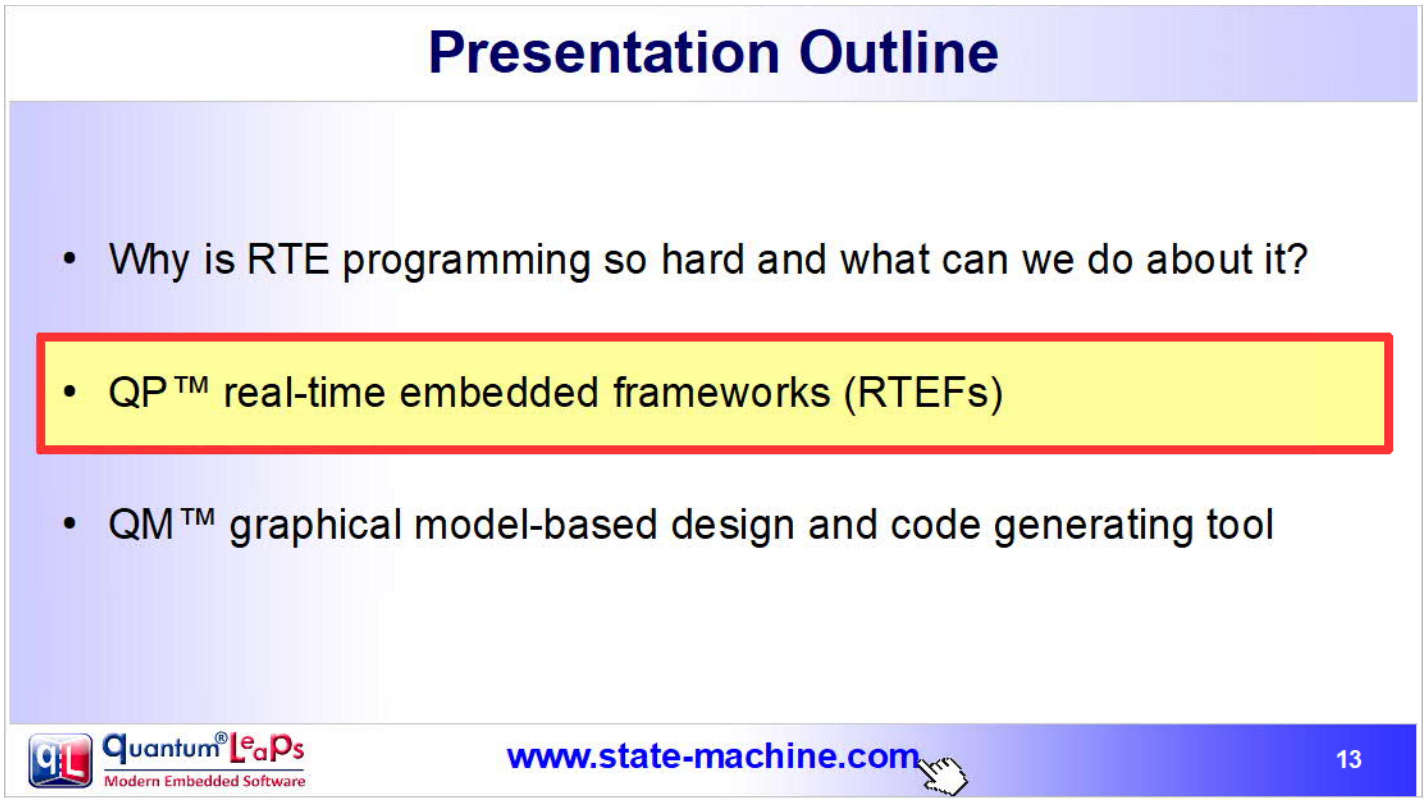
Traditional FSMs "explode" due to **repetitions**

State hierarchy eliminates repetitions → programming-by-difference

Traditional FSMs have a major shortcoming known as "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).

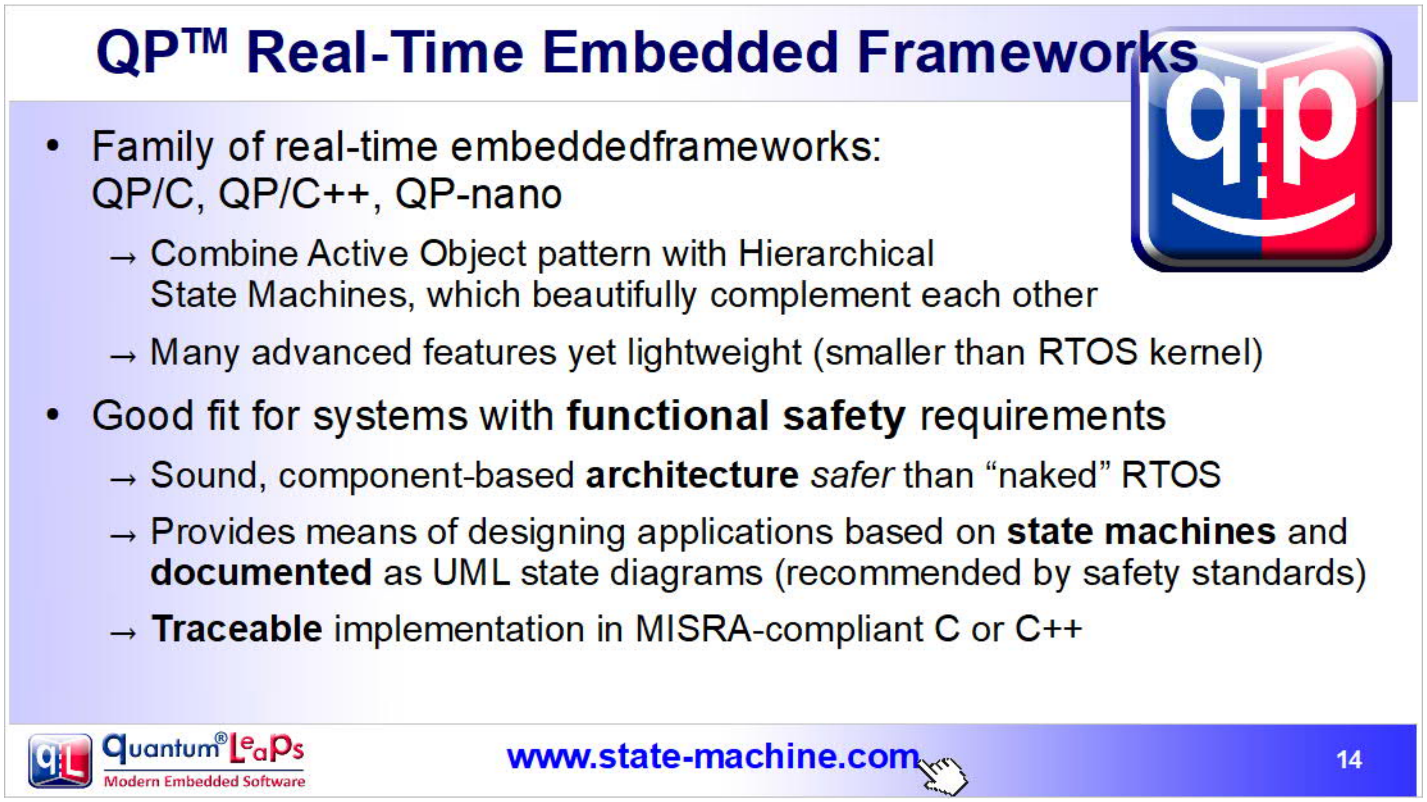


Presentation Outline

- Why is RTE programming so hard and what can we do about it?
- QP™ real-time embedded frameworks (RTEFs)
- QM™ graphical model-based design and code generating tool

Quantum® LeaPs
Modern Embedded Software
www.state-machine.com
13

This section introduces the QP real-time embedded frameworks (RTEFs) specifically designed for real-time embedded (RTE) systems, such as single-chip microcontrollers.



QP™ Real-Time Embedded Frameworks

- Family of real-time embeddedframeworks: QP/C, QP/C++, QP-nano
  - → Combine Active Object pattern with Hierarchical State Machines, which beautifully complement each other
  - → Many advanced features yet lightweight (smaller than RTOS kernel)
- Good fit for systems with **functional safety** requirements
  - → Sound, component-based **architecture** *safer* than "naked" RTOS
  - → Provides means of designing applications based on **state machines** and **documented** as UML state diagrams (recommended by safety standards)
  - → **Traceable** implementation in MISRA-compliant C or C++

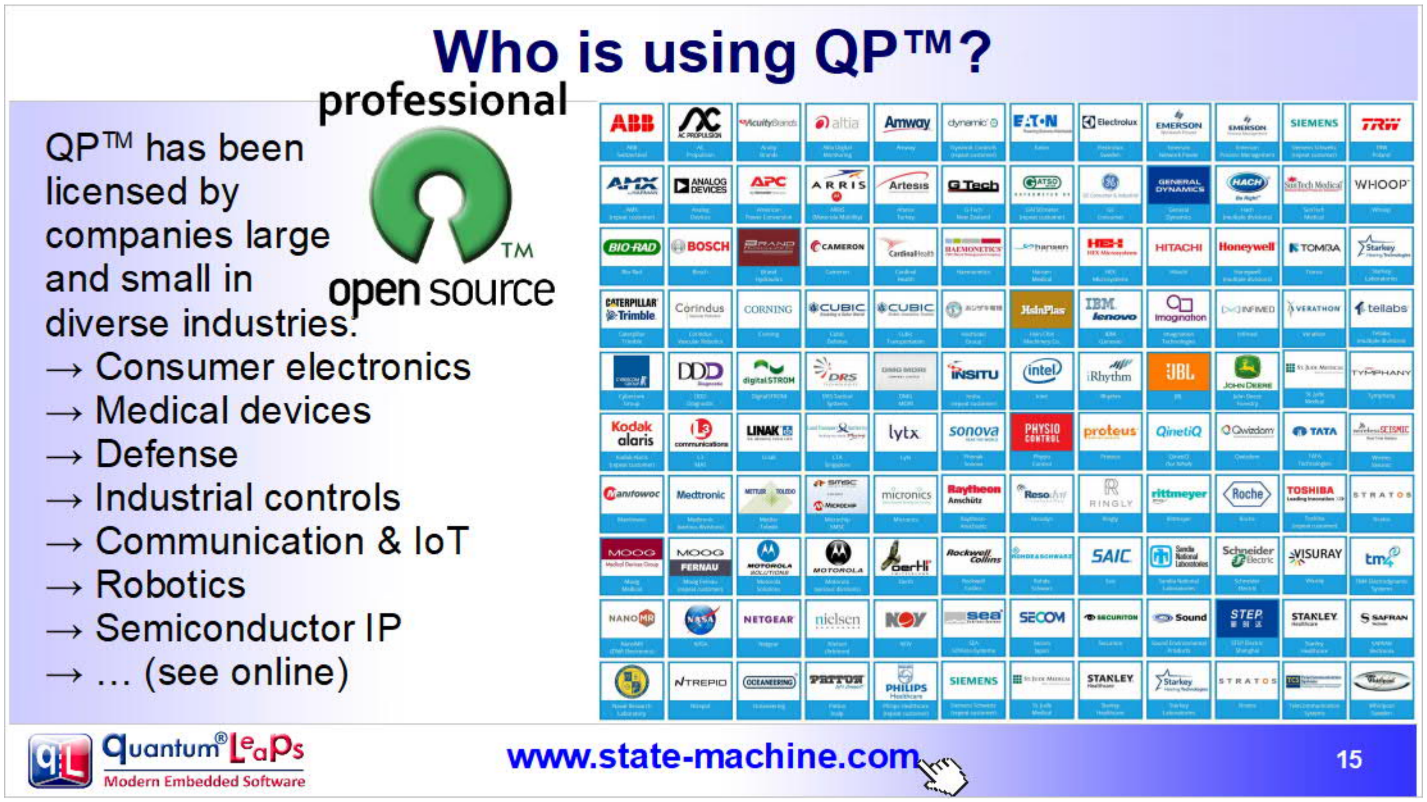
Quantum® LeaPs Modern Embedded Software   www.state-machine.com   14

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™ modeling tool (discussed later).

QP™ frameworks are especially applicable to systems with functional-safety requirements, such as medical devices, defense, aerospace, industrial control, robotics, transportation, automotive, etc.



Who is using QP™?

