

Modern Embedded Systems Programming: Beyond the RTOS

Miro Samek
Quantum Leaps, LLC



Quantum[®]LeaPs
Modern Embedded Software

state-machine.com



Presentation Outline

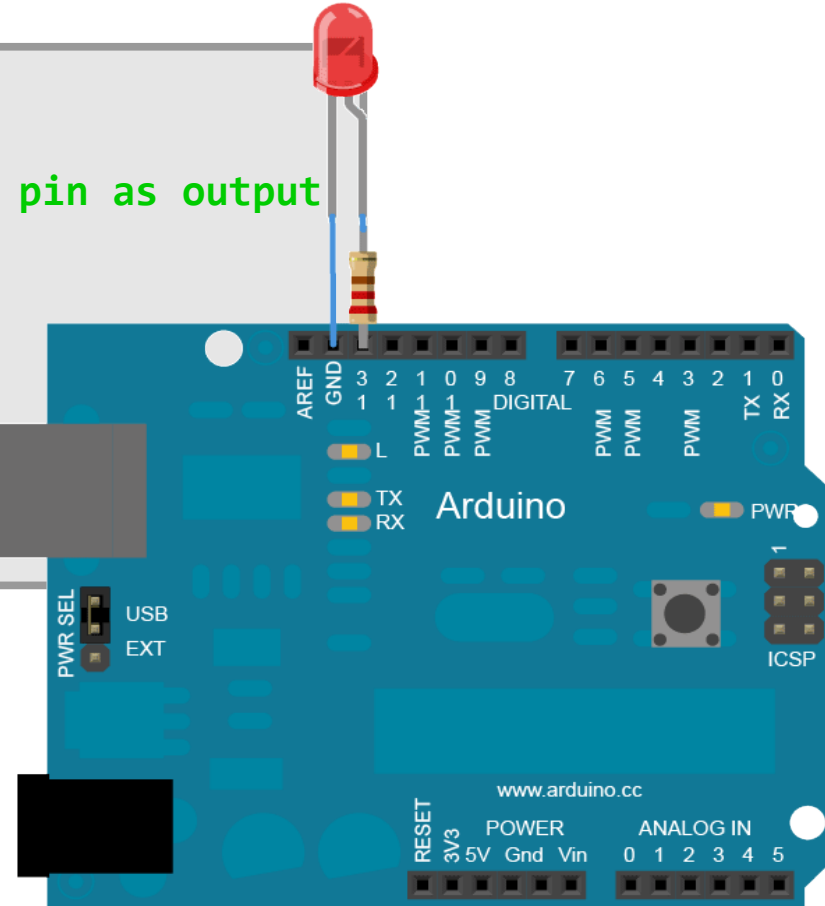
- A quick introduction to RTOS and the perils of blocking
 - Active objects
 - State machines
 - Active object frameworks for deeply embedded systems
 - Demonstrations | ~10 min
