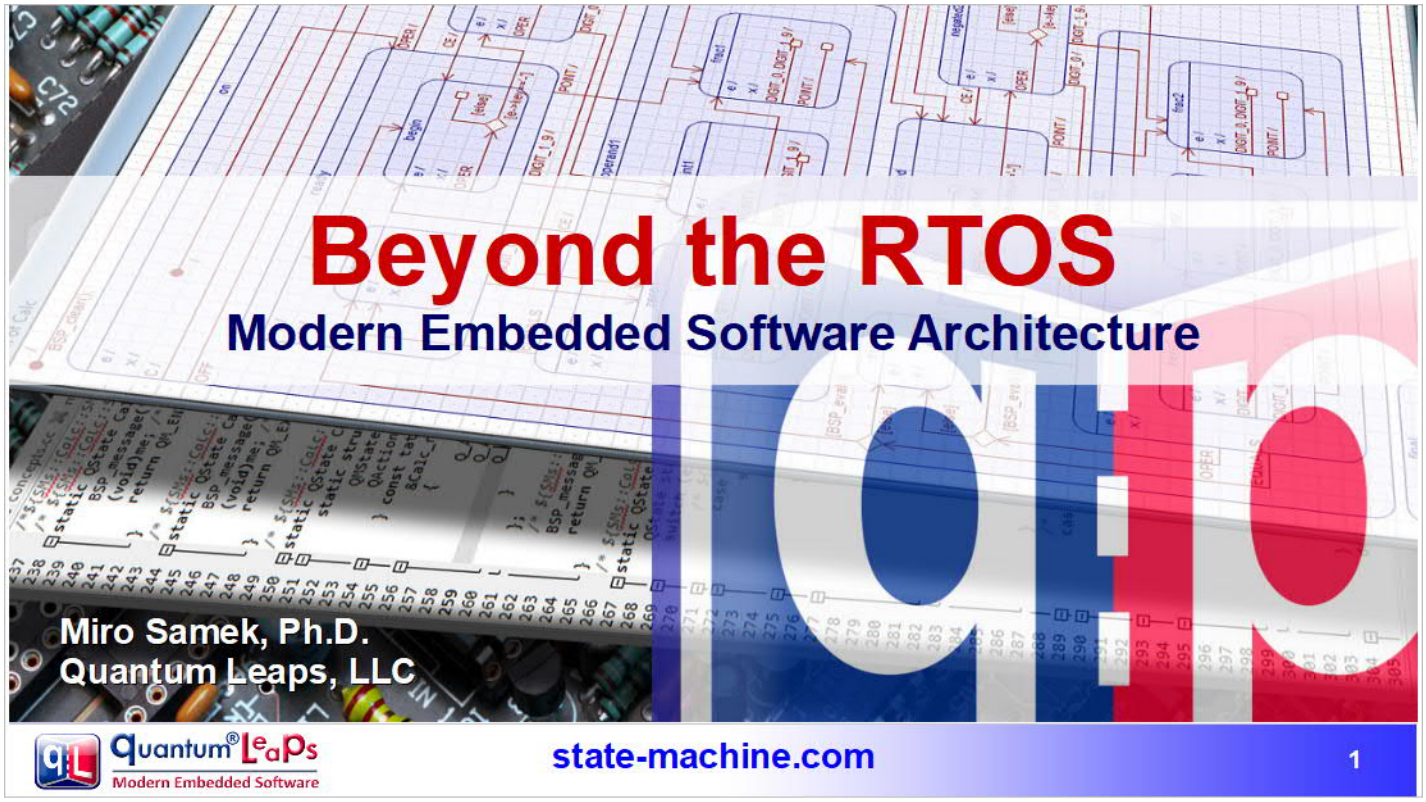
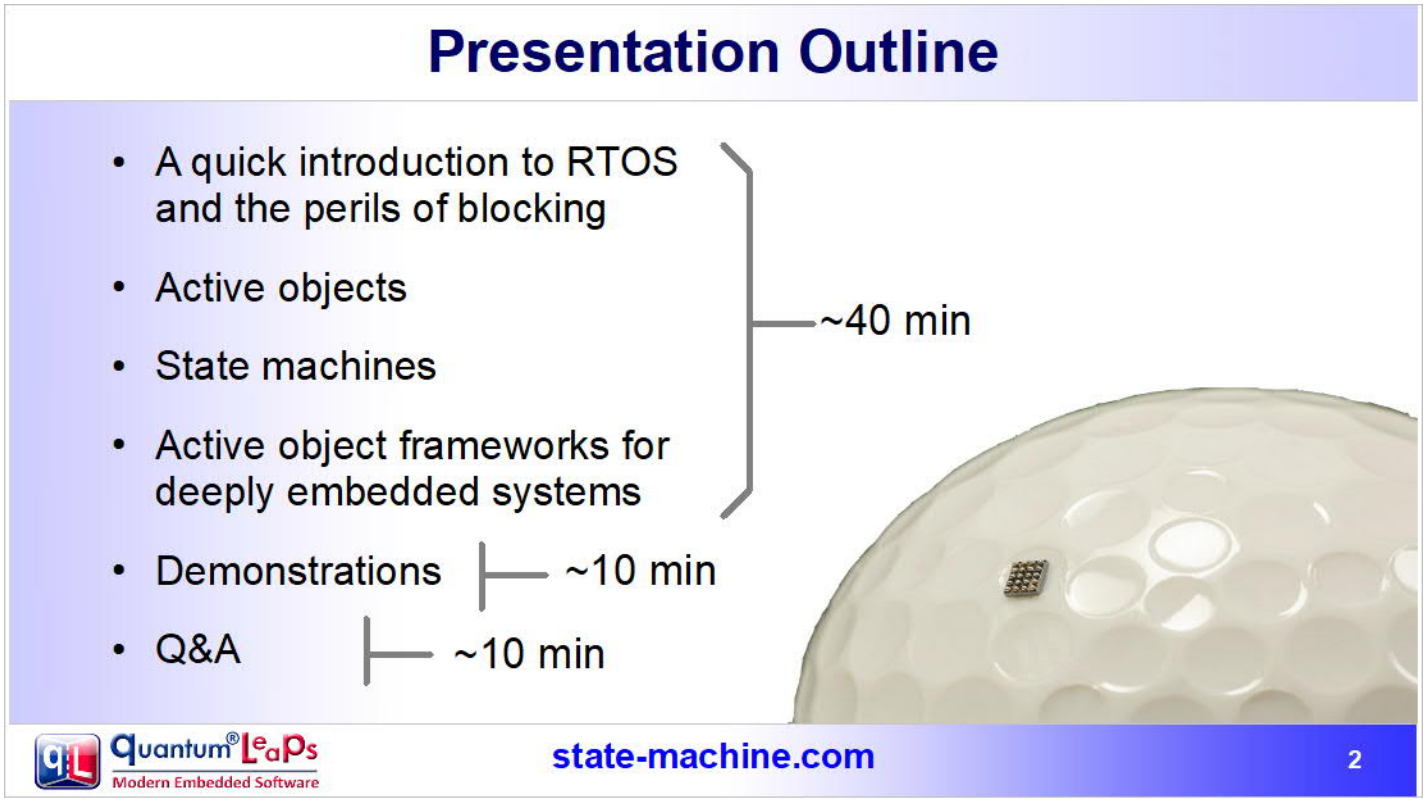


In this presentation I'd like to talk about a modern way to build real-time embedded software that goes beyond the traditional Real-Time Operating System (RTOS).

Even though the ideas I'll discuss today are certainly not new, the “reactive approach” that I'll present has been only recently gaining popularity as software developers from different industries independently re-discover better ways of designing concurrent software. These best practices universally favor event-driven, asynchronous, non-blocking, encapsulated active objects (a.k.a. actors) instead of sequential programming based on traditional RTOS.

This talk is based on my 15 years of experience with developing and refining active object frameworks for deeply embedded real-time systems.



My talk should take about 40 minutes, followed by 10 minutes of demonstrations and 10 minutes for questions and answers.

One comment, to avoid any confusion from the beginning: when I say RTOS, I mean a small real-time kernel designed for deeply embedded systems, such as single-chip, single-core microcontrollers. I specifically don't mean here embedded Linux, embedded Windows, or other such big OSes.

An example of a representative hardware that I have in mind for this talk is shown in the picture. Here, in a dimple of a golf ball, you can see a complete microcontroller in a silicon-size package. The chip contains the 32-bit ARM Cortex-M core, an impressive set of peripherals as well as several kilobytes of static-RAM for data and few hundred kilobytes of flash-ROM for code.

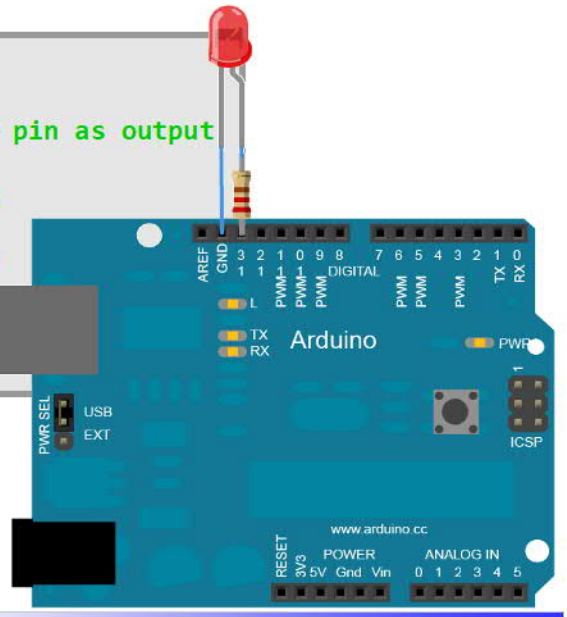
If you think that it's too small for any significant complexity, let me tell you from experience that this is plenty to shoot yourself in the foot.

BTW, worldwide shipments of microcontrollers reach some 20 billion units a year.

In the beginning was the “Superloop”

```
// adapted from the Arduino Blink Tutorial (*)
void main() {
    pinMode(LED_PIN, OUTPUT); // setup: set the LED pin as output
    while (1) {  // endless loop
        digitalWrite(LED_PIN, HIGH); // turn LED on
        delay(1000); // wait for 1000ms
        digitalWrite(LED_PIN, LOW); // turn LED off
        delay(1000); // wait for 1000ms
    }
}
```

(*) Arduino Blink Tutorial: http://www.arduino.cc/en/Tutorial/Blink



Smaller embedded systems are typically designed as a “superloop” that runs on a bare-metal CPU, without any underlying operating system. This is also the most basic structure that all embedded programmers learn in the beginning of their careers.

For example, here you can see a superloop adapted from the basic Arduino Blink Tutorual. The code is structured as an endless “while (1)” loop, which turns an LED on, waits for 1000 ms, turns the LED off, and waits for another 1000ms. All this results in blinking the LED. The main characteristics of this approach is that the code often waits **in-line** for various conditions, for example a time delay. “In-line” means that the code won't proceed until the specified condition is met. Programming that way is called **sequential programming**.