QP™ has been licensed by companies large and small in diverse industries:

→ Consumer electronics
→ Medical devices
→ Defense
→ Industrial controls
→ Communication & IoT
→ Robotics
→ Semiconductor IP
→ … (see online)

professional
open source
TM

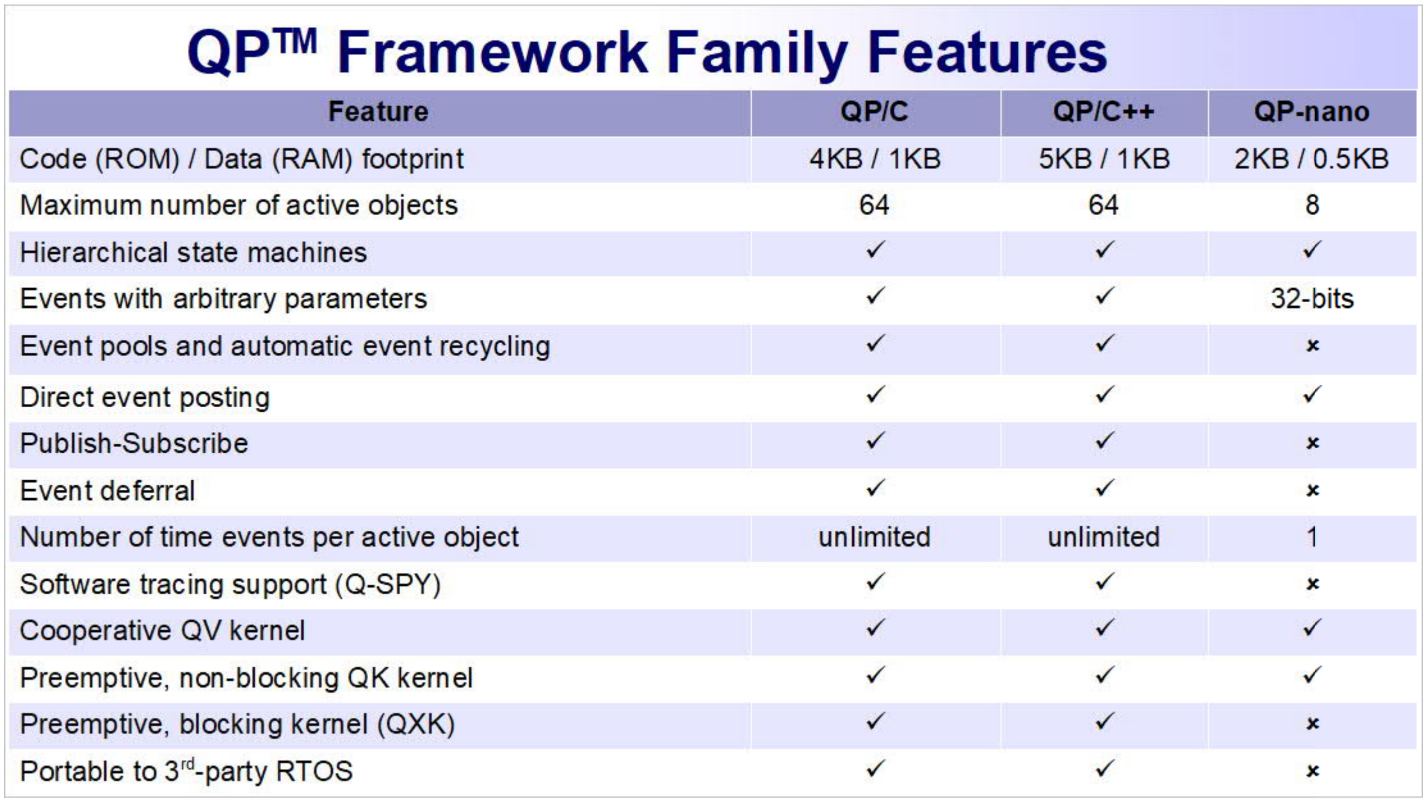
Quantum® LeaPs
Modern Embedded Software

www.state-machine.com

15

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 elegant QP™ software architecture and modern design philosophy have distinct advantages.



QP™ Framework Family Features

Feature	QP/C	QP/C++	QP-nano
Code (ROM) / Data (RAM) footprint	4KB / 1KB	5KB / 1KB	2KB / 0.5KB
Maximum number of active objects	64	64	8
Hierarchical state machines	✓	✓	✓
Events with arbitrary parameters	✓	✓	32-bits
Event pools and automatic event recycling	✓	✓	×
Direct event posting	✓	✓	✓
Publish-Subscribe	✓	✓	×
Event deferral	✓	✓	×
Number of time events per active object	unlimited	unlimited	1
Software tracing support (Q-SPY)	✓	✓	×
Cooperative QV kernel	✓	✓	✓
Preemptive, non-blocking QK kernel	✓	✓	✓
Preemptive, blocking kernel (QXK)	✓	✓	×
Portable to 3rd-party RTOS	✓	✓	×

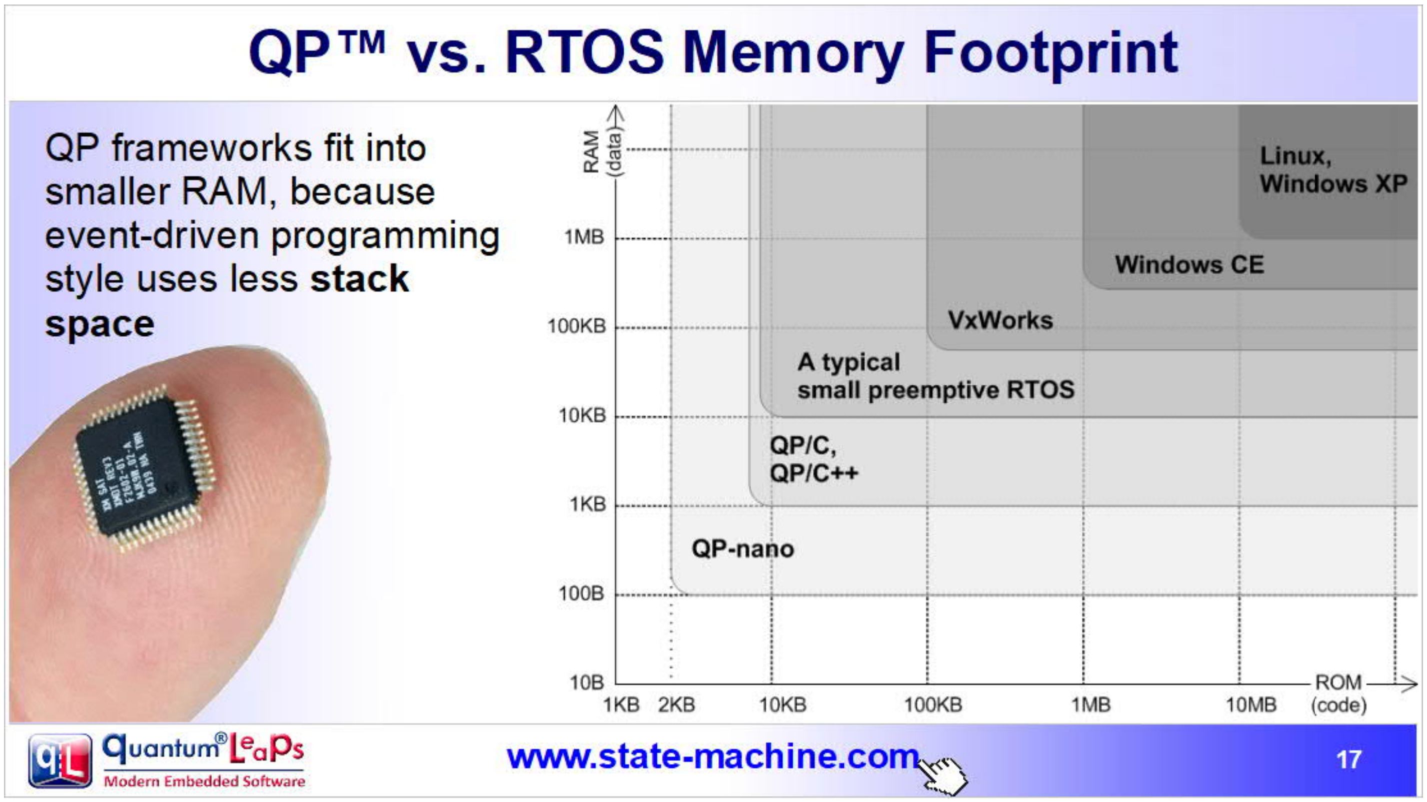
NOTE: All QP™ frameworks are fundamentally object-oriented, which means that the frameworks themselves and your applications derived from the frameworks are fundamentally composed of *classes* and only classes can have *state machines* associated with them.

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++**



QP™ vs. RTOS Memory Footprint

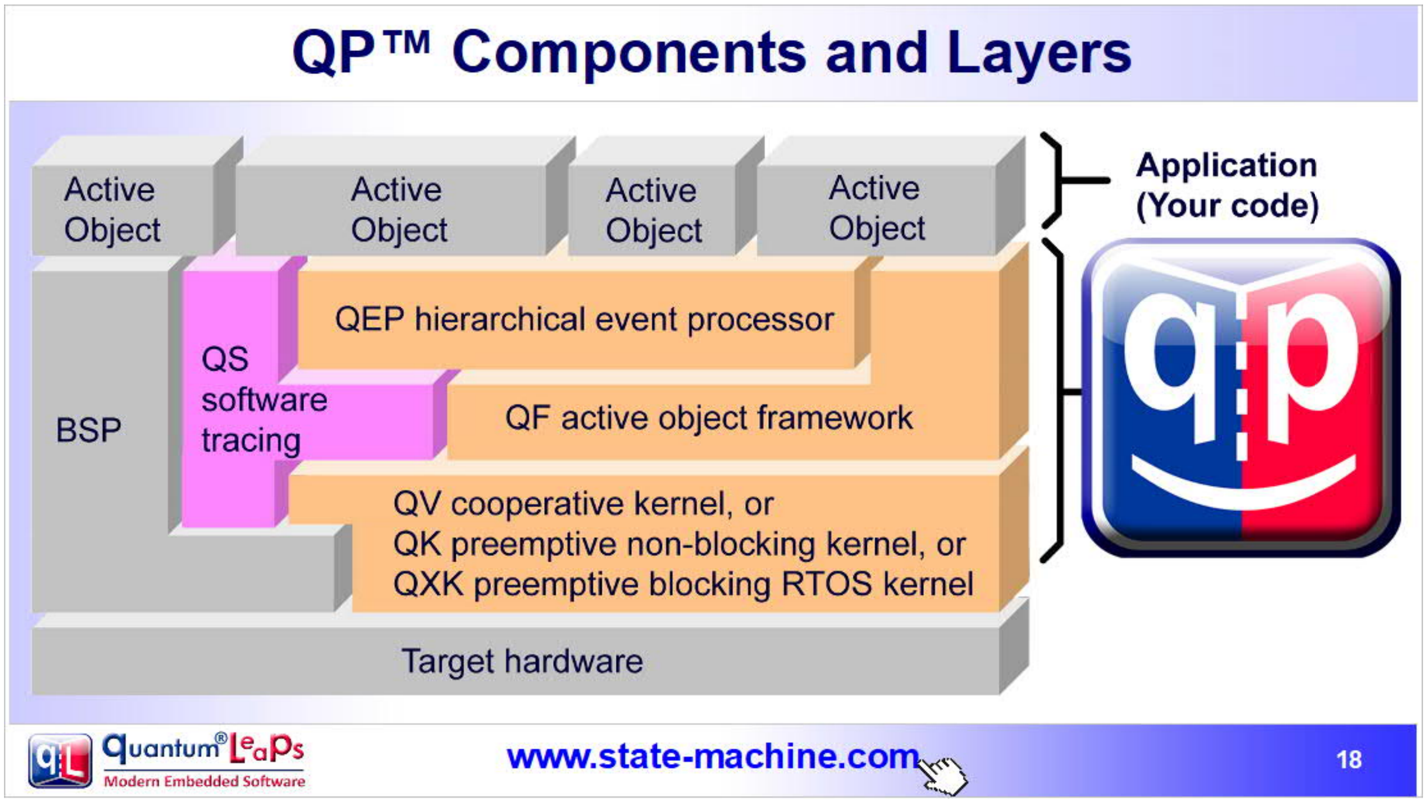
QP frameworks fit into smaller RAM, because event-driven programming style uses less **stack space**

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 an Active Object framework *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.