 - Q&A | ~10 min
- ~40 min



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>



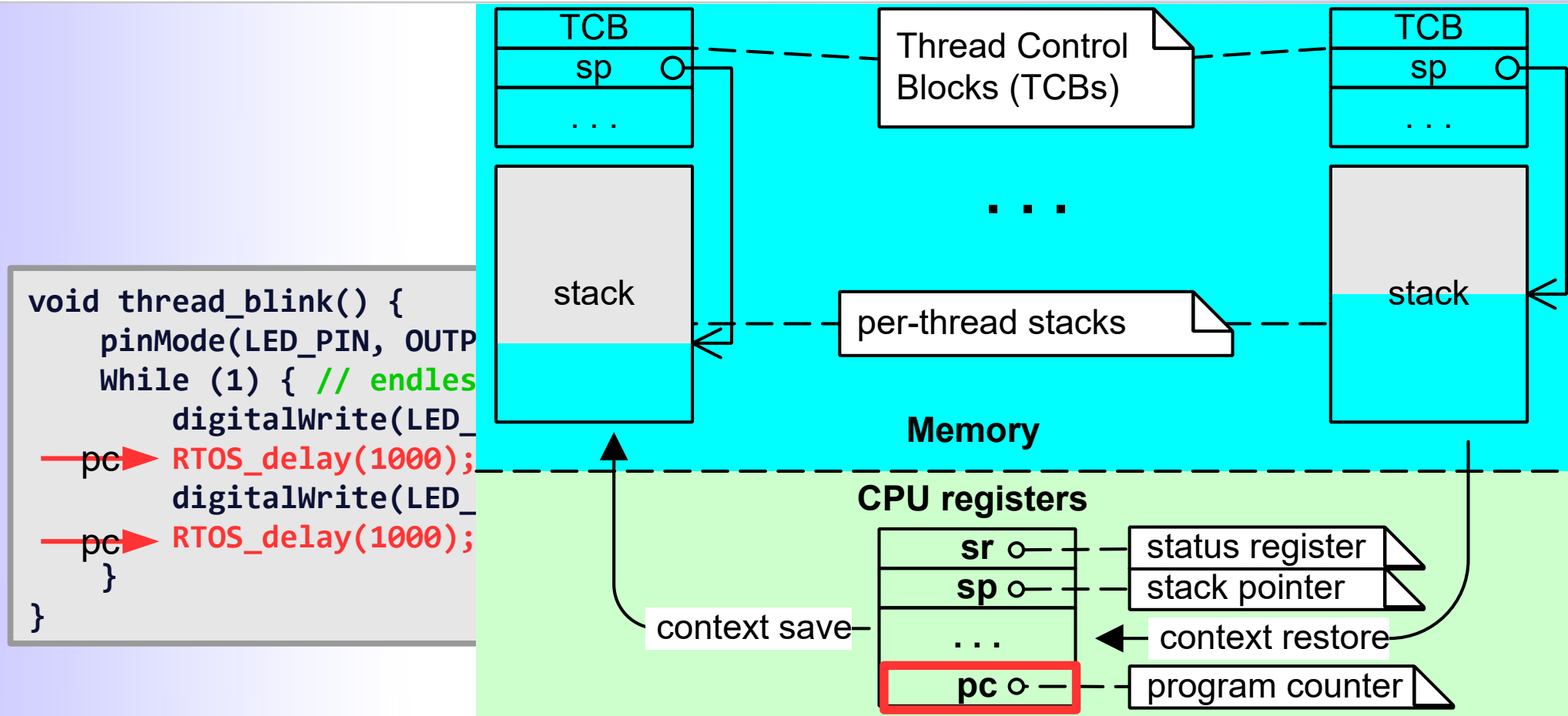
RTOS Multithreading: Multiple “Superloops”

```
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);
    }
}
```

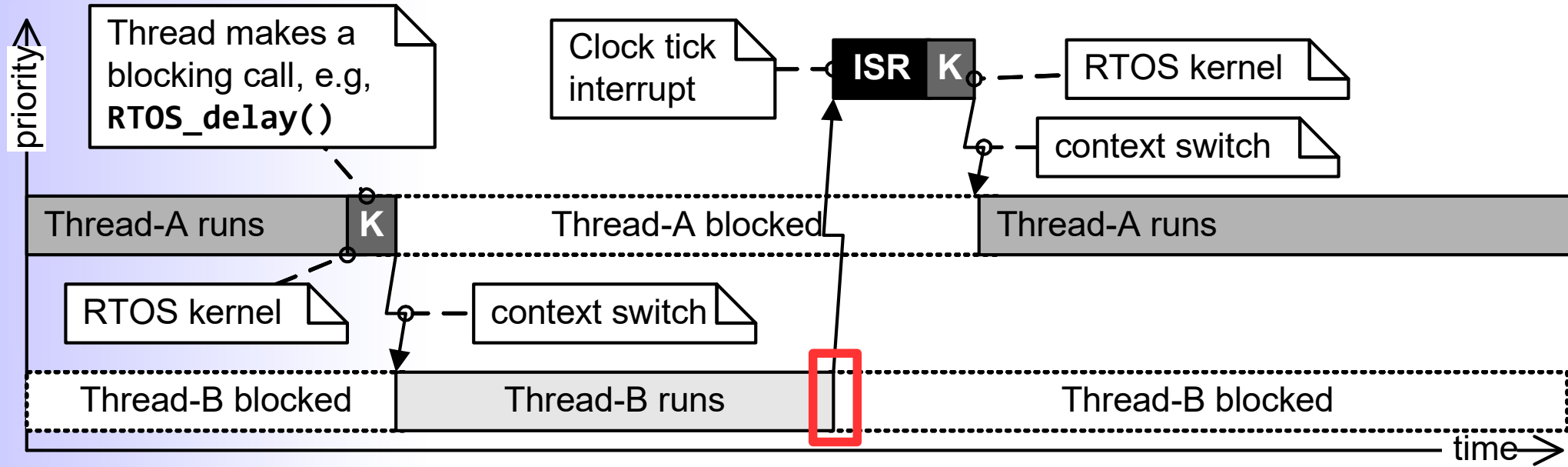
```
void thread_blink() {          // RTOS thread routine
    pinMode(LED_PIN, OUTPUT);  // setup: set pin as output