The main problem with this sequential approach is that while waiting for one kind of event, the “superloop” is **unresponsive** to any other events, so it is difficult to add new events to the loop.

Of course, the loop can be modified to wait for ever shorter periods of time to check for various conditions more often. But adding new events to the loop becomes increasingly difficult and often causes an upheaval to the whole structure and timing of the entire loop.

RTOS Multithreading: Multiple "Superloops"

```
void thread_alarm() {          // RTOS thread routine
    pinMode(SW_PIN, INPUT);    // setup: set the Switch pin as input
    while (1) {                // endless loop
        if (digitalRead(SW_PIN) == HIGH) { // is the switch depressed?
           digitalWrite(ALARM_PIN, HIGH); // start the alarm
        }
        else {
           digitalWrite(ALARM_PIN, LOW);  // stop the alarm
        }
        RTOS_delay(100);
    }
}
```

```
void thread_blink() {          // RTOS thread routine
    pinMode(LED_PIN, OUTPUT); // setup: set pin as output
    while (1) {                // endless loop
        digitalWrite(LED_PIN, HIGH); // turn the LED on
        RTOS_delay(1000);      // wait for 1000ms
        digitalWrite(LED_PIN, LOW);  // turn the LED off
        RTOS_delay(1000);      // wait for 1000ms
    }
}
```

Quantum® LeaPs Modern Embedded Software

state-machine.com

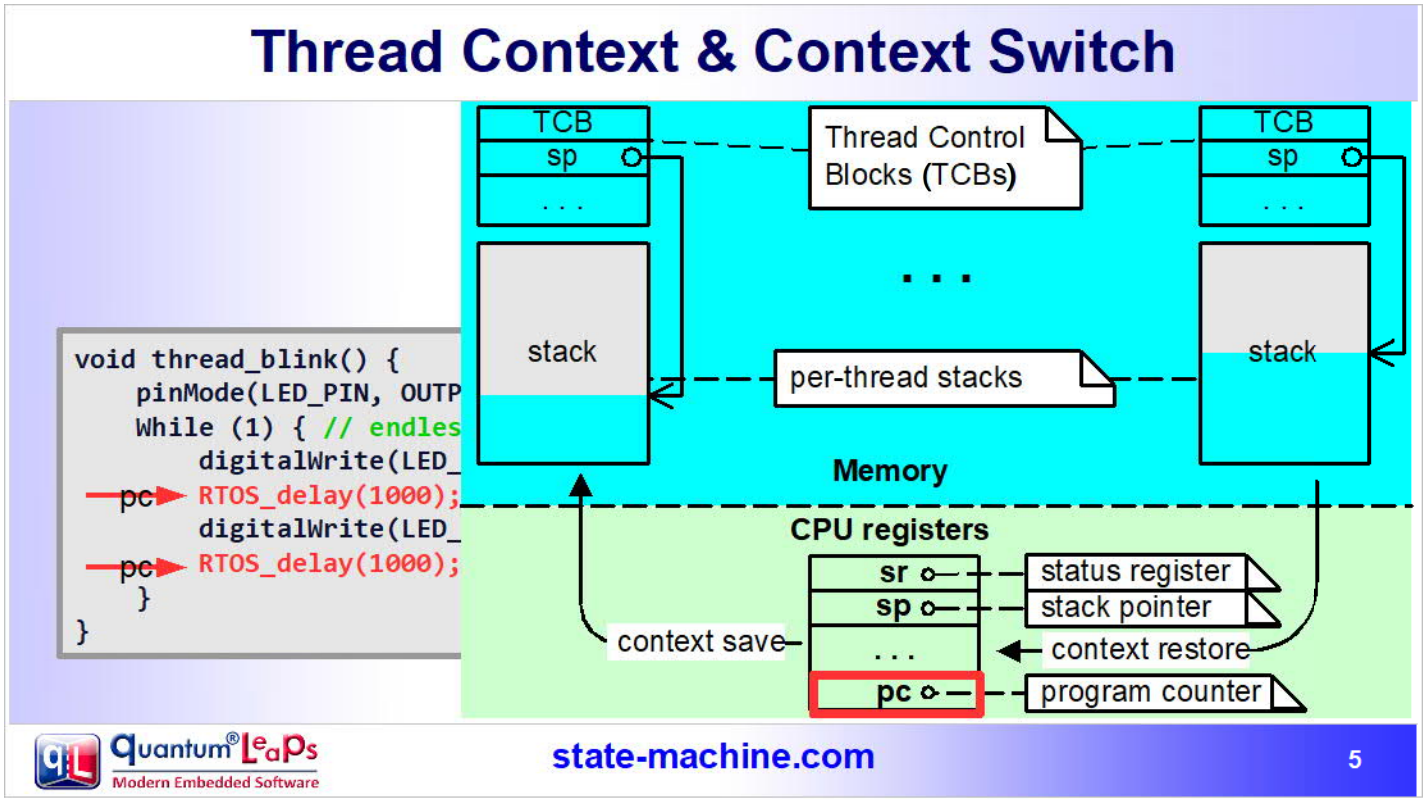
4

An obvious solution to the unresponsiveness of a single superloop is to allow multiple superloops to run on the same CPU. Multiple superloops can wait for multiple events in parallel.

And this is exactly what a Real-Time Operating System (RTOS) allows you to do. Through the process of scheduling and switching the CPU, which is called *multitasking* or *multithreading,* an RTOS allows you to run multiple superloops on the same CPU. The main job of the RTOS is to create an illusion that each superloop, called now a **thread**, has the entire CPU all to itself.

For example, here you have two threads: one for blinking an LED and another for sounding an alarm when a button is pressed.

As you can see, the code for the Blink thread is really identical to the Blink superloop, so it is also sequential and structured as an endless while(1) loop. The only difference now is that instead of the polling delay() function, you use RTOS_delay(), which is very different internally, but from the programming point of view it performs exactly the same function.

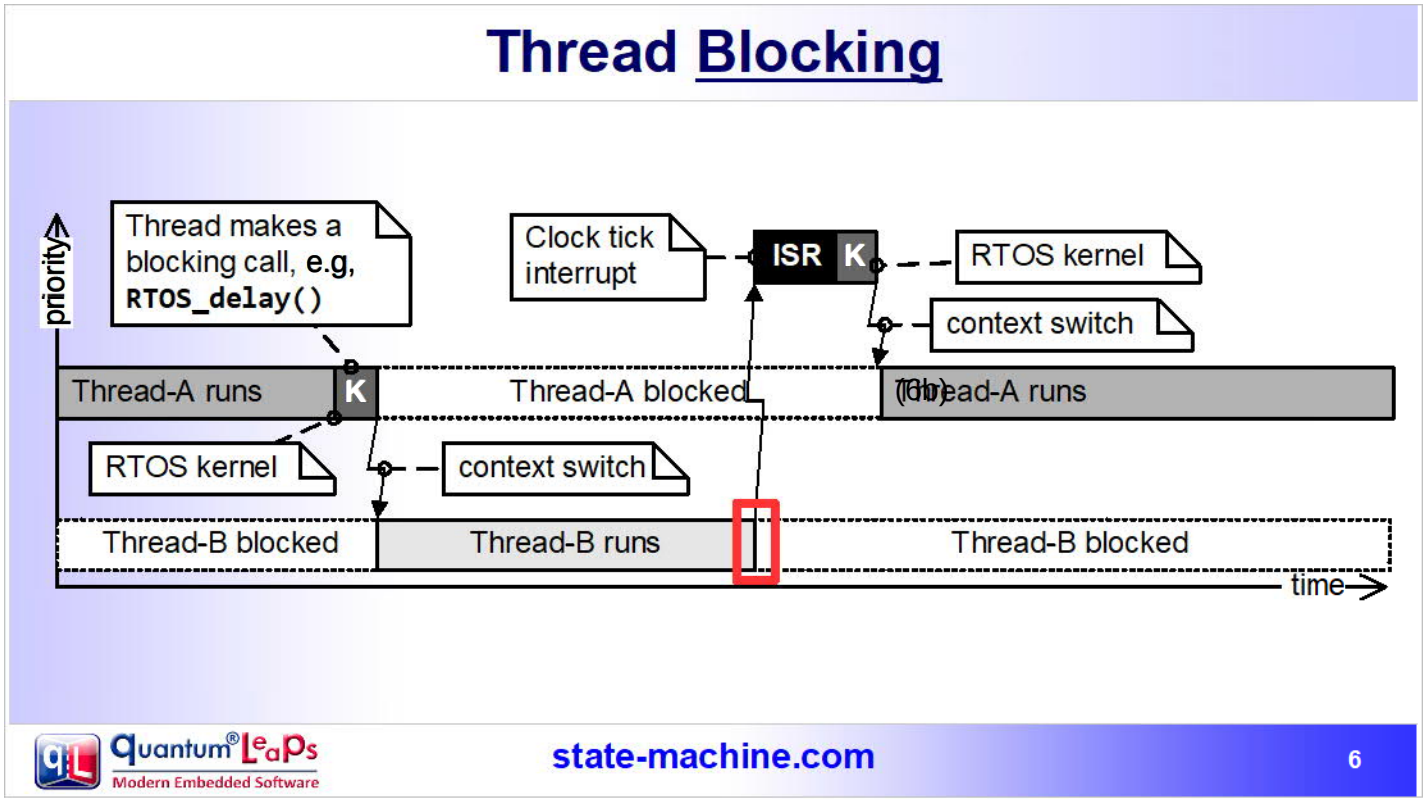


How does the RTOS achieve multitasking? Well, each thread in an RTOS has a dedicated private **context** in RAM, consisting of a private stack area and a thread-control-block (TCB).

The context for every thread must be that big, because in a **sequential code** like that, the context must remember the whole nested function call tree and the exact place in the code, that is, the program counter. For example, in the Blink thread, the contexts of the two calls to RTOS_delay(), will have identical call stack, but will differ in the values of the program counter (PC).

Every time a thread makes a blocking call, such as RTOS_delay() the RTOS saves CPU registers on that thread's stack and updates it's TCB. The RTOS then finds the next thread to run in the process called *scheduling*. Finally, the RTOS restores the CPU registers from that next thread's stack. At this point the next thread resumes the execution and becomes the current thread.

The whole context-switch process is typically coded in CPU-specific assembly language, and takes a few microseconds to complete.



For example, a call to RTOS_delay() from Thread-A results in a context switch to Thread-B.

Thread-A switched “away” in this process stops consuming any CPU cycles, so it becomes **efficiently blocked**.

Instead, the CPU cycles that a primitive superloop would waste in a polling loop go to the other Thread-B that has something useful to do.

Please note that in a single CPU system, for any given thread to run, all other threads must be blocked. This means that blocking is quite fundamental to multitasking.

Finally, note that a context switch can be also triggered by an interrupt, which is asynchronous to the execution of threads. For example, unblocking of Thread-A and blocking of Thread-B, can be triggered by the system clock tick interrupt. An RTOS kernel in which interrupts can trigger context switch is called a **preemptive** RTOS.