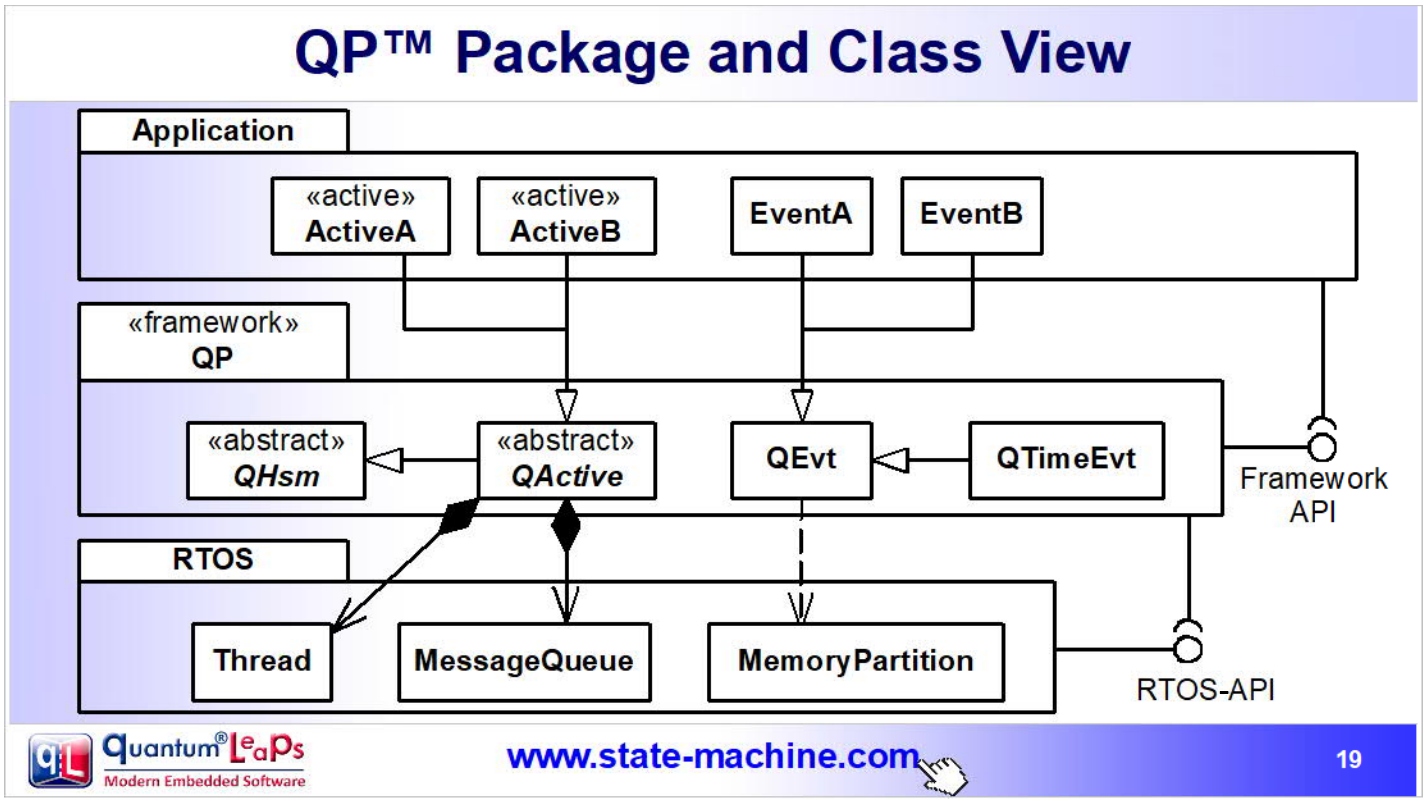
For example, the diagram shows the RAM/ROM sizes of the QP/C, QP/C++, and QP-nano real-time embedded frameworks versus a number of conventional (RT)OSes. The diagram shows the total system size as opposed to just the RTOS/OS footprints. As you can see, when compared to conventional RTOSes, QP™ frameworks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

All these characteristics make event-driven 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.


QP™ Components and Layers
Active Object
Active Object
Active Object
Active Object
Application (Your code)
QEP hierarchical event processor
QS software tracing
BSP
QF active object framework
QV cooperative kernel, or
QK preemptive non-blocking kernel, or
QXK preemptive blocking RTOS kernel
Target hardware
Quantum® LeaPs
Modern Embedded Software
www.state-machine.com
18

The QP™ frameworks 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.

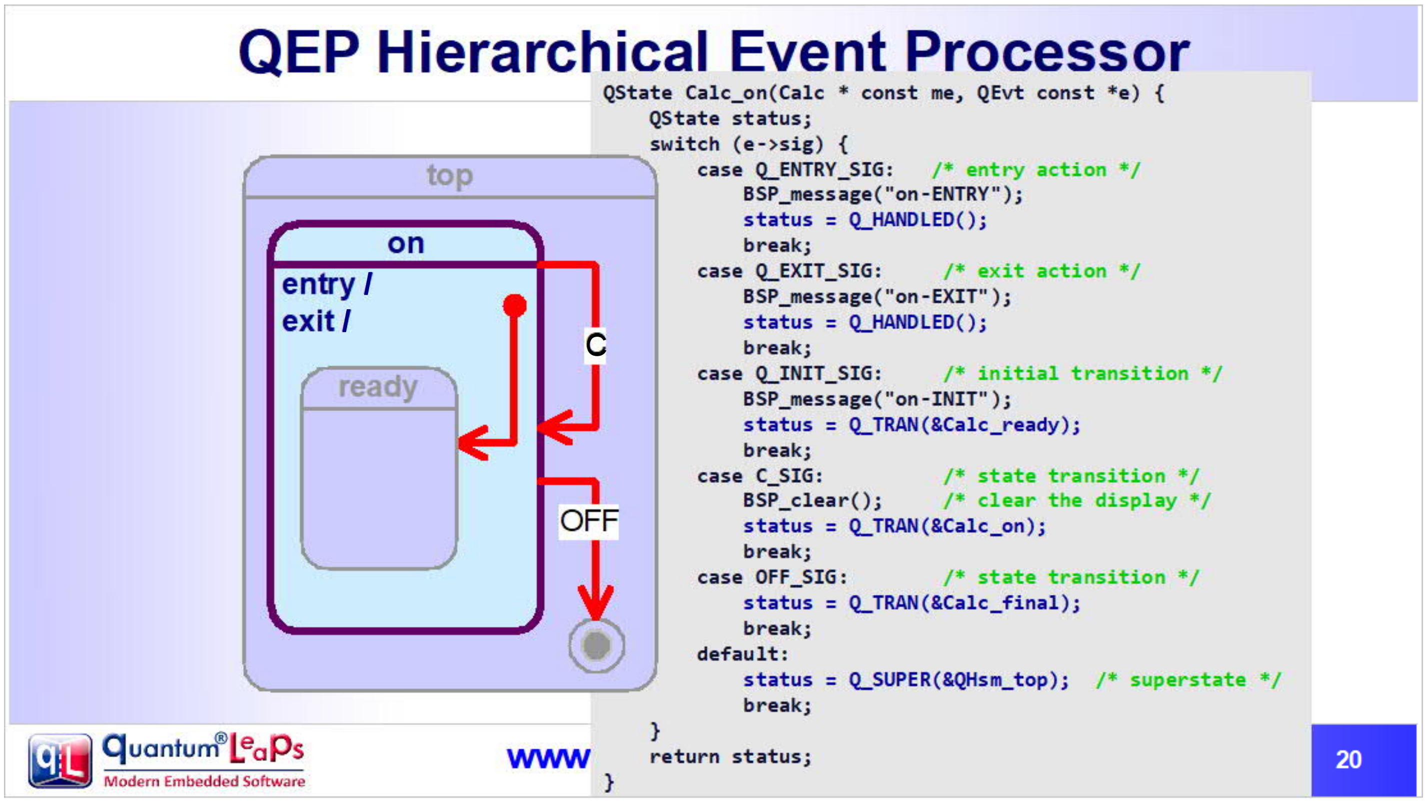


QP™ Package and Class View

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 sub-classed into application-specific time events.



QEP Hierarchical Event Processor

```
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG:    /* entry action */
            BSP_message("on-ENTRY");
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG:     /* exit action */
            BSP_message("on-EXIT");
            status = Q_HANDLED();
            break;
        case Q_INIT_SIG:     /* initial transition */
            BSP_message("on-INIT");
            status = Q_TRAN(&Calc_ready);
            break;
        case C_SIG:          /* state transition */
            BSP_clear();     /* clear the display */
            status = Q_TRAN(&Calc_on);
            break;
        case OFF_SIG:        /* state transition */
            status = Q_TRAN(&Calc_final);
            break;
        default:
            status = Q_SUPER(&QHsm_top);  /* superstate */
            break;
    }
    return status;
}
```

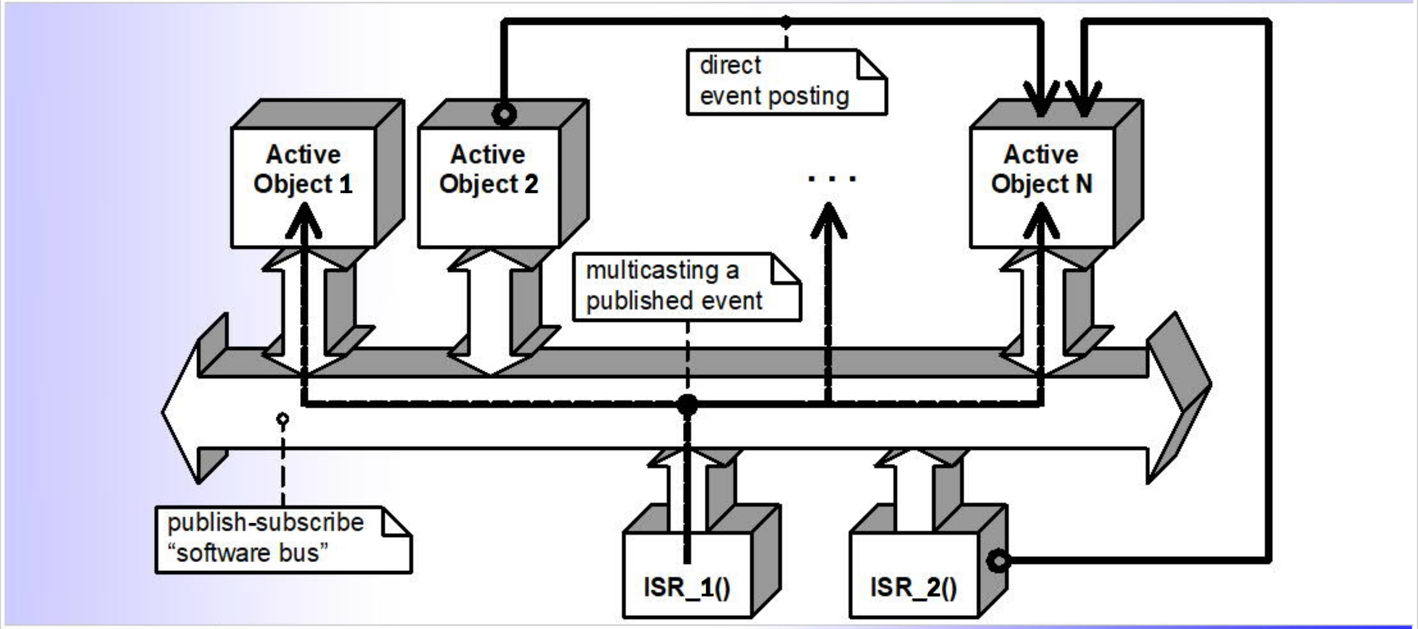
20

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:

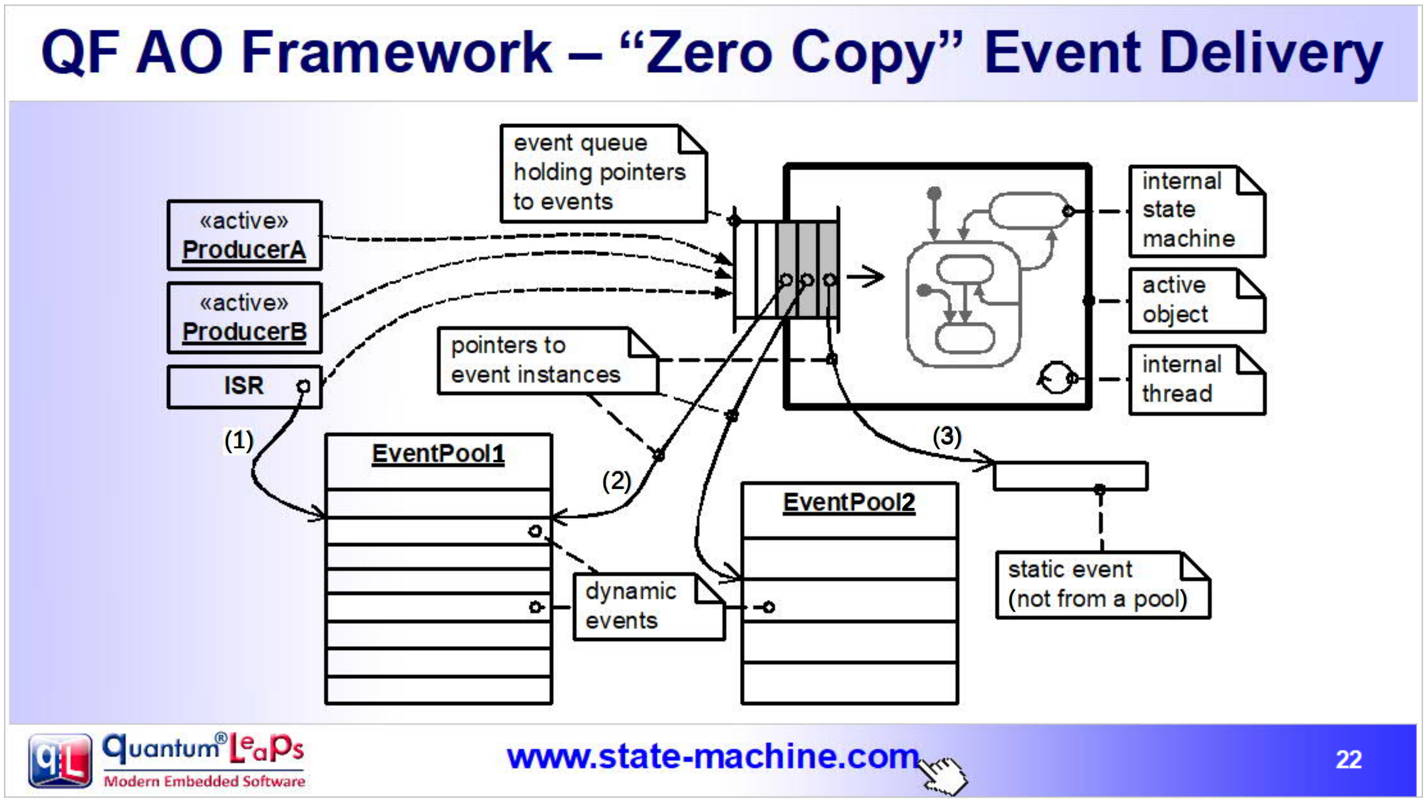
- Hierarchical state nesting
- Entry/exit actions in states
- Regular transitions
- Internal transitions
- Nested initial transitions
- Guard conditions on all transition types

QF AO Framework – “Software Bus”



The main job of an active-object framework is to provide execution environment (thread) to each active object and to provide **thread-safe**, asynchronous mechanisms to exchange events.

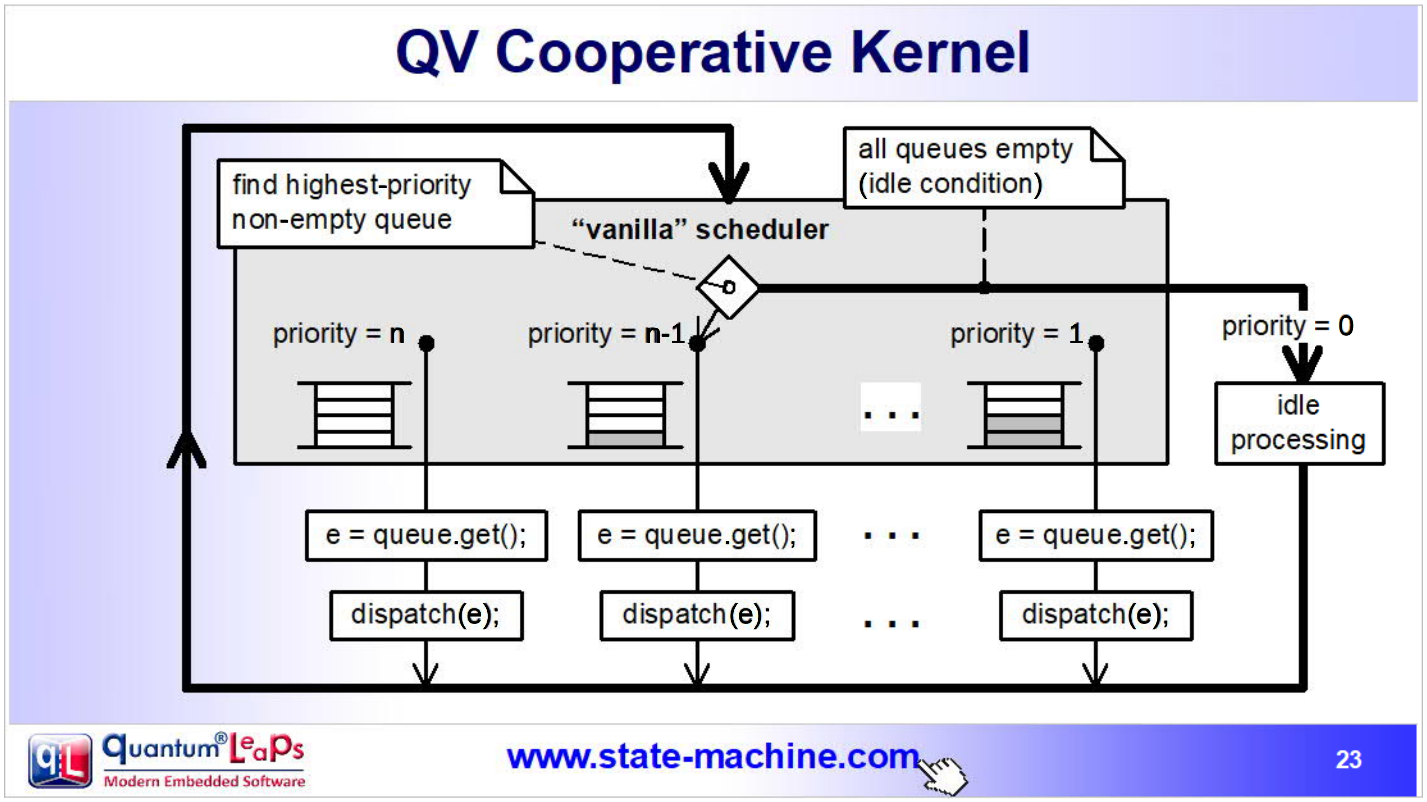
The QF framework serves as a “software bus” to connect active objects. The framework supports direct event posting as well as publish-subscribe event exchange.



QF AO Framework – "Zero Copy" Event Delivery

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.

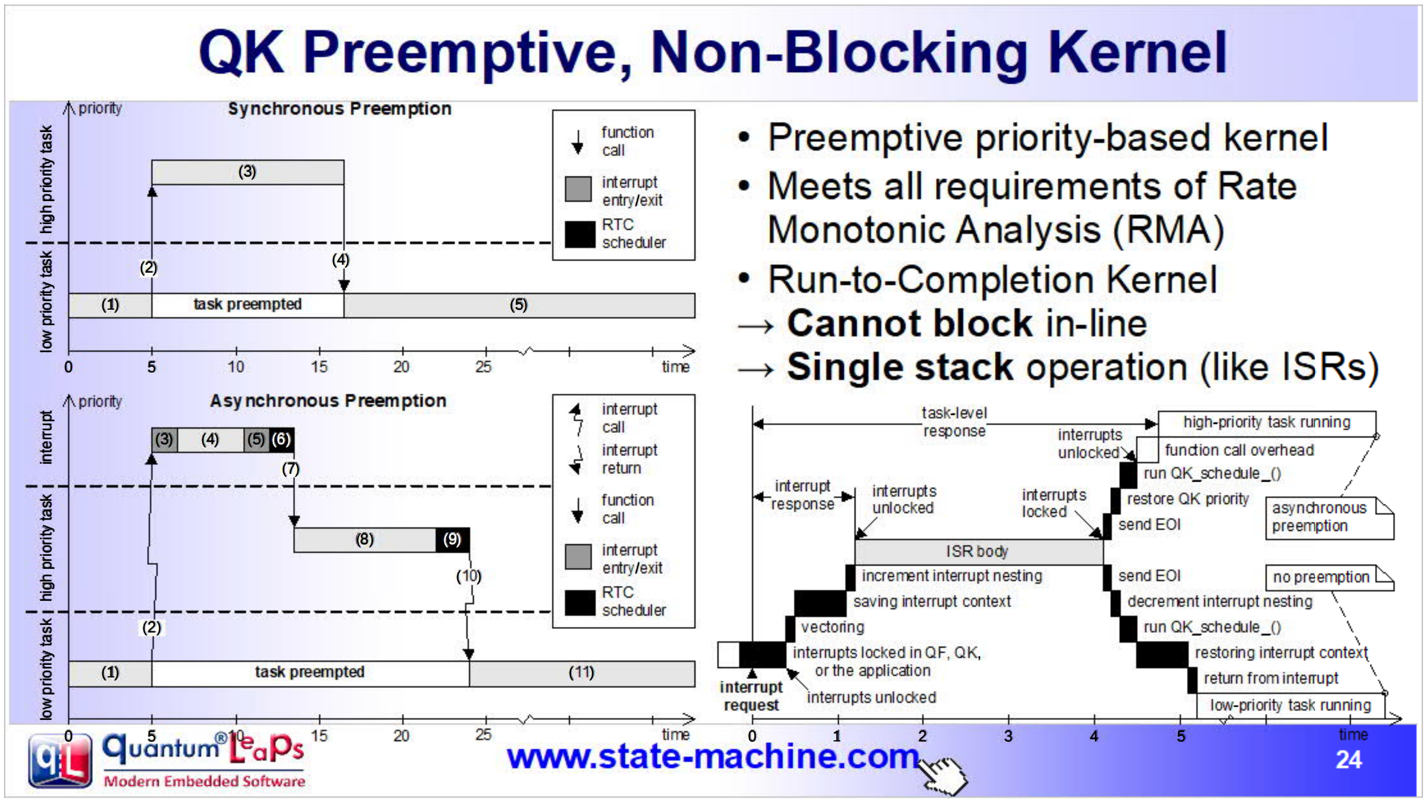

QV Cooperative Kernel

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 callack*, 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.



QK Preemptive, Non-Blocking Kernel

- Preemptive priority-based kernel
- Meets all requirements of Rate Monotonic Analysis (RMA)
- Run-to-Completion Kernel

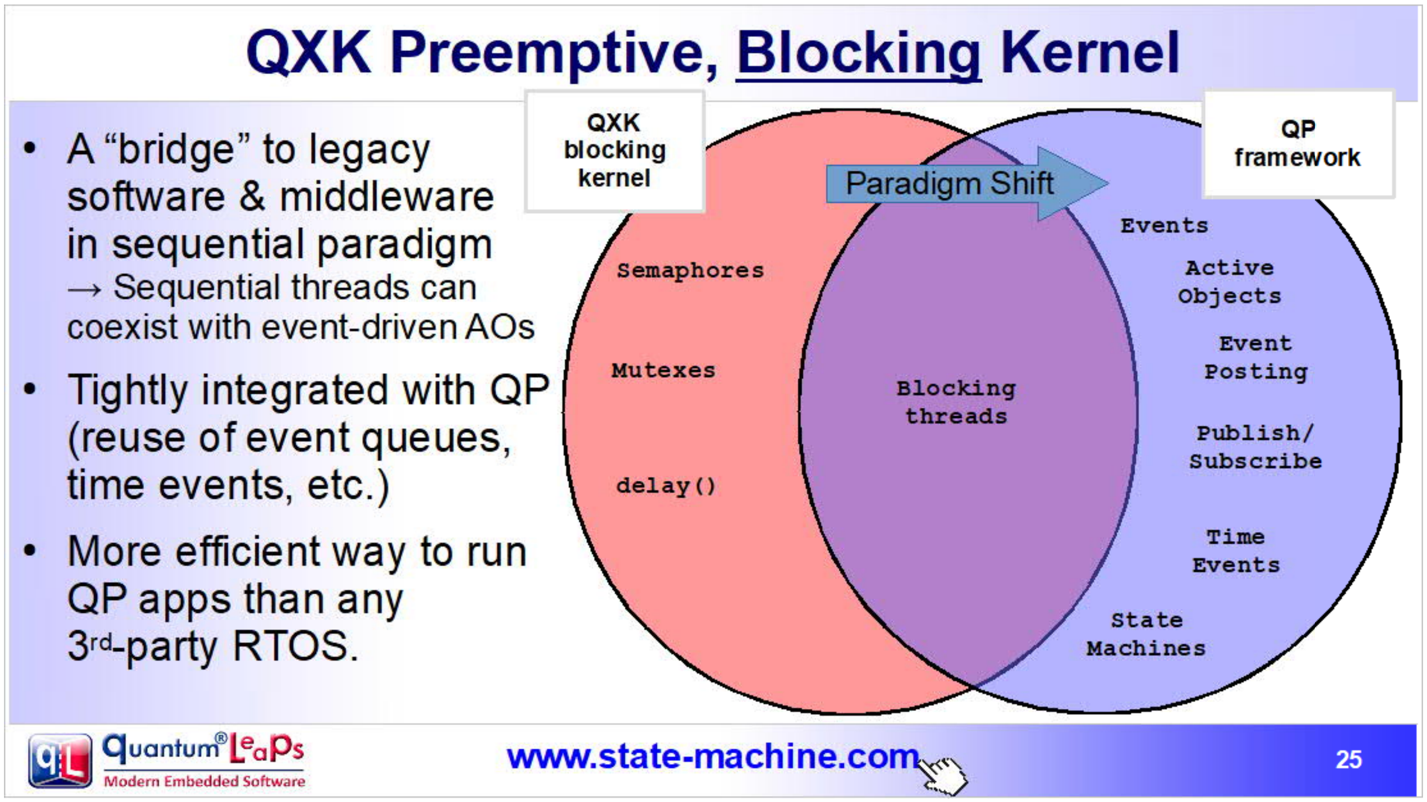
→ **Cannot block** in-line

→ **Single stack** operation (like ISRs)

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.