    while (1) {                // endless loop
        digitalWrite(LED_PIN, HIGH); // turn the LED on
        RTOS_delay(1000);          // wait for 1000ms
        digitalWrite(LED_PIN,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



Thread Context & Context Switch



Thread Blocking



RTOS Benefits

1) Divide and conquer strategy

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

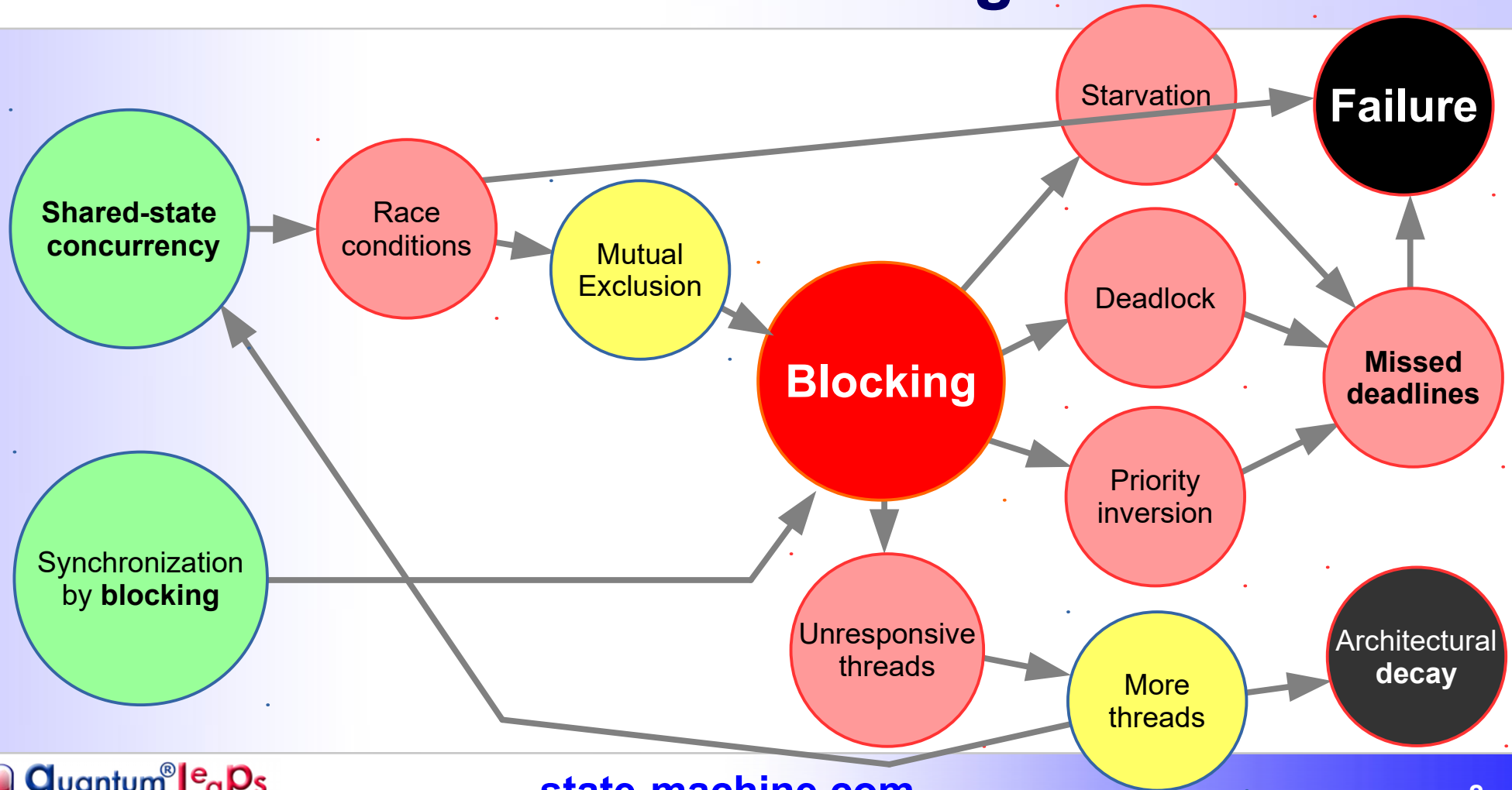
2) More efficient CPU use

→ 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))

Perils of Blocking

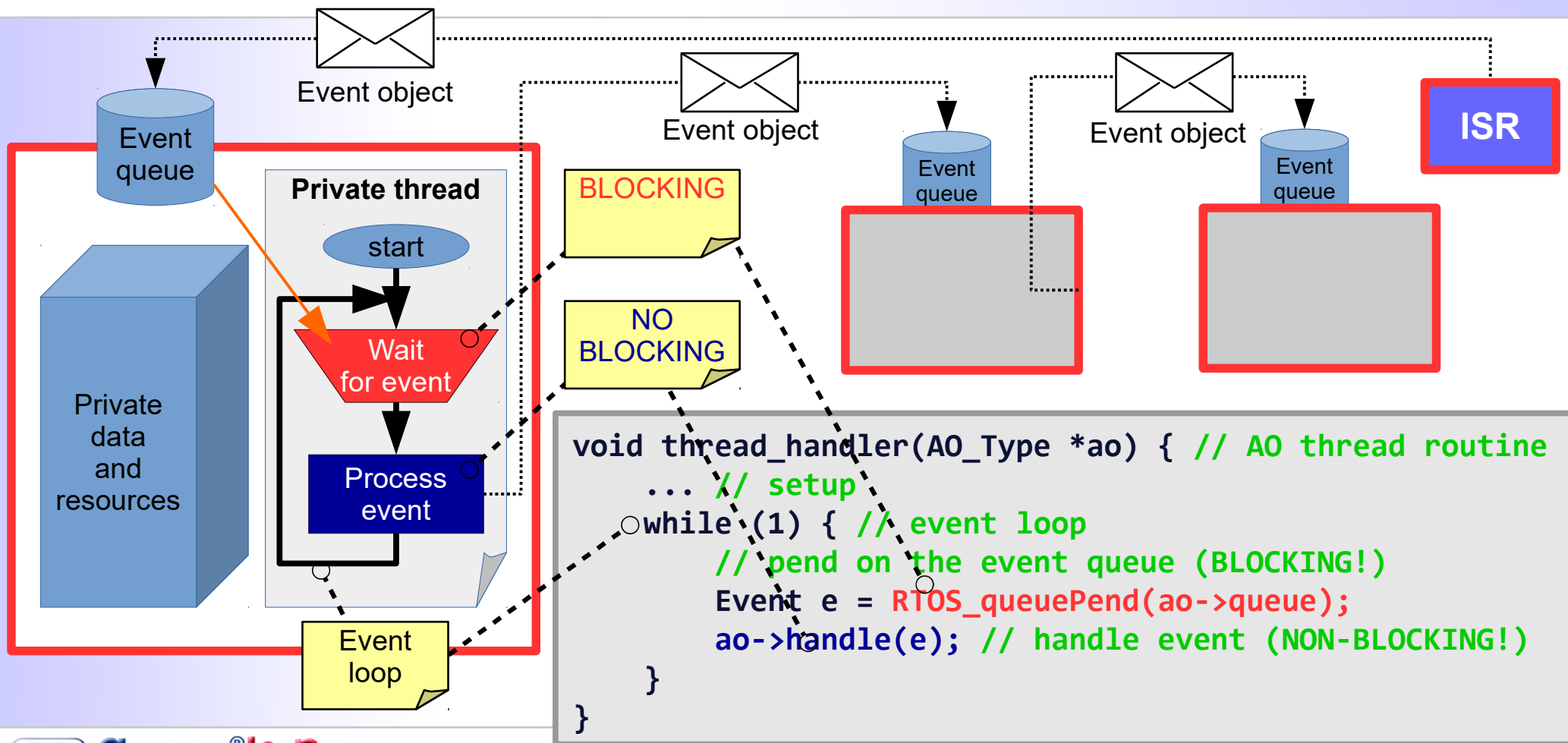


Best Practices of Concurrent Programming(*)

- **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”

Best Practices: RTOS Implementation



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.