RTOS Benefits

1) Divide and conquer strategy

→ Multiple threads are easier to develop than one “kitchen sink” superloop

2) More efficient CPU use

→ Threads that are efficiently blocked don't consume CPU cycles

3) Threads can be decoupled in the time domain

→ Under a preemptive, priority-based scheduler, changes in low-priority threads have no impact on the timing of high-priority threads (Rate Monotonic Analysis (RMA))

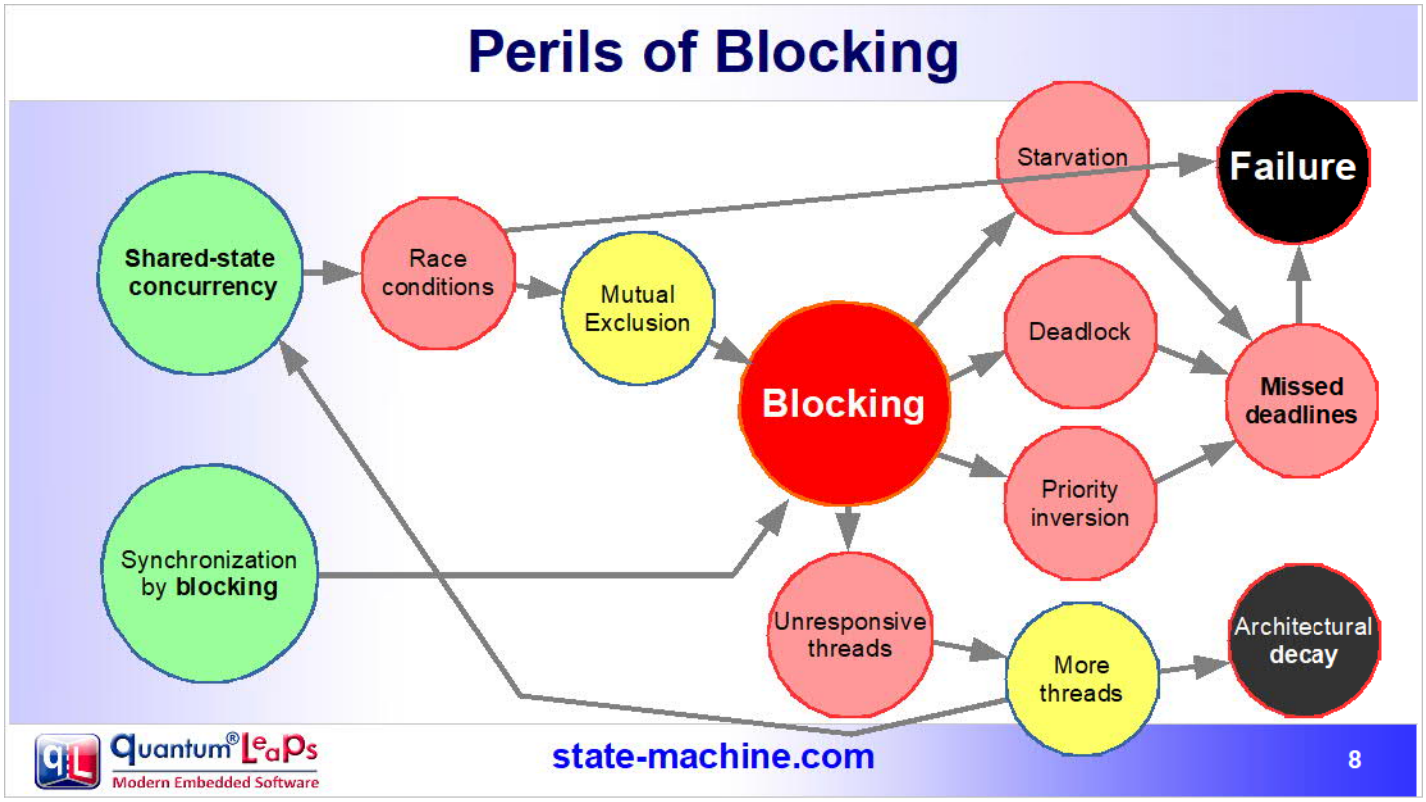
Quantum® LeaPs Modern Embedded Software state-machine.com 7

Compared to a “superloop”, an RTOS kernel brings a number of very important benefits:

1. It provides a “divide-and-conquer” strategy, because it allows you to partition your application into multiple threads.
→ Each one of these threads is much easier to develop and maintain than one “kitchen sink” superloop

2. Threads that wait for events are efficiently blocked and don't consume CPU cycles. This is in contrast to wasteful polling loops often used in the superloop.

3. Certain schedulers, most notably *preemptive, priority-based schedulers,* can execute your applications such that the timing of high-priority threads can be insensitive to changes in low-priority threads (if the threads don't share resources). This is because under these conditions, high-priority threads can always preempt lower-priority threads. This enables you to apply formal timing analysis methods, such as Rate Monotonic Analysis (**RMA**), which can guarantee that (under certain conditions) all your higher-priority threads will meet their deadlines.



Multiple, dedicated threads are great and bring a quantum leap of improvement compared to a kitchen-sink “superloop”, but the problems begin when the threads need to synchronize and communicate with each other.

Generally, threads synchronize their activities by blocking and unblocking each other by means of such mechanisms as semaphores, event flags, or message queues. But this causes additional blocking, which reduces the responsiveness of the existing threads and forces developers to create more threads, which ultimately leads to **architectural decay**

Also any form of shared-state communication among threads requires applying **mutual exclusion** to avoid ***race conditions*** around the shared resources. But using *mutual exclusion* leads to additional blocking of threads, which can cause the whole slew of second-order problems, such as *thread starvation*, *deadlock*, or *priority inversion*. Any of these problems might lead to missed deadlines, which means *failure* in a real-time system.

Speaking of failures, they are typically subtle, intermittent, and notoriously hard to reproduce, isolate, and fix. Such problems are the worst kind you can possibly have in your software.

Best Practices of Concurrent Programming(*)

- **Don't block** inside your code
  - → Communicate and synchronize threads **asynchronously** via **event objects**
- **Don't share** data or resources among threads
  - → Keep data isolated and bound to threads (strict **encapsulation**)
- Structure your threads as "message pumps"

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

Quantum® LeaPs Modern Embedded Software state-machine.com 9

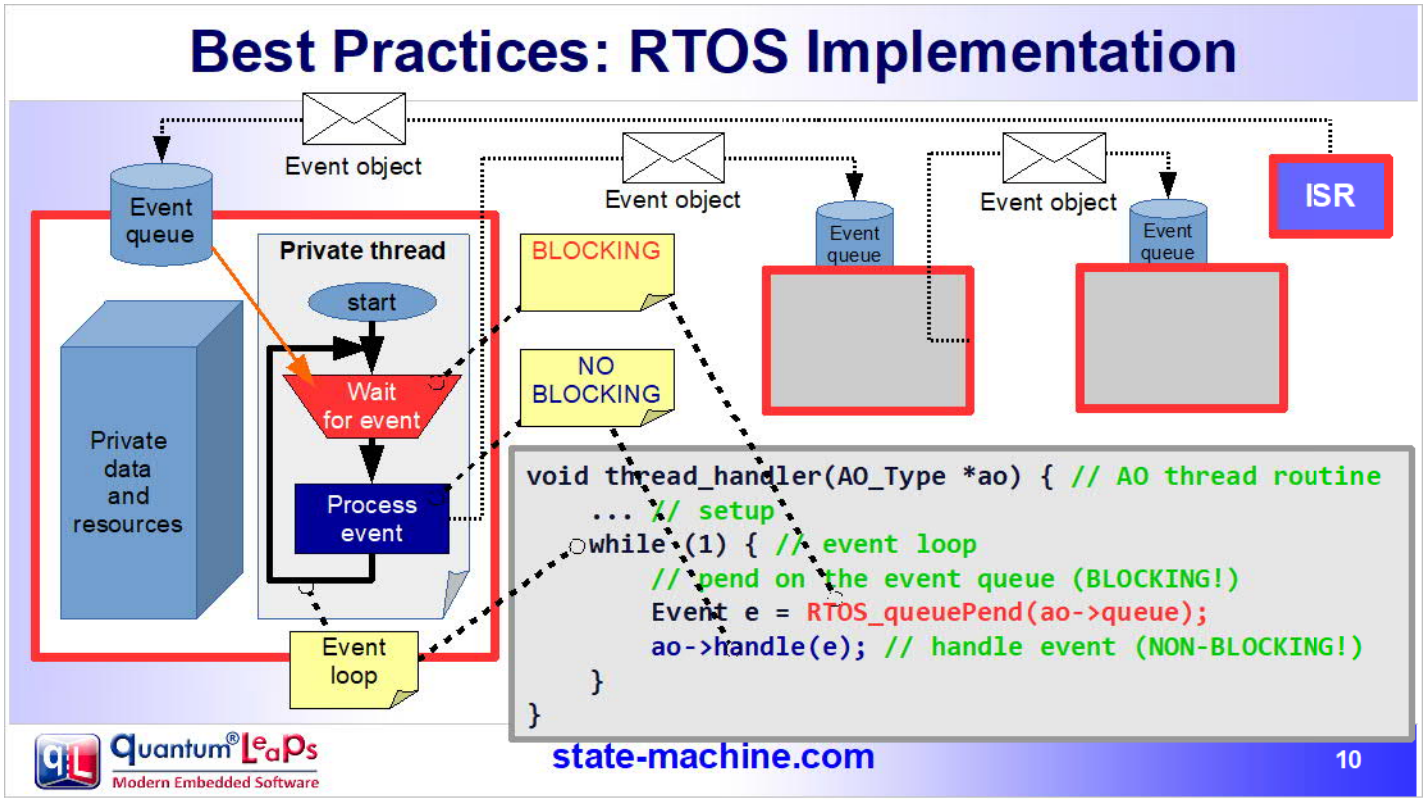
For all these reasons, expert real-time programmers have learned to be very weary of **blocking**. Instead, experts came up with the following **best practices**:

1. Don't block inside your code. Instead communicate among threads **asynchronously** via **event** objects
→ This makes threads run truly independently, without **blocking** on each other

2. Don't share any data or resources among threads. Keep data and resources **encapsulated** inside threads ("share-nothing" principle) and instead use **events** to share information

3. Organize your threads as "message pumps" (event queue + event loop)

In other words, these best practices combine multithreading with **event-driven programming.**



Perhaps the easiest way to understand these best practices is to see how they can be implemented with a traditional RTOS:

- You start off by defining your own basic **event** data type, which carries the event *signal* and event parameters. For example, an event with signal ADC_COMPLETE tells you that ADC conversion has just completed, and in the event parameter it can carry the numeric value produced by the ADC.
- Each thread owns an event queue (or a message queue) capable of storing your event objects
- The threads communicate and synchronize **only** by posting events to their queues. Specifically, the threads are not allowed to share any data or resources, which are private and **strictly encapsulated**.
- Event posting is **asynchronous** meaning that threads don't wait until the event is processed. They just drop it into a queue and move on.
- The thread code is organized as a “message pump”
  → a thread blocks **only** when its queue is empty, and does **not block** anywhere in the event-handler code
  → such a “message pump” naturally implements the **Run-to-Completion** event processing semantics, which simply means that the thread must necessarily finish processing of one event before it can start processing the next event. This eliminates any concurrency hazards within a thread itself.