QXK Preemptive, Blocking Kernel

- A “bridge” to legacy software & middleware in sequential paradigm
  → Sequential threads can coexist with event-driven AOs
- Tightly integrated with QP (reuse of event queues, time events, etc.)
- More efficient way to run QP apps than any 3rd-party RTOS.

QXK blocking kernel
QP framework
Paradigm Shift
Semaphores
Mutexes
delay()
Blocking threads
Events
Active Objects
Event Posting
Publish/ Subscribe
Time Events
State Machines

Quantum® LeaPs Modern Embedded Software

www.state-machine.com

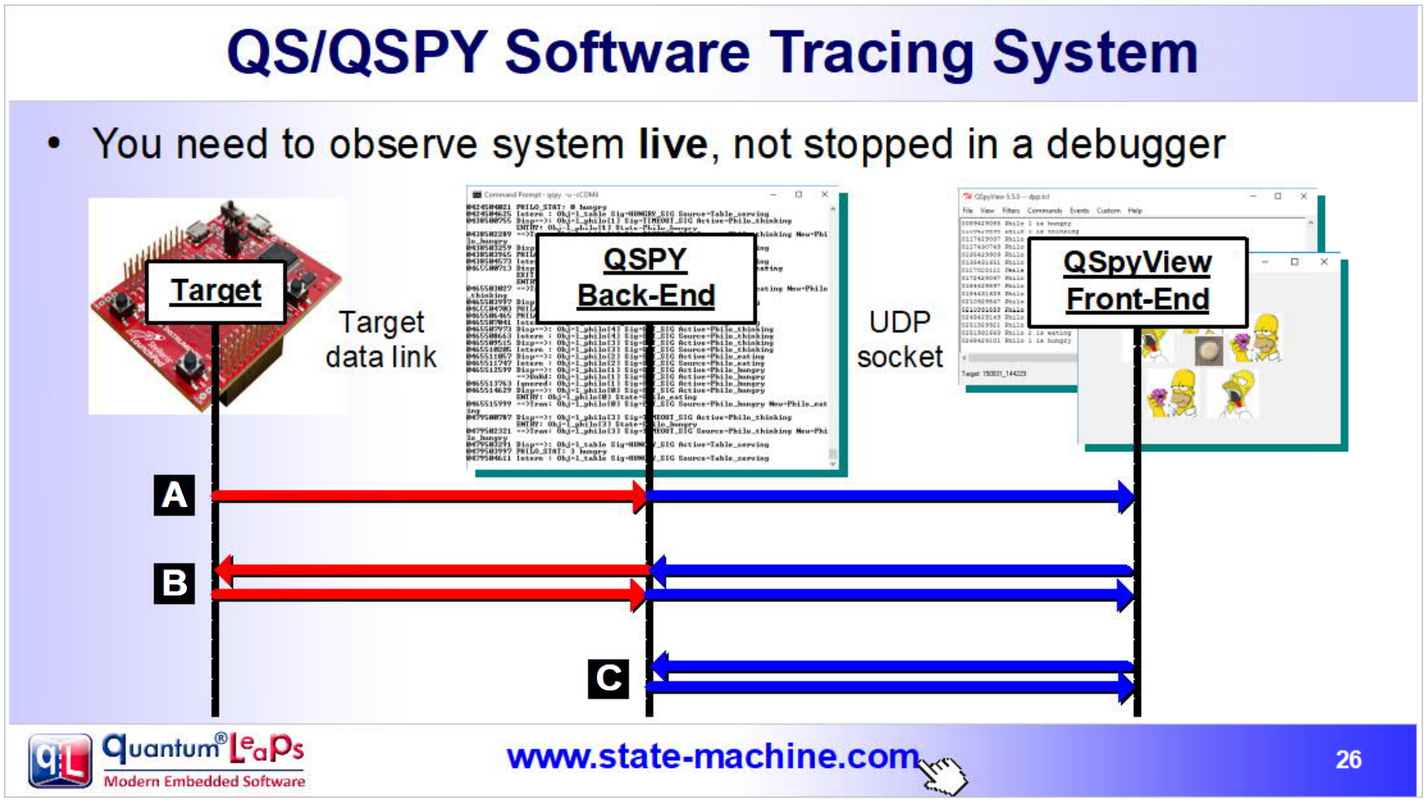
25

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, mutextes, 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 facilites, such as event queues. QXK is currently available for all ARM Cortex-M cores (M0/M0+/M3/M4/M4F/M7).



QS/QSPY Software Tracing System

- You need to observe system **live**, not stopped in a debugger

Quantum® LeaPs Modern Embedded Software

www.state-machine.com

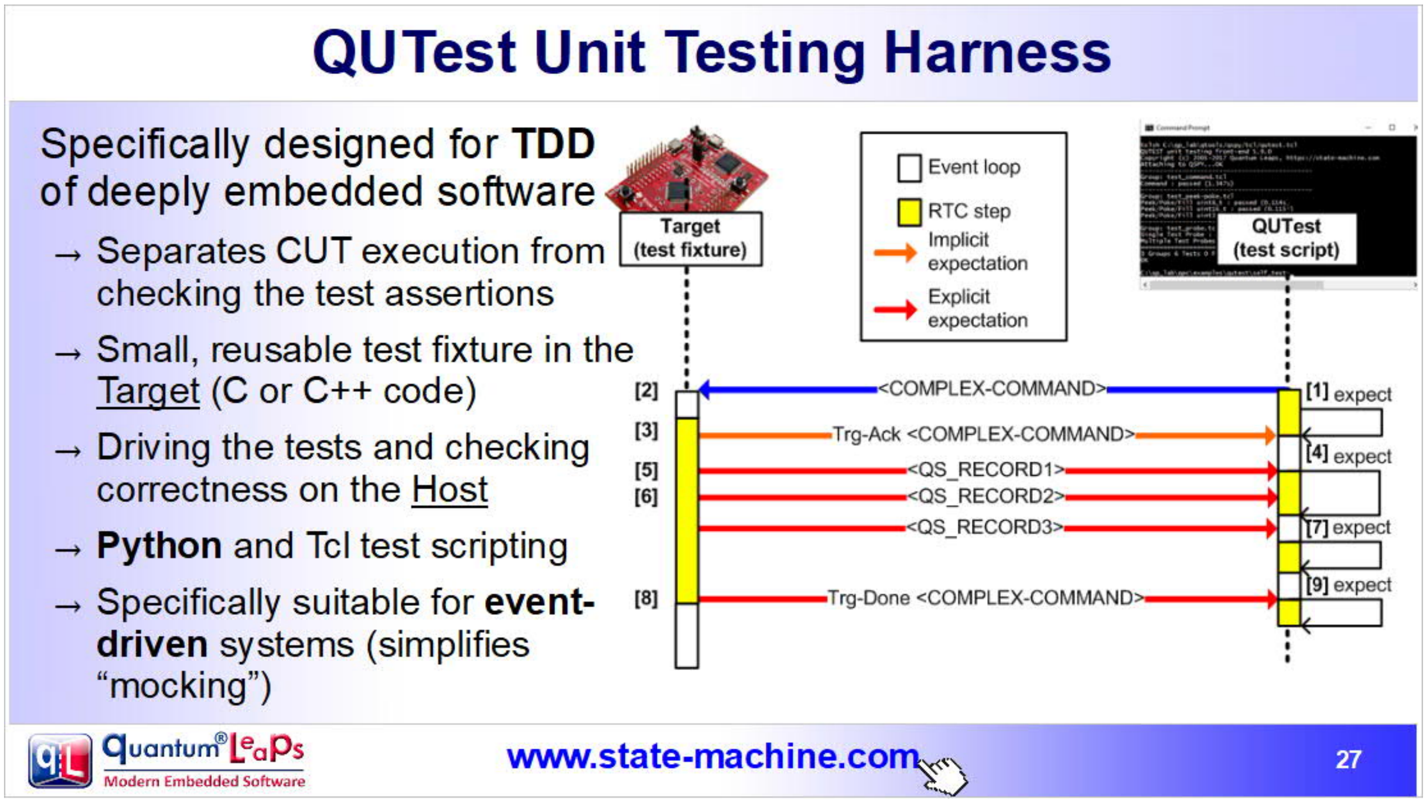
26

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 in to 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 Unit Testing Harness

Specifically designed for **TDD** of deeply embedded software

→ Separates CUT execution from checking the test assertions

→ Small, reusable test fixture in the Target (C or C++ code)

→ Driving the tests and checking correctness on the Host

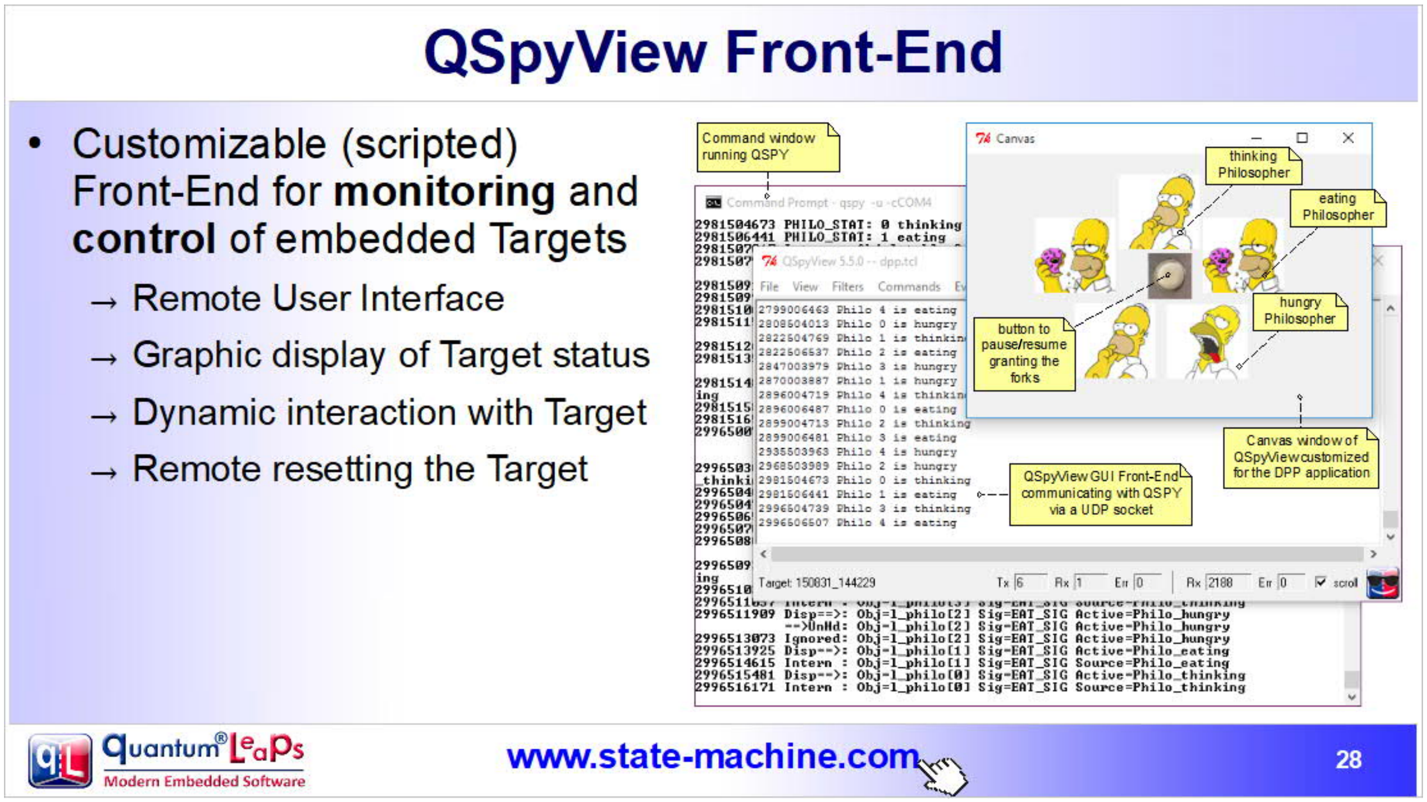
→ **Python** and Tcl test scripting

→ Specifically suitable for **event-driven** systems (simplifies “mocking”)

The QSPY host application can be extended with a UDP socket, which is open for communication with various Front-Ends (such as QUTest and QspyView).