Active Object Framework

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

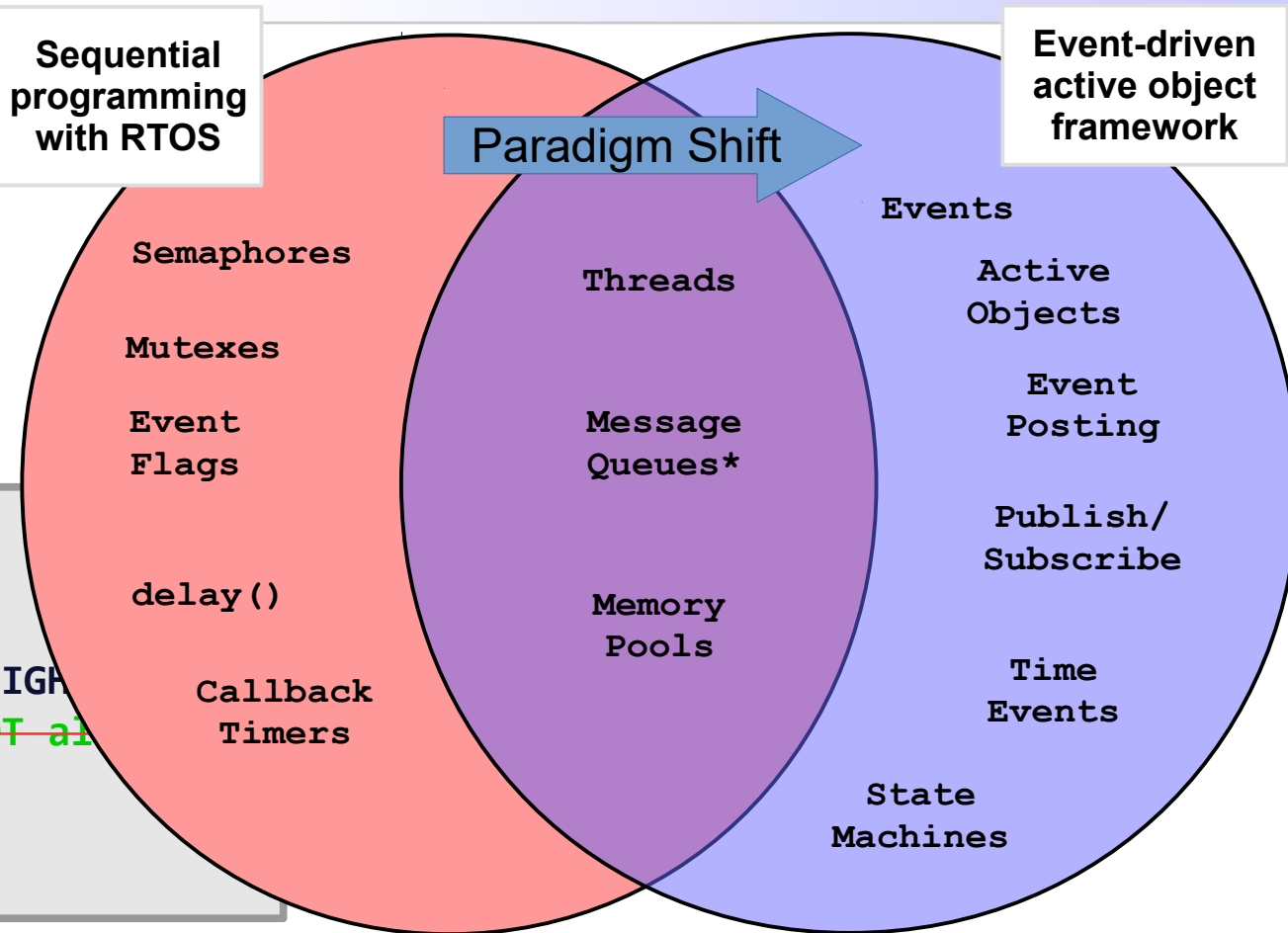
```
void thread_handler(AO_Type *ao) { // AO thread routine
    ... // setup
    while (1) { // event loop
        // pend on the event queue (BLOCKING!)
        Event e = RTOS_queuePend(ao->queue);
        ao->handle(e); // handle event (NON-BLOCKING!)