Active Object (Actor) Design Pattern

- **Active Objects (Actors)** are event-driven, strictly **encapsulated** software objects running in their **own threads** and communicating **asynchronously** by means of **events**.
- Not a novelty. Carl Hewitt's actors 1970s. ROOM actors 1990s.
- Adapted from ROOM into UML as **active objects**

  → ROOM actors and UML active objects use **hierarchical state machines** (UML statecharts) to specify the *behavior* of event-driven active objects.

Quantum® LeaPs Modern Embedded Software state-machine.com 11

Of course, all these best practices and specific implementation guidelines establish a *design pattern*, which is known as the **Active Object** or **Actor** design pattern.

As most good ideas in software, the concept of autonomous software objects communicating by message passing is not new. It can be traced back to the 1970s when Carl Hewitt at MIT came up with Actors.

In the 1990s, methodologies like Real-Time Object-Oriented Modeling (ROOM) adapted actors for real-time computing.

More recently, UML introduced the concept of Active Objects that are essentially synonymous with the ROOM actors. Both variants use **hierarchical state machines** (UML statecharts) to model the internal behavior of active objects. I'll talk about state machines a bit later.

Active Object Framework

- Implement the Active Object pattern as a **framework**

```
void thread_handler(AO_Type *ao) { // AO thread routine
    ... // setup
    while (1) { // event loop
        // pend on the event queue (BLOCKING!)
        Event e = RTOS_queuePend(ao->queue);
        ao->handle(e); // handle event (NON-BLOCKING!)
    }
}
```

- **Inversion of control** (main difference from RTOS)
  - → *automates* and *enforces* the best practices (**safer** design)
  - → brings **conceptual integrity** and consistency to the applications

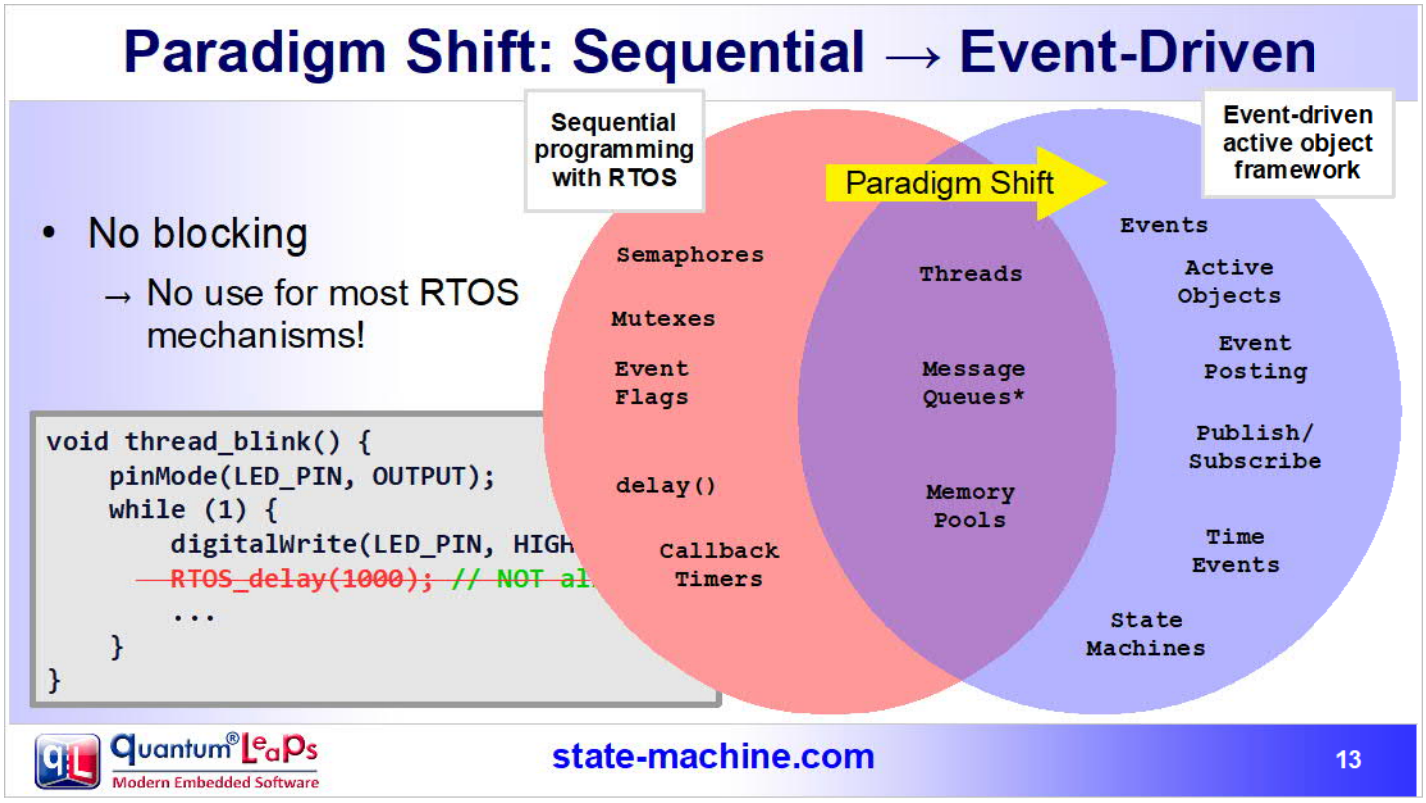
Quantum® LeaPs Modern Embedded Software state-machine.com 12

In an earlier slide, you saw how to implement the Active Object pattern manually on top of a conventional RTOS. But an even better way is to implement this pattern as a software **framework**, because a framework is the best known method to capture and reuse a software **architecture**.

In fact, such a rudimentary framework already started to emerge in the thread handler for the active objects. If you understand the call ao→handle() as being virtual, that is, dependent on the type of the active object, the whole thread handler will become generic and can be considered a part of an active object framework, instead of being repeated in each specific application.

This also illustrates the most important characteristics of a framework called **inversion of control**. When you use an RTOS, you write the main body of each thread and you call the code from the RTOS, such as RTOS_queuePend(). In contrast, when you use a framework, you reuse the **architecture**, such as the message pump here, and write the code that *it* calls.

The **inversion of control** is very characteristic to 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.

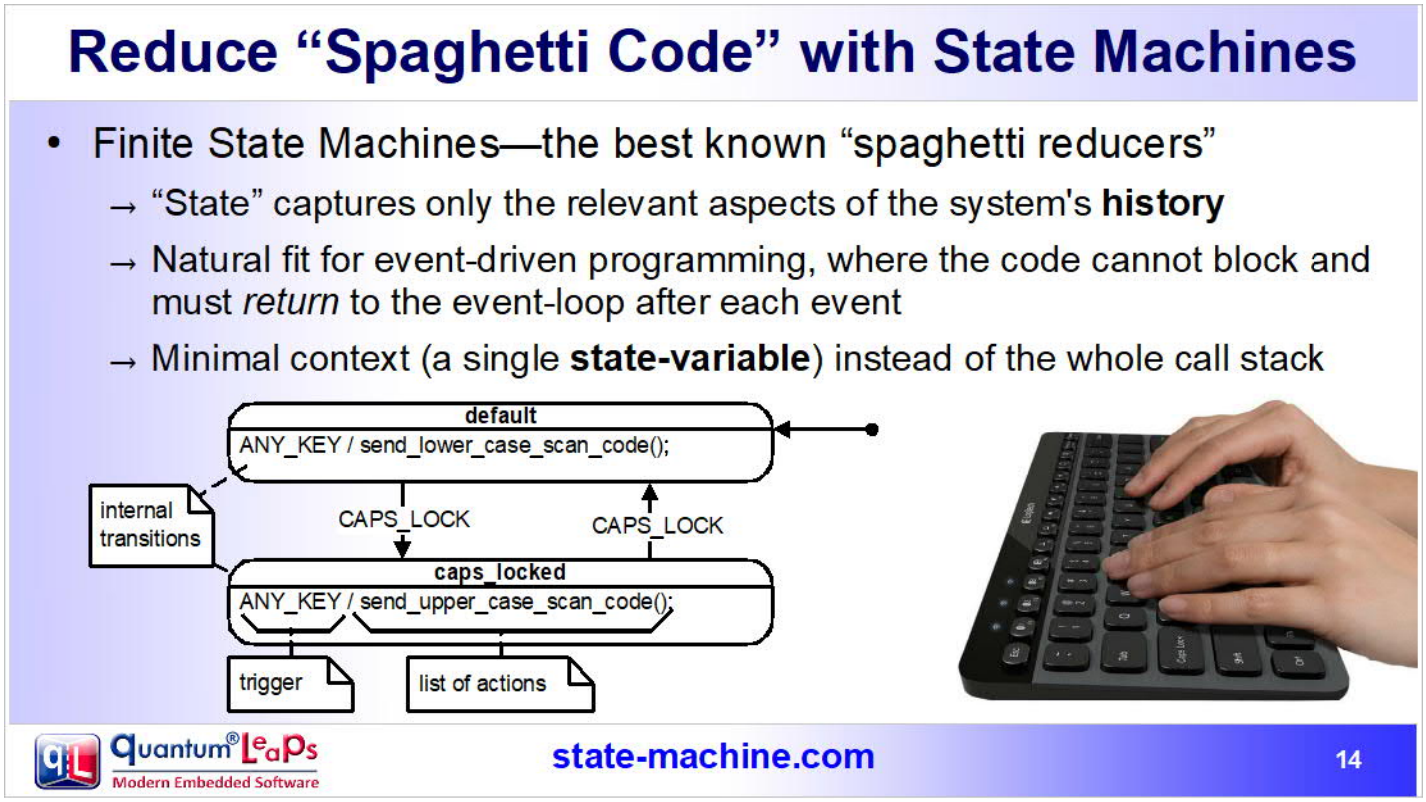


When you start building an Active Object framework, you will see that it will require a paradigm shift.

On one hand, most RTOS mechanisms based on blocking will be useless, or outright **harmful**, if applied inside active objects. For example, you will have no use for the blocking delay() function, semaphores, or other similar mechanisms.

On the other hand hand, a conventional RTOS will not provide much support for event-driven active objects, which you will then need to create yourself. For example, instead of the delay() function, you will need to create an event-driven mechanism based on Time Events.