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. Features are:

- Separates CUT execution from checking the test assertions
- Small, reusable test fixtures running in the Target (C or C++ code)
- Driving the tests and checking correctness performed from the Host
- **Python** and Tcl test scripting
- Specifically suitable for event-driven systems (simplifies “mocking”)



QSpyView Front-End

- Customizable (scripted) Front-End for **monitoring** and **control** of embedded Targets
  - → Remote User Interface
  - → Graphic display of Target status
  - → Dynamic interaction with Target
  - → Remote resetting the Target

Command window running QSPY
Command Prompt - qspy -u -cCOM4
QSpyView 5.5.0 -- dpp.tcl
File View Filters Commands
Canvas
thinking Philosopher
eating Philosopher
hungry Philosopher
button to pause/resume granting the forks
Canvas window of QSpyView customized for the DPP application
QSpyView GUI Front-End communicating with QSPY via a UDP socket
Target: 150831_144229 Tx 6 Rx 1 Err 0 Rx 2188 Err 0 scroll

```
2981504673 PHILO_STAT: 0 thinking
2981506441 PHILO_STAT: 1 eating
```

```
2799006463 Philo 4 is eating
2808504013 Philo 0 is hungry
2822504769 Philo 1 is thinking
2822506537 Philo 2 is eating
2847003979 Philo 3 is hungry
2870003887 Philo 1 is hungry
2896004719 Philo 4 is thinking
2896006487 Philo 0 is eating
2899004713 Philo 2 is thinking
2899006481 Philo 3 is eating
2935503963 Philo 4 is hungry
2968503989 Philo 2 is hungry
2981504673 Philo 0 is thinking
2981506441 Philo 1 is eating
2996504739 Philo 3 is thinking
2996506507 Philo 4 is eating
```

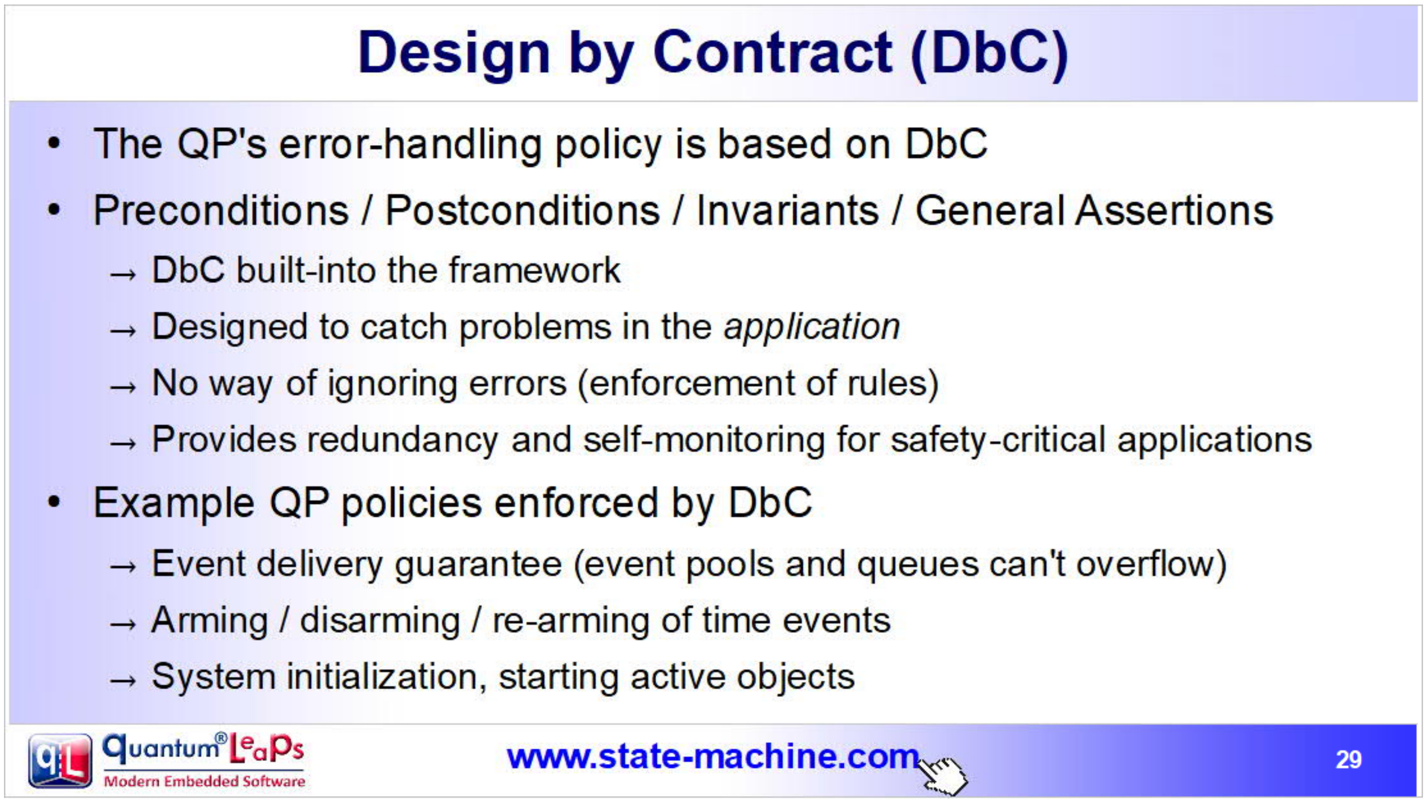
```
2996511909 Disp==>: Obj=l_philo[2] Sig=EAT_SIG Active=Philo_hungry
           ==>UnHd: Obj=l_philo[2] Sig=EAT_SIG Active=Philo_hungry
2996513073 Ignored: Obj=l_philo[2] Sig=EAT_SIG Active=Philo_hungry
2996513925 Disp==>: Obj=l_philo[1] Sig=EAT_SIG Active=Philo_eating
2996514615 Intern : Obj=l_philo[1] Sig=EAT_SIG Source=Philo_eating
2996515481 Disp==>: Obj=l_philo[0] Sig=EAT_SIG Active=Philo_thinking
2996516171 Intern : Obj=l_philo[0] Sig=EAT_SIG Source=Philo_thinking
```

Quantum® LeaPs Modern Embedded Software

www.state-machine.com

QSpyView leverages the QP/Spy software tracing system to provide a customizable (Tcl/Tk) **remote user interface** to embedded targets for monitoring and control of embedded devices:

- Graphically display information about the running Target
- Dynamically interact with the running Target
- Set global QS filters inside the Target
- Remotely reset of the Target



Design by Contract (DbC)

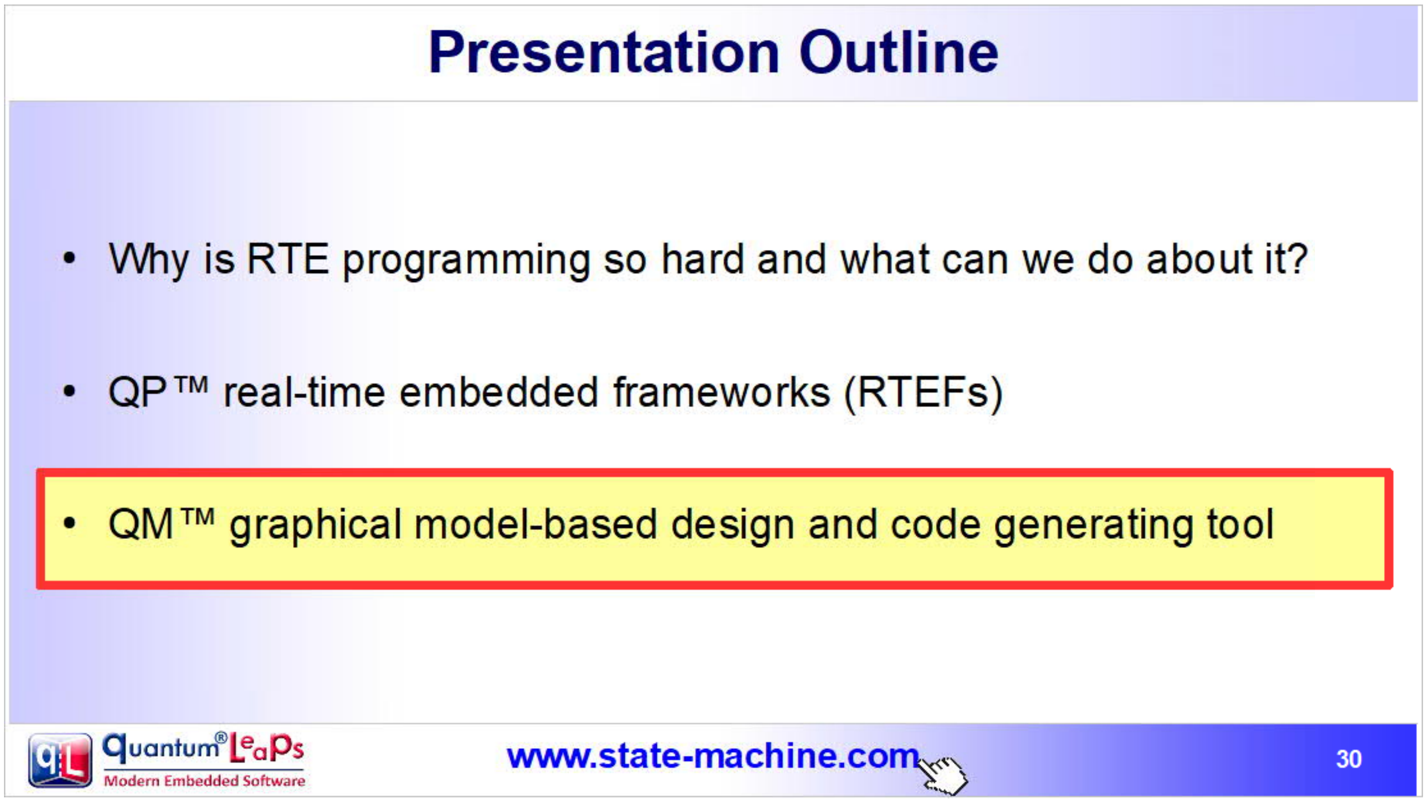
- The QP's error-handling policy is based on DbC
- Preconditions / Postconditions / Invariants / General Assertions
  - → DbC built-into the framework
  - → Designed to catch problems in the *application*
  - → No way of ignoring errors (enforcement of rules)
  - → Provides redundancy and self-monitoring for safety-critical applications
- Example QP policies enforced by DbC
  - → Event delivery guarantee (event pools and queues can't overflow)
  - → Arming / disarming / re-arming of time events
  - → System initialization, starting active objects

Quantum® LeaPs Modern Embedded Software www.state-machine.com 29

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.