    }
}
```

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

Paradigm Shift: Sequential → Event-Driven

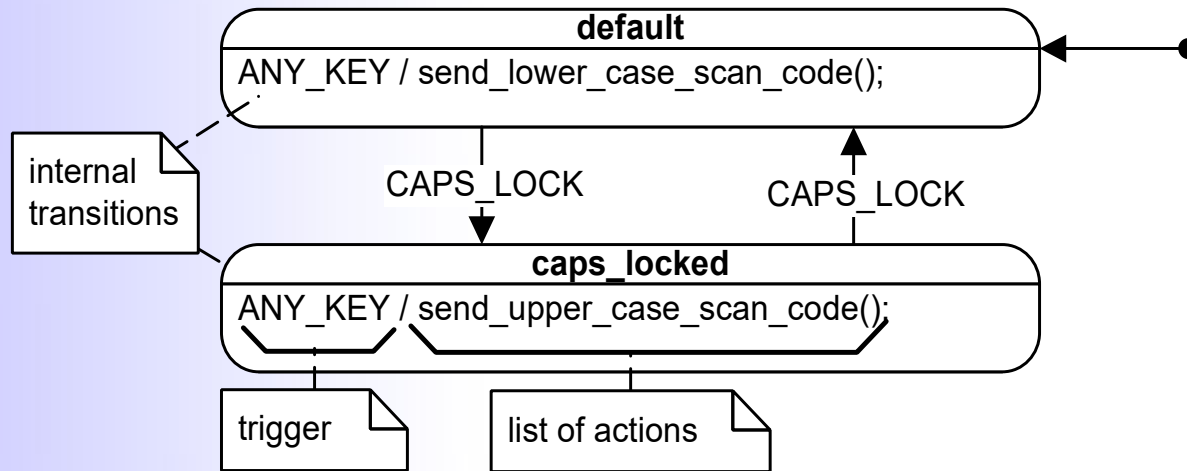
- No blocking
→ No use for most RTOS mechanisms!

```
void thread_blink() {  
    pinMode(LED_PIN, OUTPUT);  
    while (1) {  
        digitalWrite(LED_PIN, HIGH);  
        RTOS_delay(1000); // NOT allowed  
        ...  
    }  
}
```



Reduce “Spaghetti Code” with State Machines

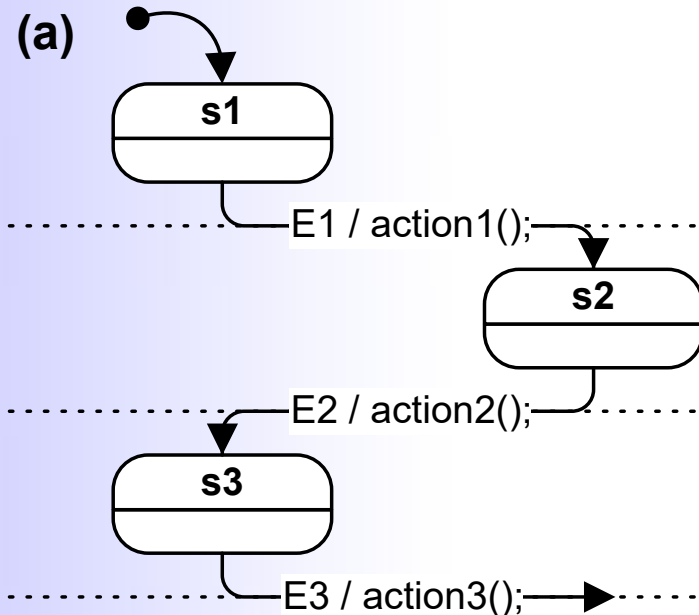
- Finite State Machines—the best known “spaghetti reducers”
 - “State” captures only the relevant aspects of the system's history
 - Natural fit for event-driven programming, where the code cannot block and must return to the event-loop after each event