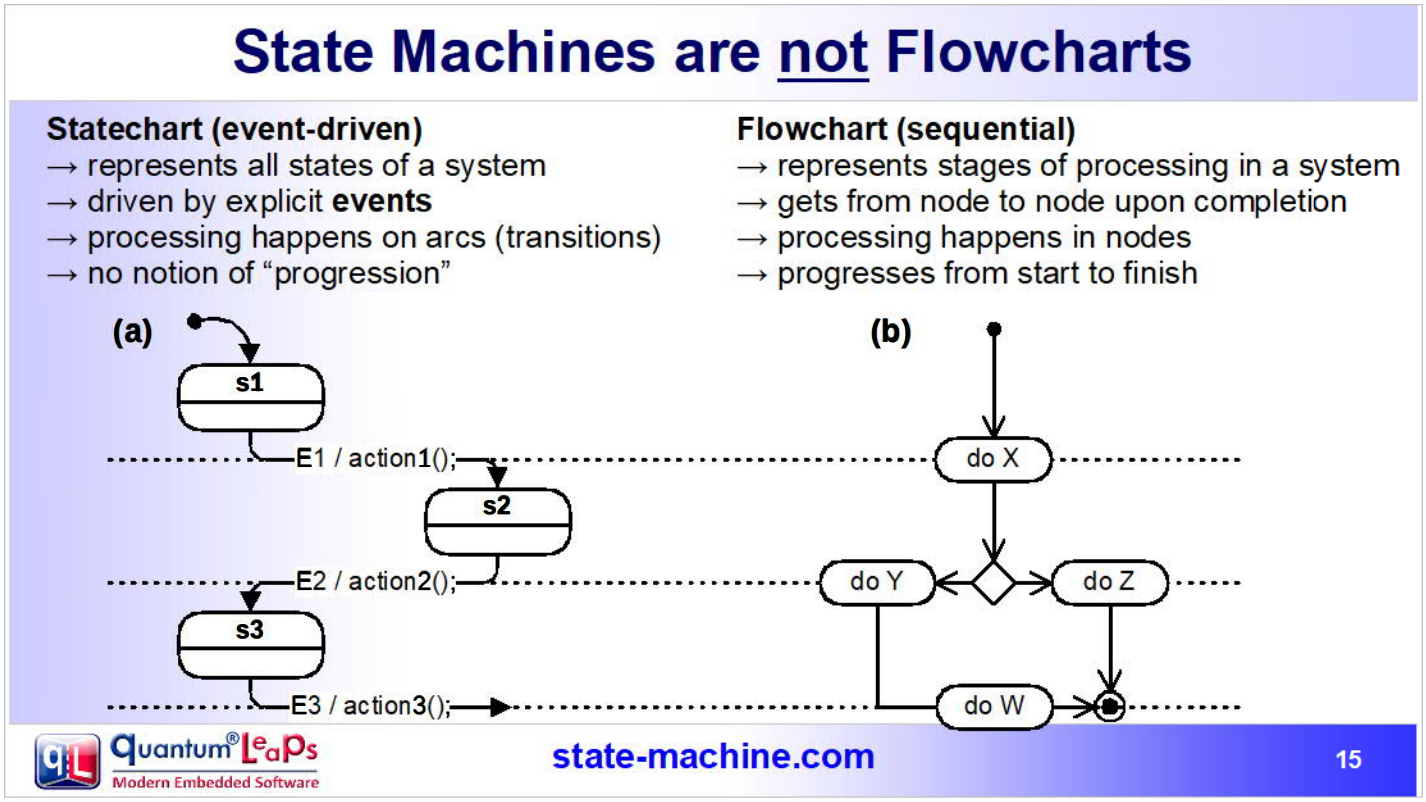
This is all because a conventional RTOS is designed for the **sequential programming** model, where you **block** and wait in-line for the occurrence of an event, whereas an Active Object framework implements an **event-driven paradigm** based on run-to-completion event processing *without* blocking.



But programming without blocking requires you to respond to events quickly and always *return* to the "message pump". This means that you cannot store the context of your computation (the relevant **history** of handled events) on the call stack and in the program counter, as the sequential code could do. Instead, you would probably store the context (event history) in static variables. A typical process looks as follows: Most event handlers start off pretty simple, but as new events are grafted on, developers add more and more flags and variables. Then they create increasingly convoluted IF-THEN-ELSE logic to test their flags and variables, until nobody can tell what's going on in the code. At this point, the code becomes "spaghetti".

And here is where Finite State Machines come in. They are the best known "spaghetti reducers". The concept of "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" where it generates lower-case characters or in "caps_locked" state, where it generates upper-case characters. The relevant system history is pressing the CAPS_LOCK event. Pressing other keys is irrelevant.

State machines are a natural fit for event-driven programming. They are exactly designed to process each event quickly and *return to the caller.* The context of the system between calls is represented by the single *state-variable*, which is much more efficient than improvising with multitude of flags.



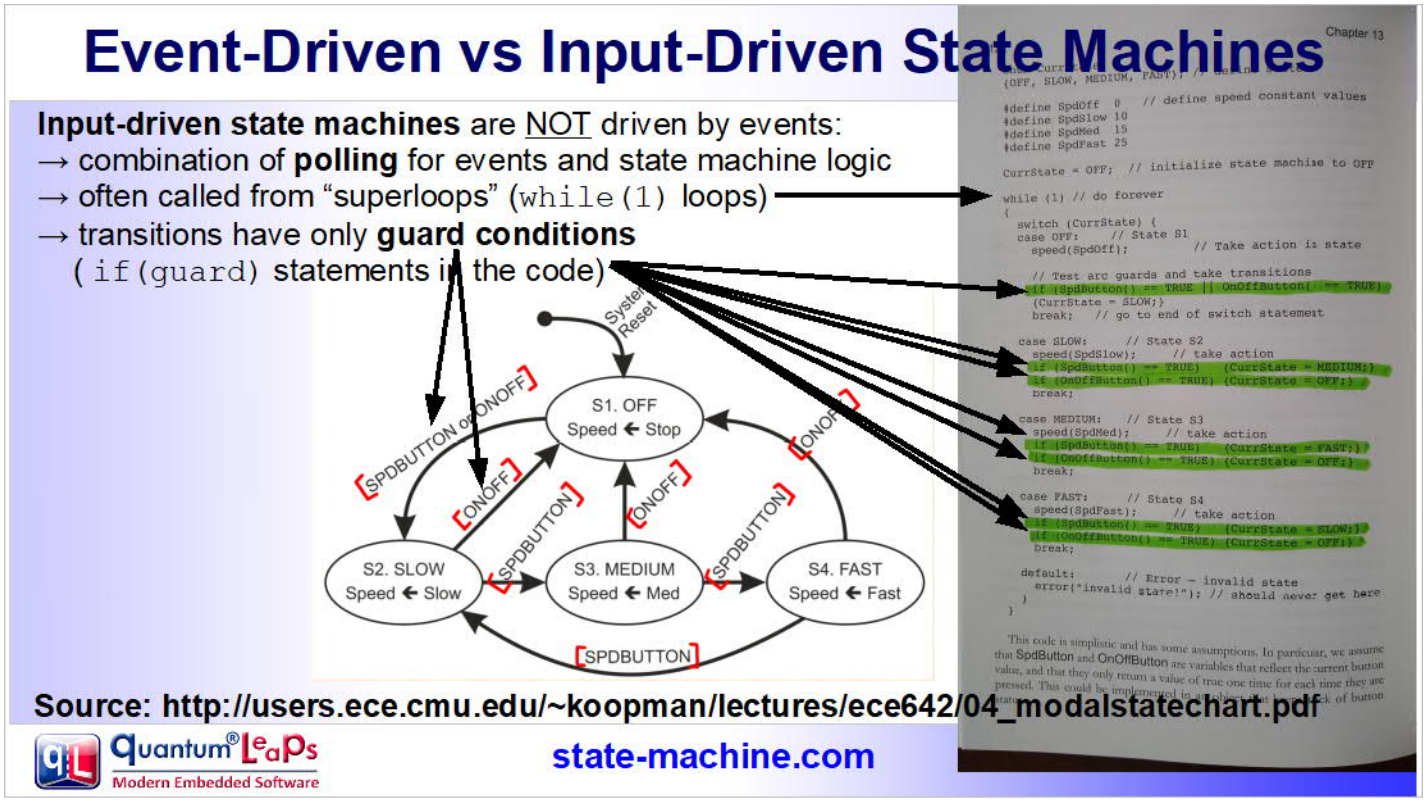
Newcomers to the state machine formalism often confuse state diagrams (statecharts) with flowcharts. This is another aspect of the paradigm shift from sequential to event-driven programming, so let me quickly explain the difference.

So, first, a statechart represent states or modes of the system. In contrast, a flowchart represents stages in processinig.

A statechart always needs **events** to perform any actions and possibly change state (execute transitions).

A flowchart doesn't need events. It progresses 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.

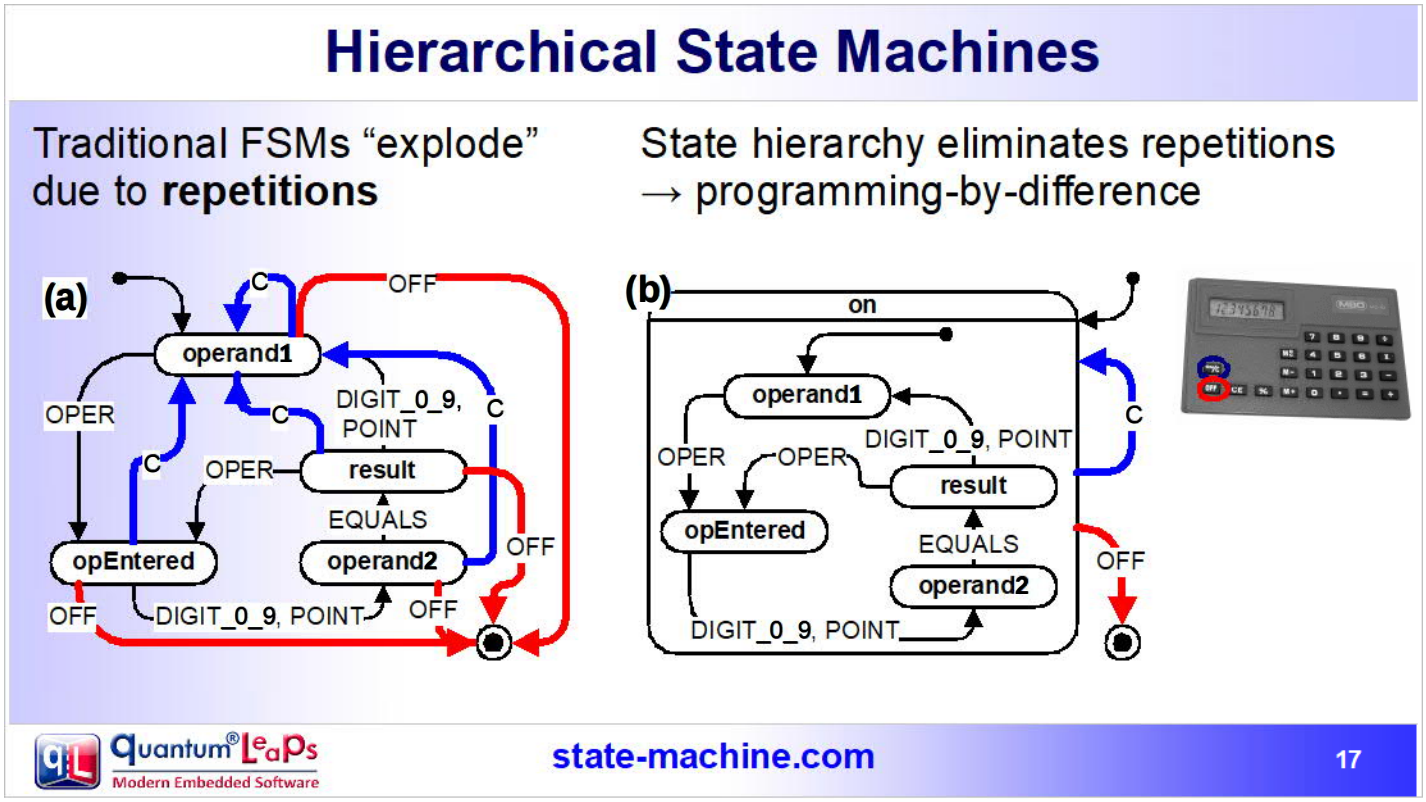


Much of the state machine examples posted online, in various books, and in the existing code pertain to **input-driven state machines**, as opposed to truly event-driven state machines introduced earlier.