Presentation Outline

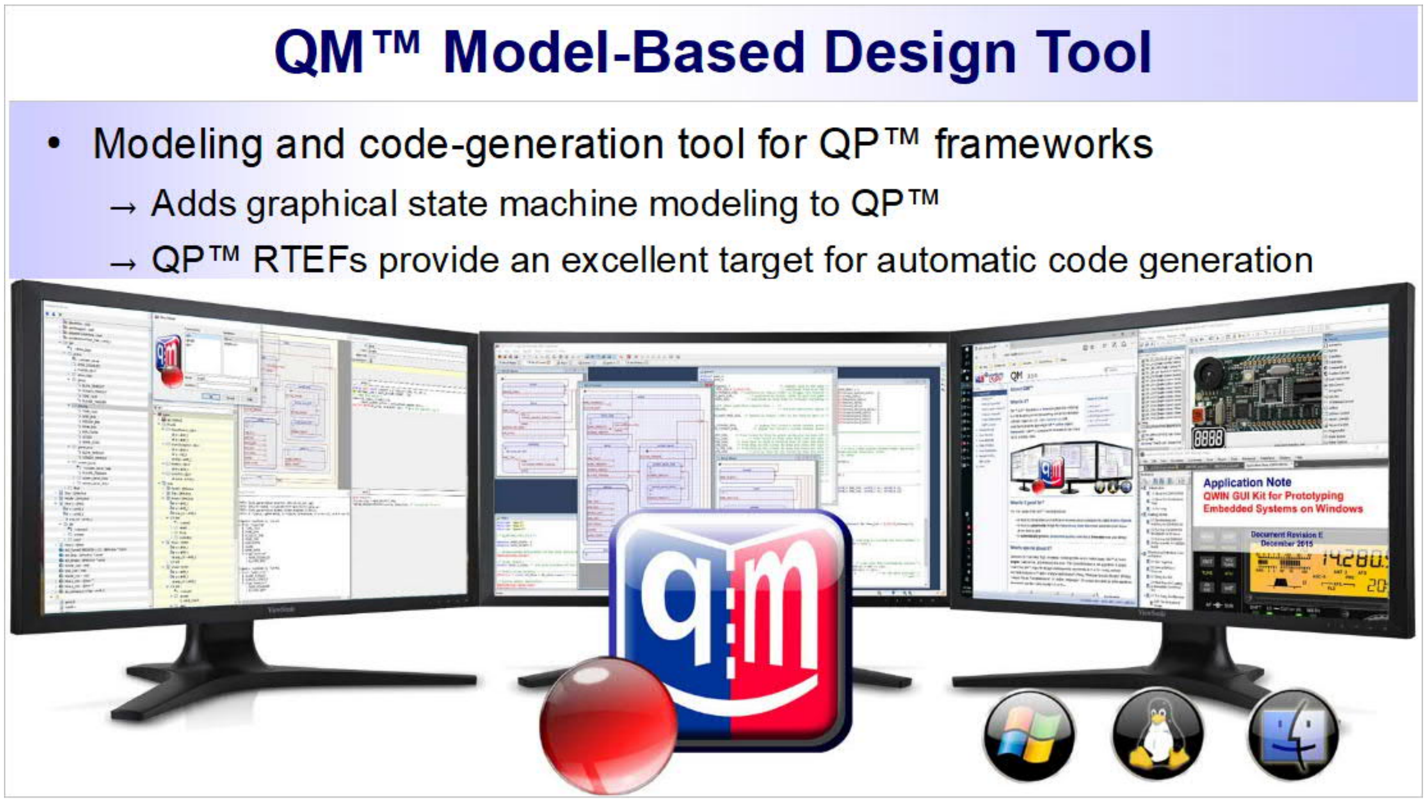
- Why is RTE programming so hard and what can we do about it?
- QP™ real-time embedded frameworks (RTEFs)
- QM™ graphical model-based design and code generating tool

Quantum® LeaPs Modern Embedded Software

www.state-machine.com

30

This section introduces the QM graphical model-based design tool for the QP real-time embedded frameworks.



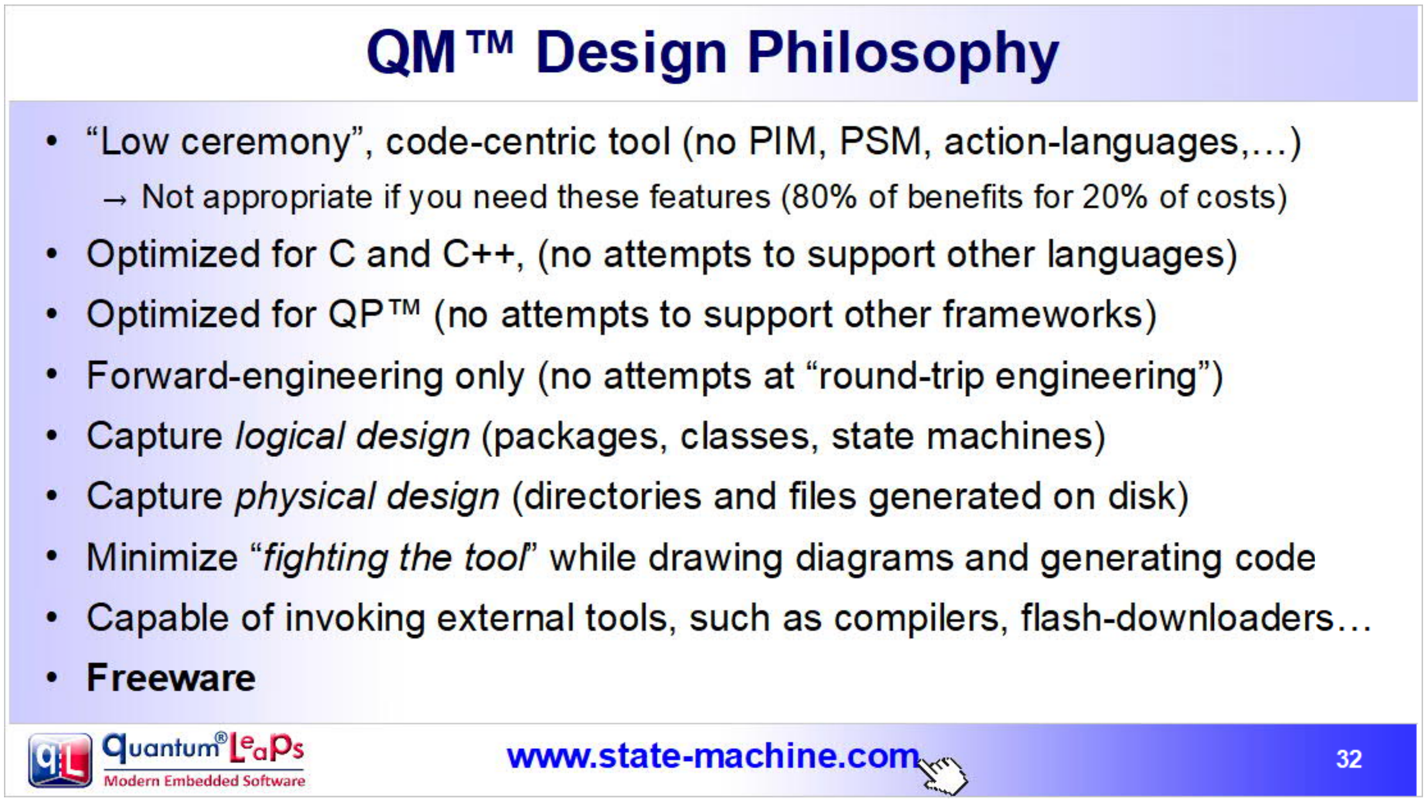
QM™ Model-Based Design Tool

- Modeling and code-generation tool for QP™ frameworks
  → Adds graphical state machine modeling to QP™
  → QP™ RTEFs provide an excellent target for automatic code generation

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.



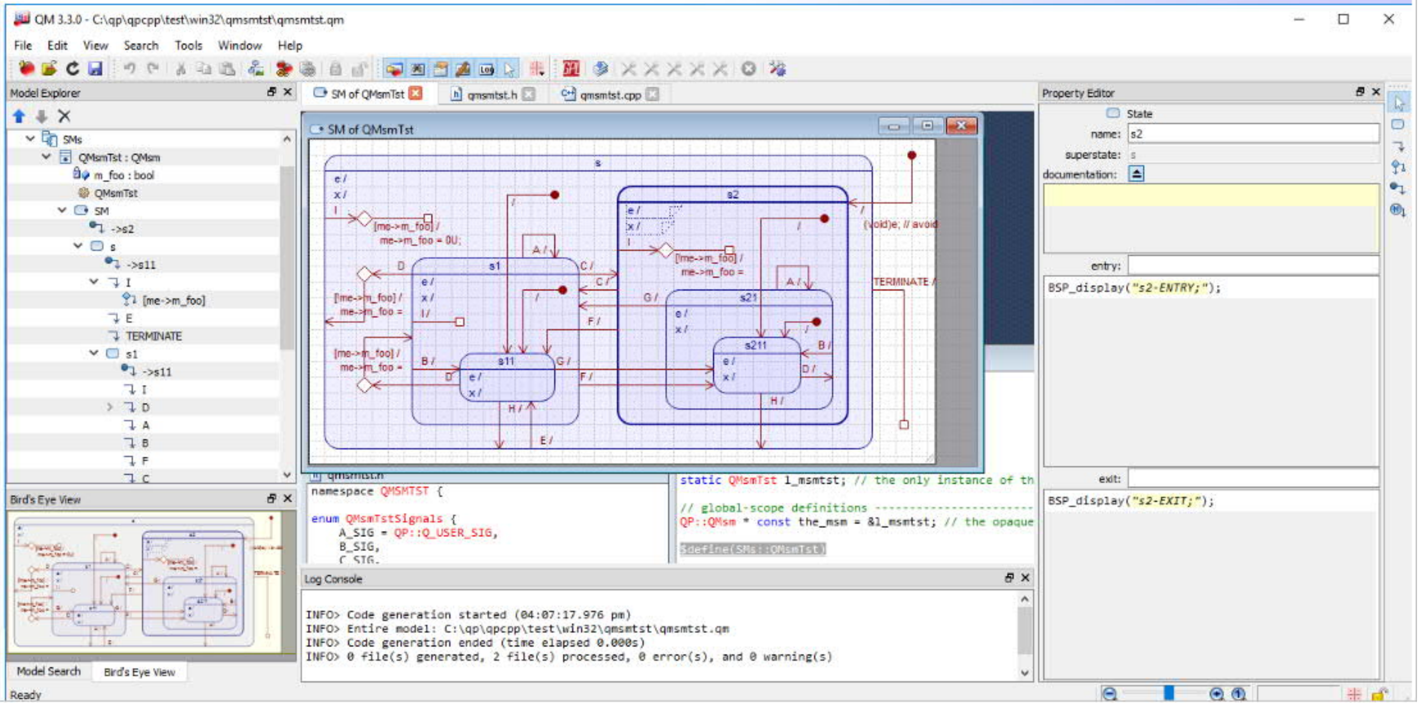
QM™ Design Philosophy

- "Low ceremony", code-centric tool (no PIM, PSM, action-languages,…)
  - → Not appropriate if you need these features (80% of benefits for 20% of costs)
- Optimized for C and C++, (no attempts to support other languages)
- Optimized for QP™ (no attempts to support other frameworks)
- Forward-engineering only (no attempts at "round-trip engineering")
- Capture *logical design* (packages, classes, state machines)
- Capture *physical design* (directories and files generated on disk)
- Minimize "*fighting the tool*" while drawing diagrams and generating code
- Capable of invoking external tools, such as compilers, flash-downloaders…
- **Freeware**

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++.