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 test various (guard) conditions in each state and only after discovering an event, they process it.

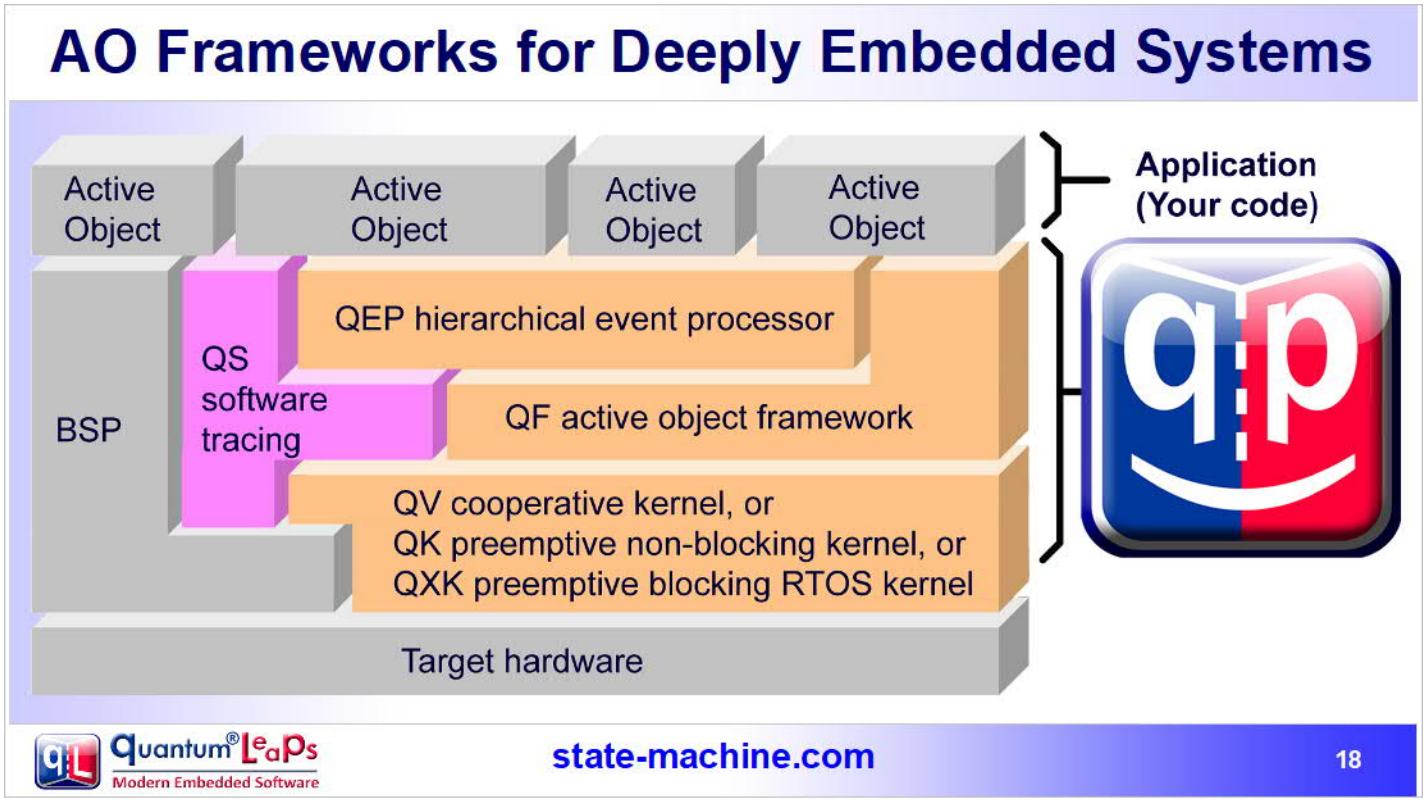
In the diagram, you can easily recognize input-driven state machines by the fact that state transitions are **NOT** labeled by events, but rather by guard conditions. The brackets around those guard conditions, which are required by the standard UML state machine notation, are often (unfortunately) omitted, 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 state machines are that they might miss events (if sampling is too slow) or recognize events in different order, depending on the timing of sampling, over each you have little control. They are also wasteful, as they need to run all the time. Finally, it is impossible to apply concepts of hierarchical event processing, because there are no explicit events.



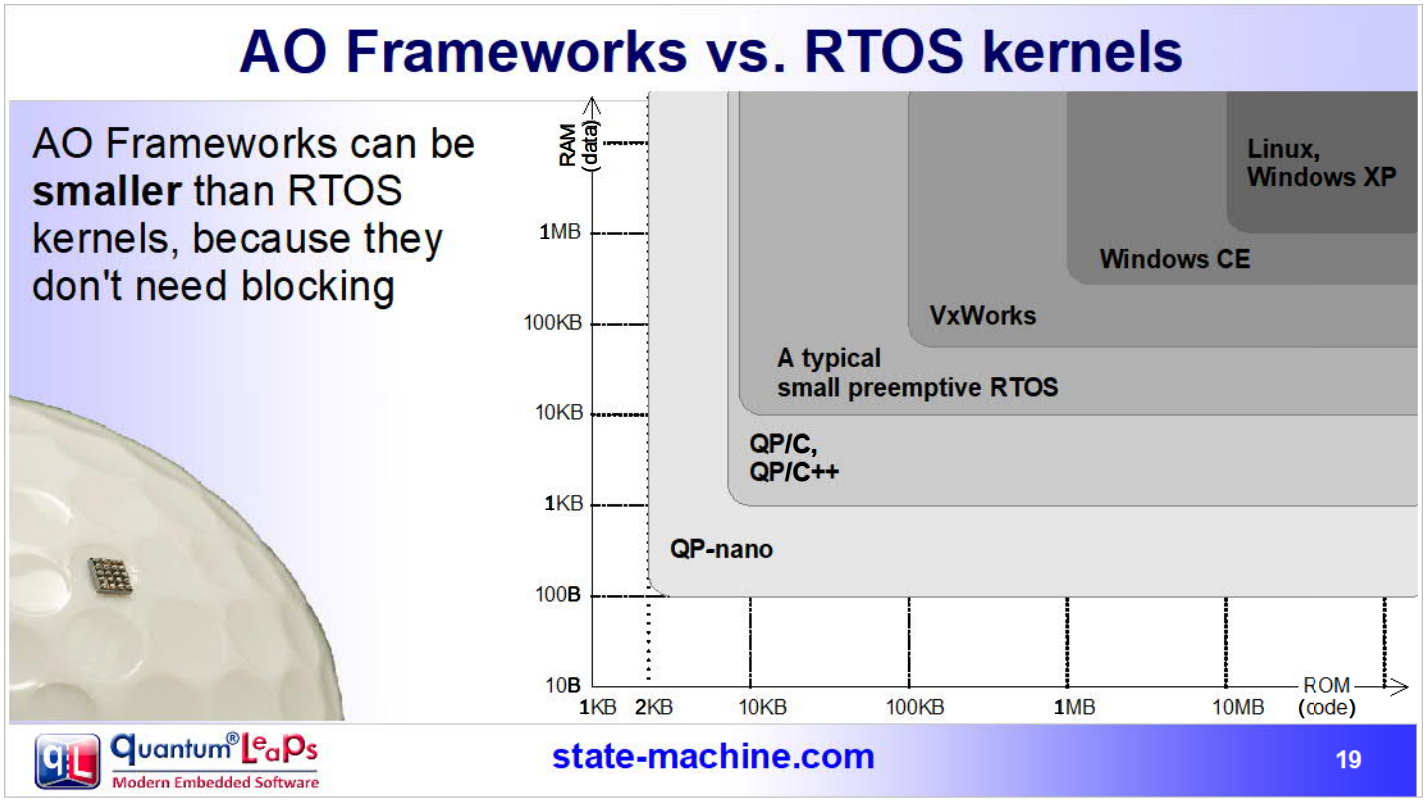
Traditional FSMs have a major shortcoming known as "state and transition explosion". For example, if you try to represent the behavior of a simple pocket calculator with a traditional FSM, you'll notice that many events (e.g., the *Clear* or *Off* button presses) are handled identically in many states. 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 address 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, such as OFF, 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).



So, how do you bring it all together, in a consistent active object framework? Well, as it turns out most of such frameworks end up with a similar layered structure. Here, for example, is the structure of a minimal active object framework for RTE systems called QP:

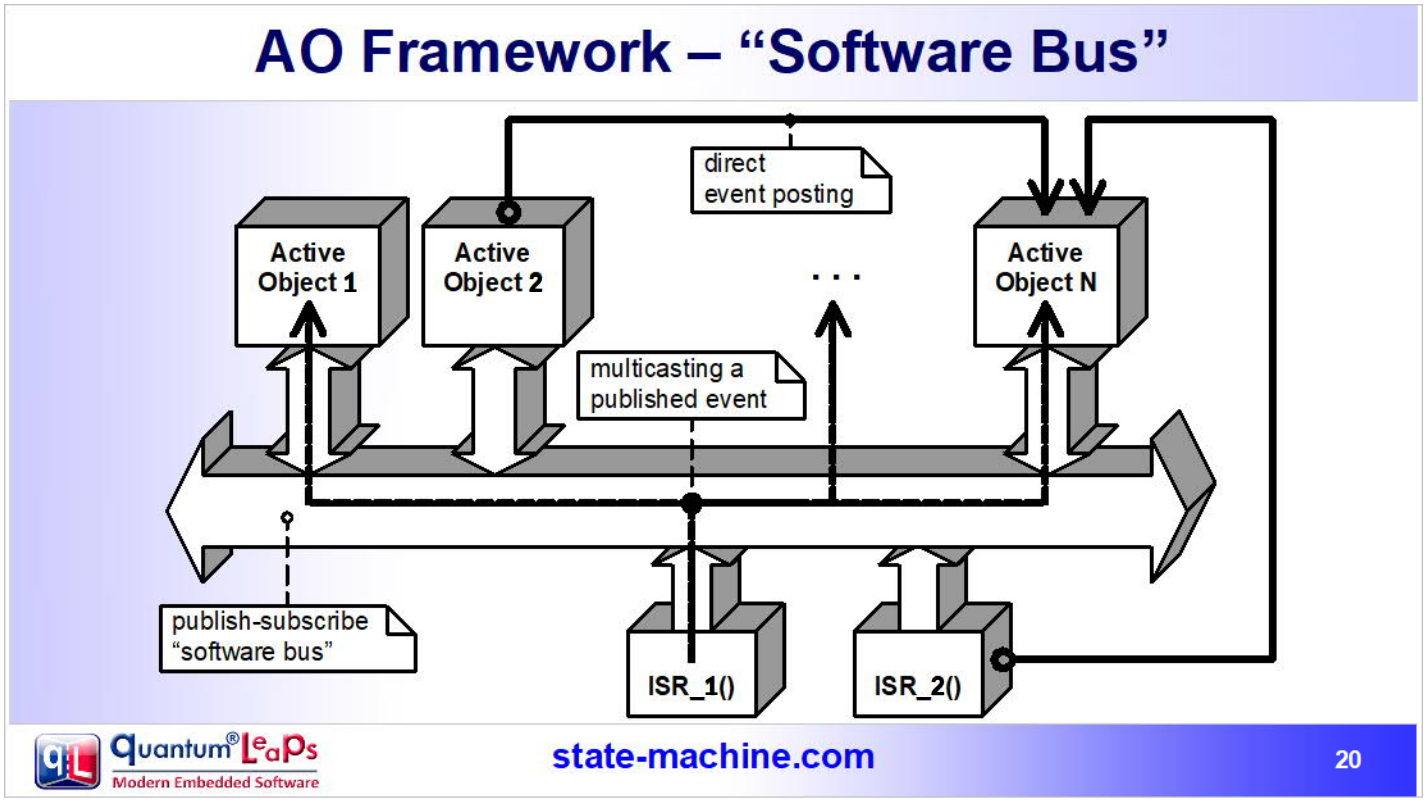
- The Target hardware sits at the bottom.
- The Board Support Package (BSP) above it provides access to the board-specific features, such as the peripherals.
- The real-time kernel (QV, QK, QXK, or a conventional 3rd-party RTOS) provides the foundation for multithreading, meaning the specific scheduling policy and context-switching.
- The event-driven framework (QF) supplies the event-driven infrastructure for executing active objects and ensuring **thread-safe** event 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. Developing applications consists mostly of **elaborating the hierarchical state machines** of the active objects.
- Finally, QS software tracing system provides live monitoring of applications for testing, troubleshooting, and optimizing. This component takes advantage of the fact that virtually all system interactions funnel through the framework, so instrumenting this small piece of code provides unprecedented visibility into the applications.



In the resource-constrained embedded systems, the biggest concern has always been about the size and efficiency of Active Object (Actor) frameworks, especially that commercial frameworks of this type 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.

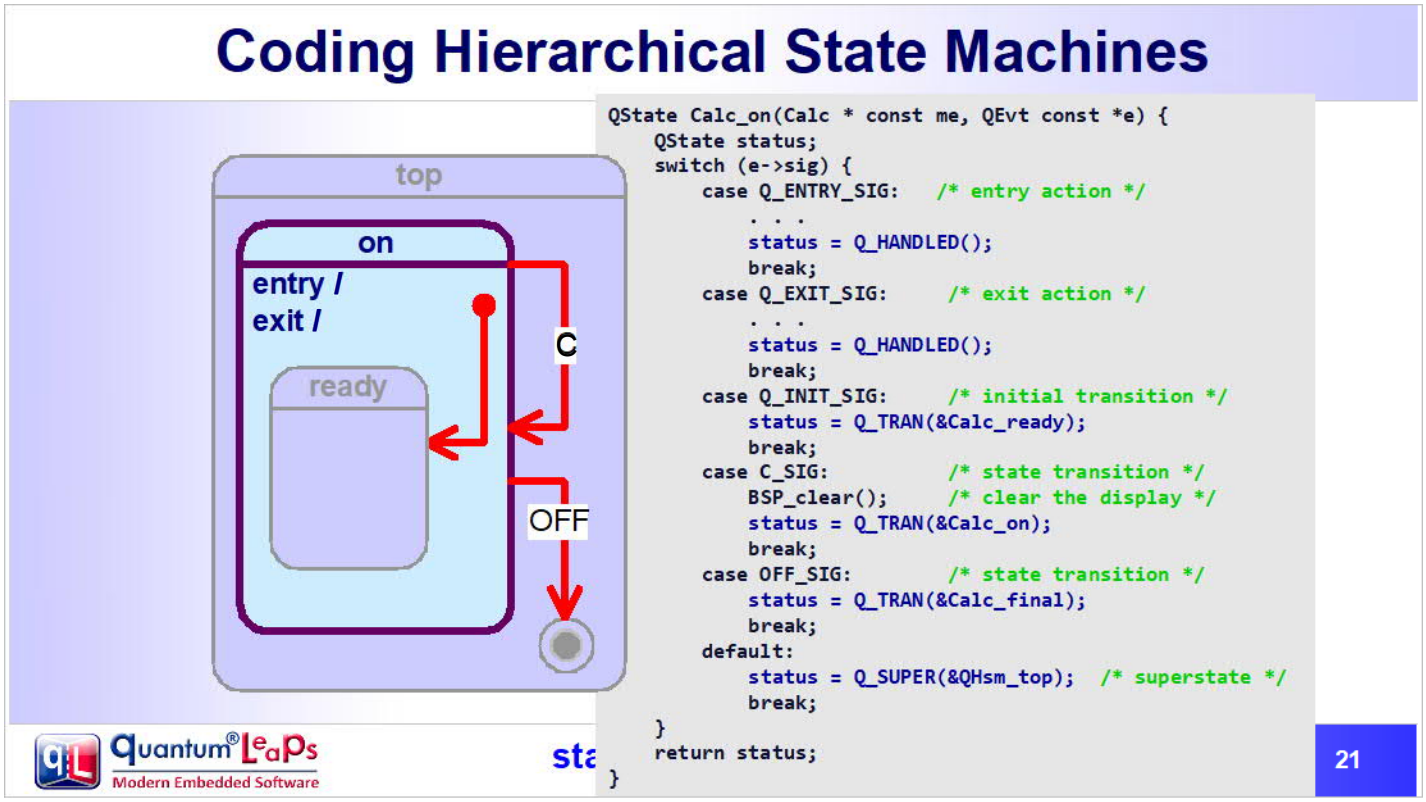
For example, here you see a comparison of RAM (data) and ROM (code) requirements of a typical application for various RTOSes and the QP active object frameworks. As you can see, the AO frameworks require significantly less RAM and somewhat less ROM, mostly because the frameworks do not need per-thread stacks. All these characteristics make event-driven Active Objects a perfect fit for single-chip microcontrollers (MCUs).



Even though an AO framework can be smaller than a conventional blocking RTOS kernel, it offers a higher level of abstraction.

For example, you can view the framework as an event-driven "software bus", into which active objects plug in. The framework performs the heavy lifting of **thread-safe** event delivery, so you—the application developer—don't need to worry about concurrency issues.

Most frameworks of this type support **direct event posting**. But some frameworks, including QP, also provide **publish-subscribe** event delivery, which reduces the coupling between the producers and consumers of events.

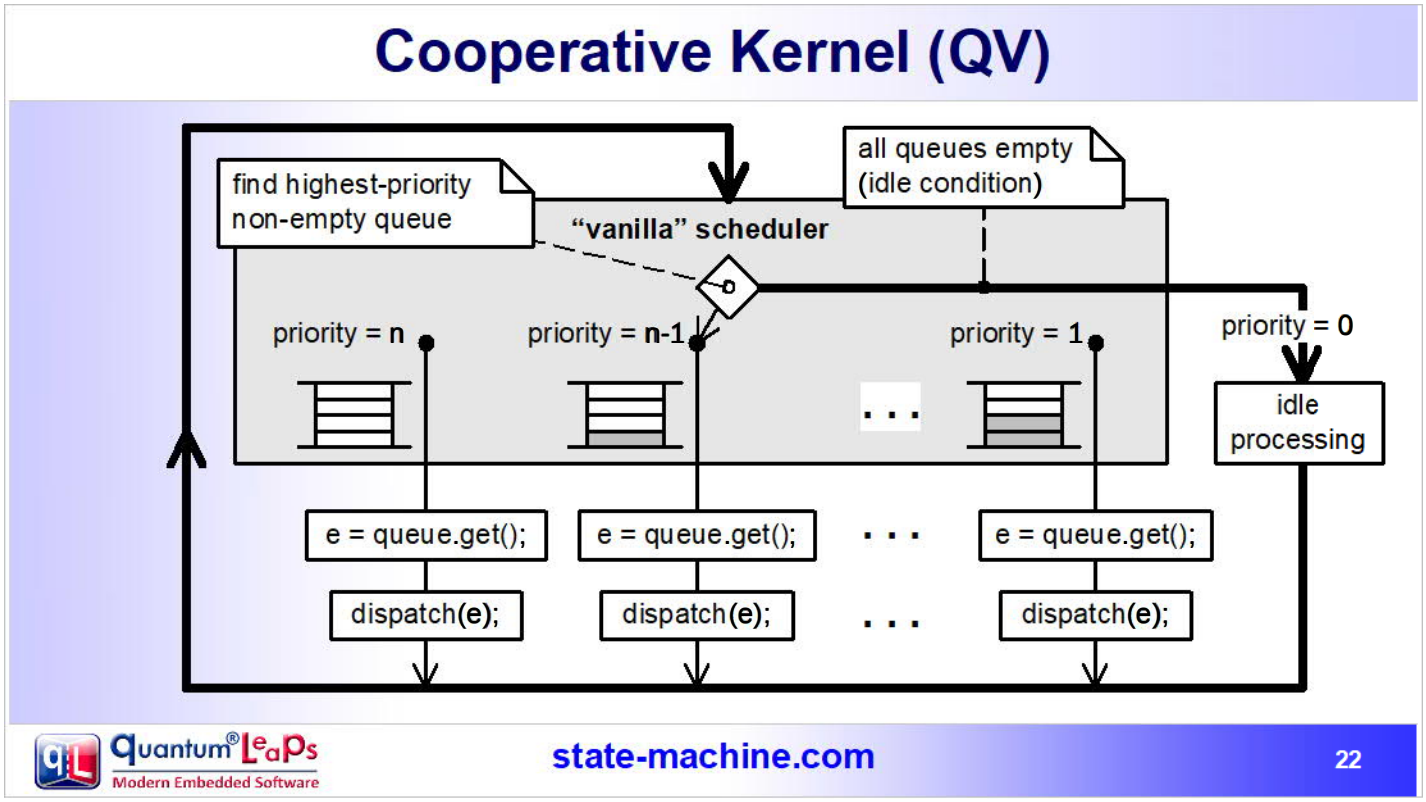


An AO framework raises the level of abstraction in another way as well. Instead of thinking in terms of individual C or C++ statements and IF-THEN-ELSE “spaghetti code”, you can now think in terms of state machine elements, such as states, transitions, entry/exit actions, initial transitions, and guards.

The provided code snippet shows how the QP/C framework allows you to map these elements to C. Every state, such as state “on”, maps to a state-handler function. This function is then called, possibly multiple times by the QEP hierarchical event processor to handle each event.

The elements that go into a state handler are: all transitions originating at the boundary of the state as well as the entry/exit actions of this state as well as the initial transition, if present. State nesting is implemented in the default case.

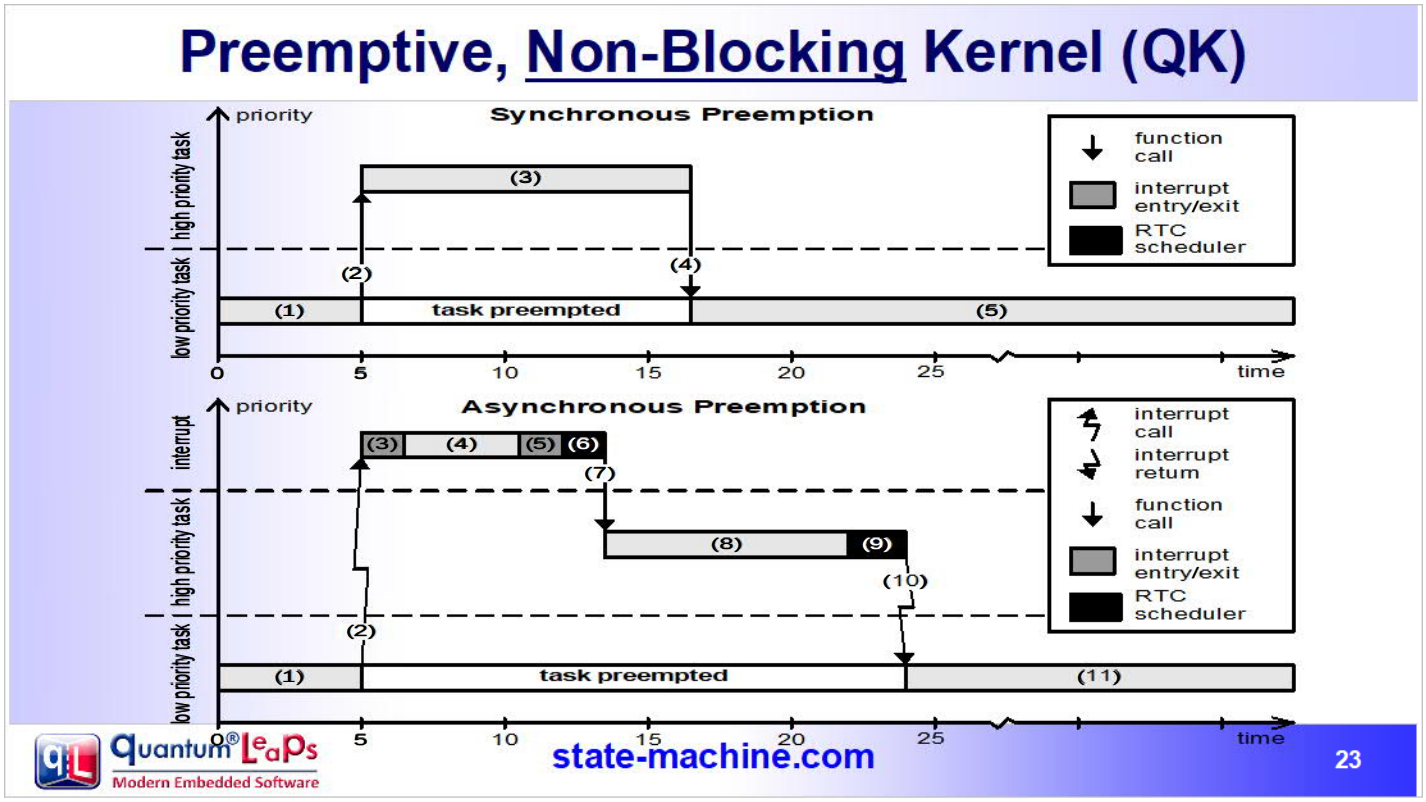
The hallmark of this implementation is **traceability**, which means that each state machine element is mapped to code unambiguously once and only once. Such traceability between design and code is required by most functional-safety standards.



An AO framework must guarantee the Run-To-Completion event processing in each active object, but it does **not** mean that the framework must monopolize the CPU while an event is processed. In fact, the underlying kernel can switch the CPU any number of times among active objects, provided that every RTC step eventually completes before the next event is processed. As long as the AOs don't share resources, there are no concurrency hazards. This means that an AO framework can work with a variety of different real-time kernels, including a traditional RTOS as you saw in the AO implementation based on an RTOS.

The QP framework can work with traditional RTOSes as well, but it also provides two lightweight built-in kernels that work differently

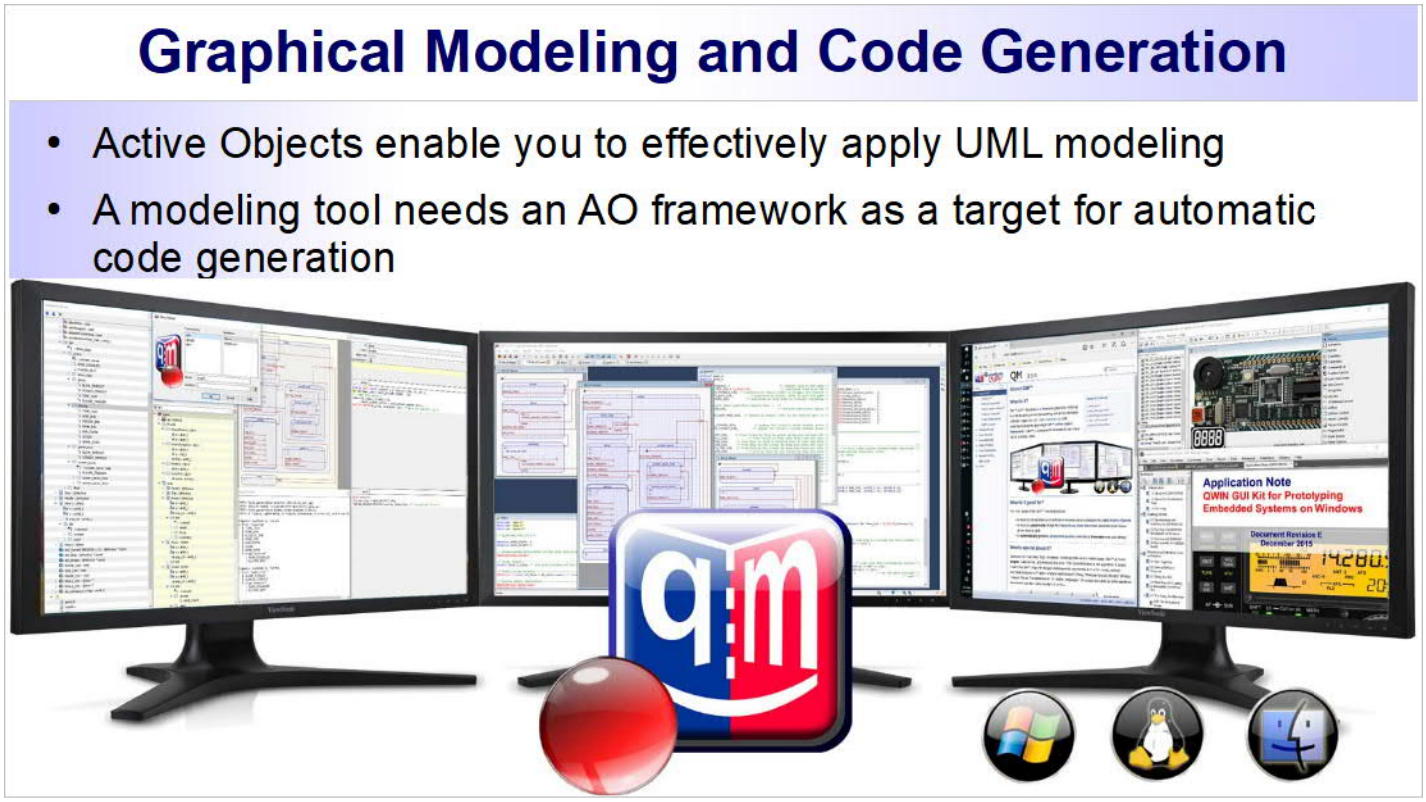
The simplest of the two is the priority-based cooperative QV kernel, which runs in a single main loop. The kernel always selects the highest-priority, not-empty event queue. Every event is always processed to completion in the main loop. If any new events are produced during the RTC step (e.g., by ISRs 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.



The second non-standard kernel provided in QP is really interesting. It is a very efficient, **preemptive**, priority-based, **non-blocking** 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 are one-shot tasks that 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 a single 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 (**RMA**) and can be used in hard real-time systems.



Active objects provide the sufficiently high-level of abstraction and the right level of abstraction to effectively apply **modeling**. This is in contrast to a traditional RTOS, which does not provide the right abstractions. You will not find threads, semaphores, or time delays in the standard UML. But you will find active objects, events, and hierarchical state machines.

An AO framework and a modeling tool beautifully complement each other. The framework benefits from a modeling tool to take full advantage of the very expressive graphical notation of state machines, which are the most constructive part of the UML.

On the other hand, a modeling tool needs a framework as a target for automatic code generation. A framework provides the necessary structure and well-defined “framework extension points” to generate code.

It should therefore come as no surprise that most modeling tools come with specialized AO frameworks. For example, the IBM Rhapsody tool comes with the OXF, IDF, and SXF frameworks. Similarly, the QM modeling tool from Quantum Leaps accompanies the QP frameworks.

Summary

- Experts use the Active Object design pattern instead of naked RTOS
- AO framework is an ideal fit for deeply embedded real-time systems
- AO framework requires a paradigm shift (sequential→event-driven)
- Compared to RTOS, AO framework opens new possibilities:
  - → Safer architecture and state-machine design method (functional safety)
  - → Simpler, more efficient kernels (lower-power applications)
  - → Easier unit testing and software tracing (V&V)
  - → Higher level of abstraction suitable for modeling and code generation
- **Welcome to the 21st century!**

Quantum® LeaPs Modern Embedded Software state-machine.com 25

Many embedded developers vastly underestimate the true costs and skills needed to program with an RTOS. The truth is that all traditional RTOS mechanisms for managing concurrency such as semaphores, mutexes, monitors, critical sections, and others based on **blocking**, are tricky to use and often lead to subtle bugs that are notoriously hard to reproduce, isolate, and fix.

But RTOS and superloop aren't the only game in town. **Actor** frameworks, such as Akka, are becoming all the rage in enterprise computing, but active object frameworks are an even better fit for deeply embedded programming.

After working with such frameworks for over 15 years, I believe that they represent a similar quantum leap of improvement over the RTOS, as the RTOS represents with respect to the “superloop”.

Active objects are closely related to such concepts as autonomous agents in artificial intelligence and behavior-based robotics.

Welcome to the 21st century!