Logical Design (Packages/Classes/State Machines)



```
namespace QMSMTST {

enum QMsmTstSignals {
    A_SIG = QP::Q_USER_SIG,
    B_SIG,
    C_SIG,
```

```
static QMsmTst l_msmtst; // the only instance of th

// global-scope definitions -----------------------
QP::QMsm * const the_msm = &l_msmtst; // the opaque

$define(SMs::QMsmTst)
```

State
name: s2
superstate: s
documentation:
entry:

```
BSP_display("s2-ENTRY;");
```

exit:

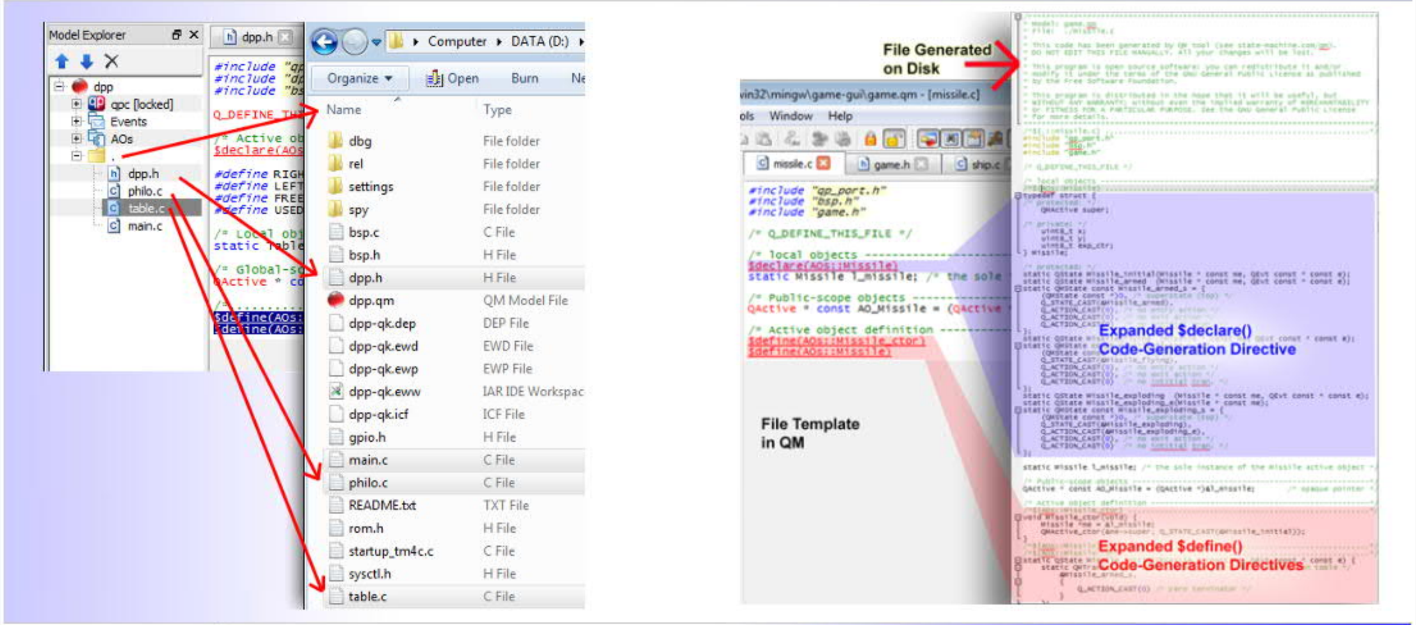
```
BSP_display("s2-EXIT;");
```

```
INFO> Code generation started (04:07:17.976 pm)
INFO> Entire model: C:\qp\qpcpp\test\win32\qmsmtst\qmsmtst.qm
INFO> Code generation ended (time elapsed 0.000s)
INFO> 0 file(s) generated, 2 file(s) processed, 0 error(s), and 0 warning(s)
```

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.

Physical Design (Directories / Files)



Name	Type
dbg	File folder
rel	File folder
settings	File folder
spy	File folder
bsp.c	C File
bsp.h	H File
dpp.h	H File
dpp.qm	QM Model File
dpp-qk.dep	DEP File
dpp-qk.ewd	EWD File
dpp-qk.ewp	EWP File
dpp-qk.eww	IAR IDE Workspac
dpp-qk.icf	ICF File
gpio.h	H File
main.c	C File
philo.c	C File
README.txt	TXT File
rom.h	H File
startup_tm4c.c	C File
sysctl.h	H File
table.c	C File

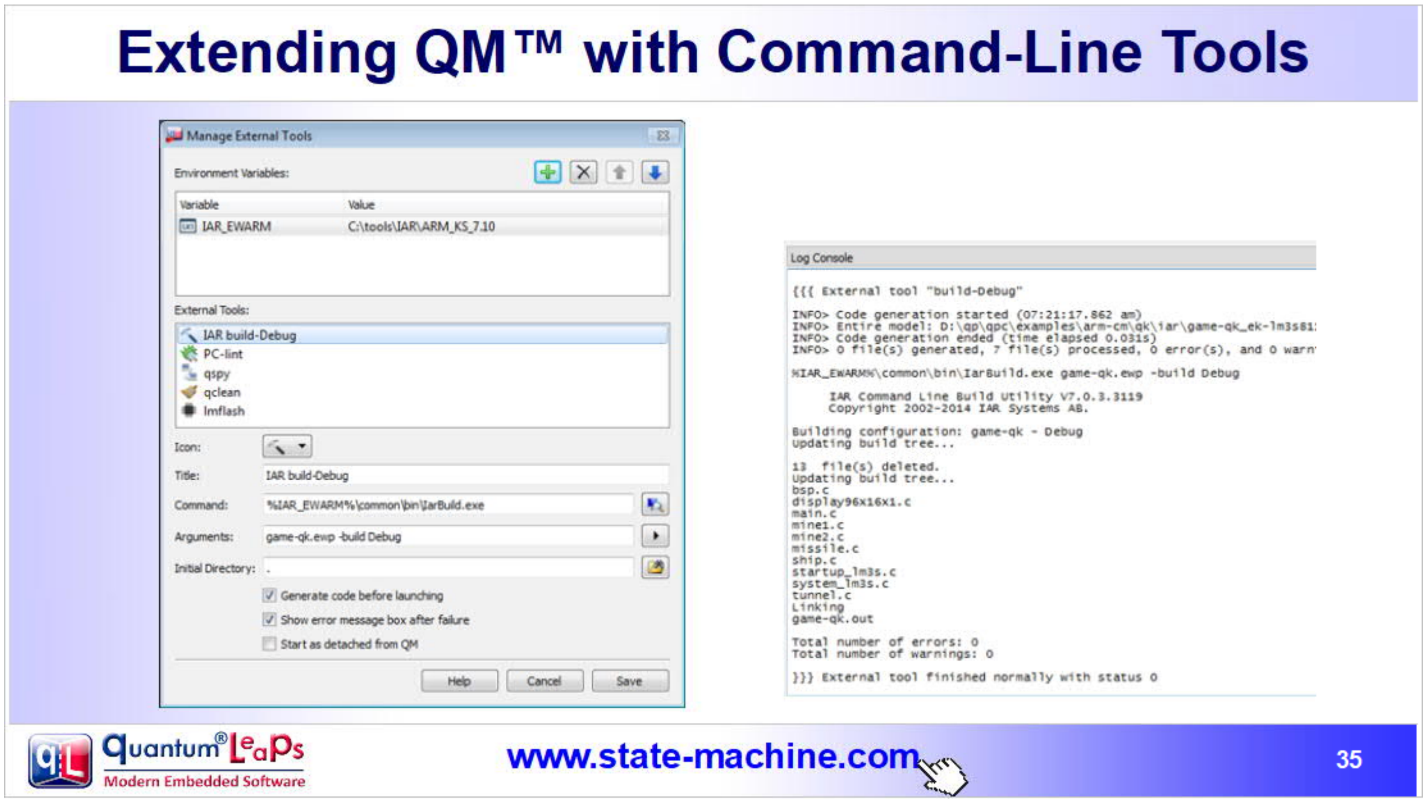
win32\mingw\game-gui\game.qm - [missile.c]

```
#include "qp_port.h"
#include "bsp.h"
#include "game.h"

/* Q_DEFINE_THIS_FILE */

/* local objects -----------
$declare(AOs::Missile)
static Missile l_missile; /* the sole

/* Public-scope objects ----------
QActive * const AO_Missile = (QActive

/* Active object definition ----------
$define(AOs::Missile_ctor)
$define(AOs::Missile)
```

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 flexibly 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.



Extending QM™ with Command-Line Tools

Manage External Tools

Environment Variables:

Variable	Value
IAR_EWARM	C:\tools\IAR\ARM_KS_7.10

External Tools:

- IAR build-Debug
- PC-lint
- qspy
- qclean
- lmflash

Icon:

Title: IAR build-Debug

Command: %IAR_EWARM%\common\bin\IarBuild.exe

Arguments: game-qk.ewp -build Debug

Initial Directory: .

[✓] Generate code before launching

[✓] Show error message box after failure

[ ] Start as detached from QM

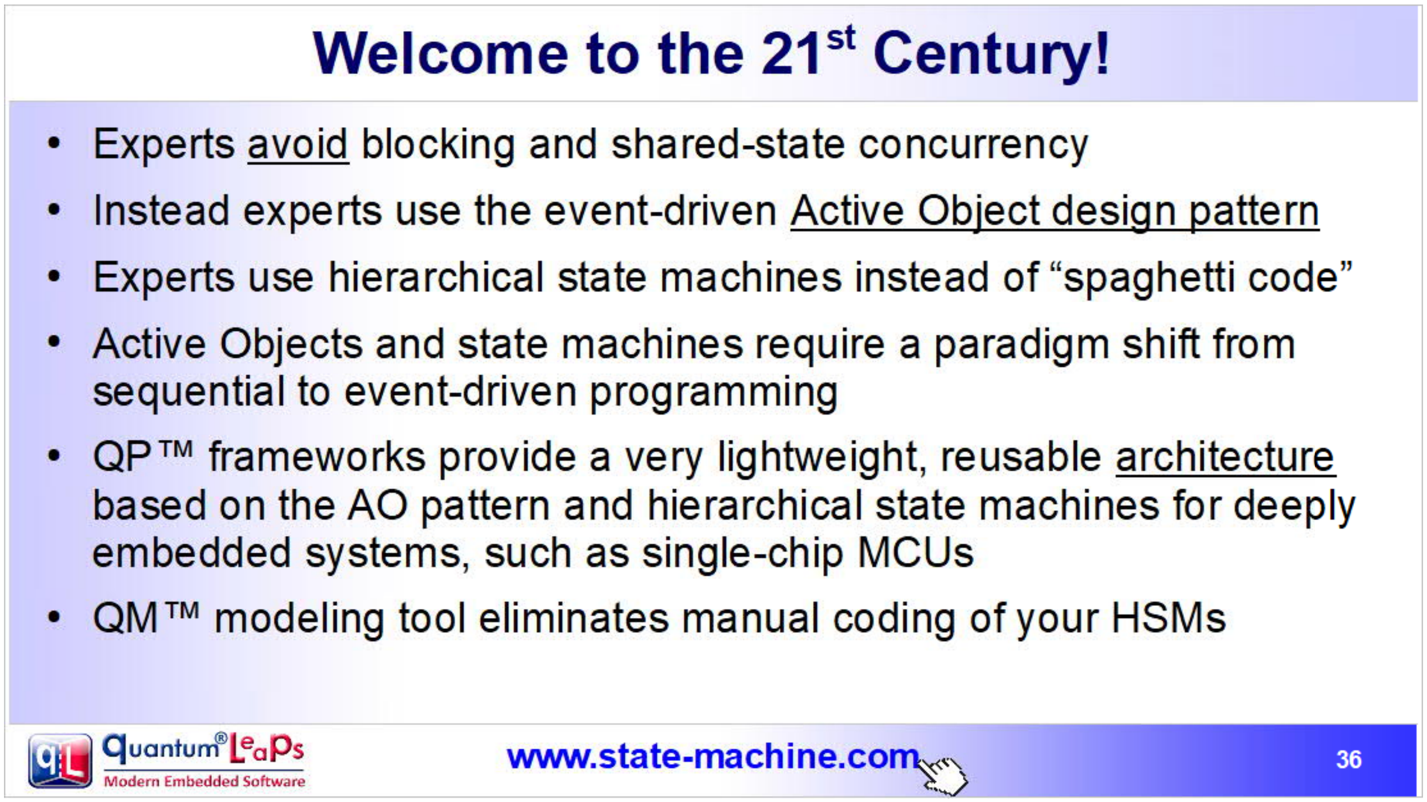
Log Console

```
{{{ External tool "build-Debug"

INFO> Code generation started (07:21:17.862 am)
INFO> Entire model: D:\qp\qpc\examples\arm-cm\qk\iar\game-qk_ek-lm3s81
INFO> Code generation ended (time elapsed 0.031s)
INFO> 0 file(s) generated, 7 file(s) processed, 0 error(s), and 0 warn

%IAR_EWARM%\common\bin\IarBuild.exe game-qk.ewp -build Debug

     IAR Command Line Build Utility V7.0.3.3119
     Copyright 2002-2014 IAR Systems AB.

Building configuration: game-qk - Debug
Updating build tree...

13  file(s) deleted.
Updating build tree...
bsp.c
display96x16x1.c
main.c
mine1.c
mine2.c
missile.c
ship.c
startup_lm3s.c
system_lm3s.c
tunnel.c
Linking
game-qk.out

Total number of errors: 0
Total number of warnings: 0

}}} External tool finished normally with status 0
```

Quantum® Leaps Modern Embedded Software www.state-machine.com 35

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 blocking and shared-state concurrency
- Instead experts use the event-driven Active Object design pattern
- Experts use hierarchical state machines instead of “spaghetti code”
- Active Objects and state machines require a paradigm shift from sequential to event-driven programming
- QP™ frameworks provide a very lightweight, reusable architecture based on the AO pattern and hierarchical state machines for deeply embedded systems, such as single-chip MCUs
- QM™ modeling tool eliminates manual coding of your HSMs

Quantum® LeaPs Modern Embedded Software www.state-machine.com