 - Minimal context (a single state-variable) instead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



State Machines are not Flowcharts

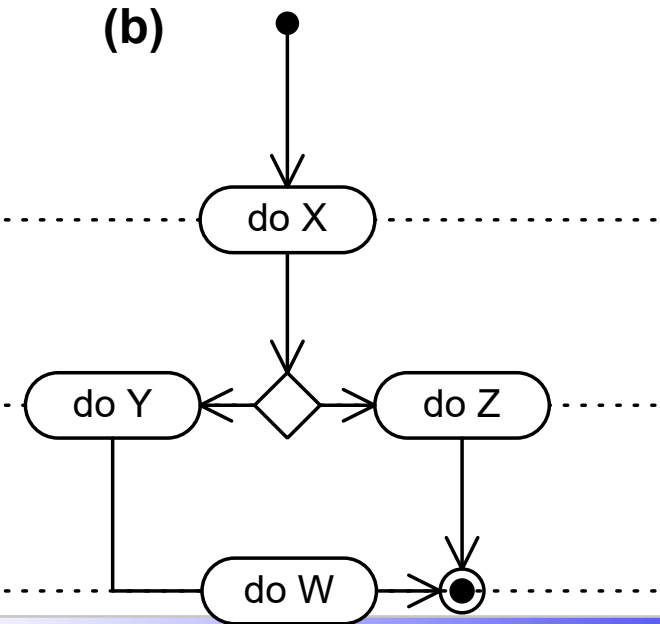
Statechart (event-driven)

- represents all states of a system
- driven by explicit **events**
- processing happens on arcs (transitions)
- no notion of “progression”



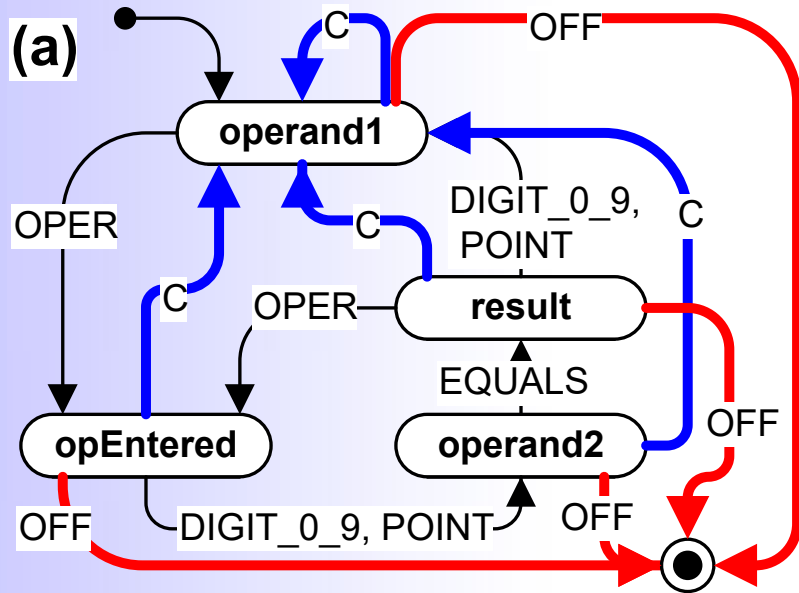
Flowchart (sequential)

- represents stages of processing in a system
- gets from node to node upon completion
- processing happens in nodes
- progresses from start to finish

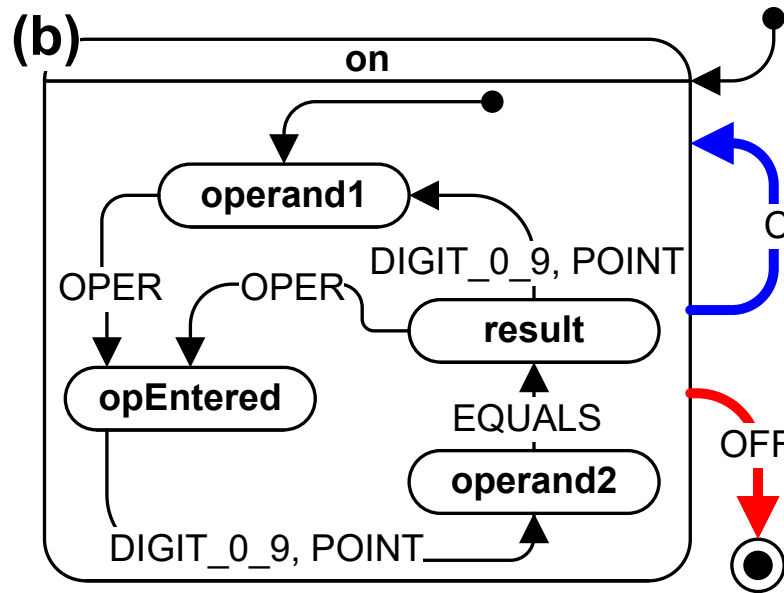


Hierarchical State Machines

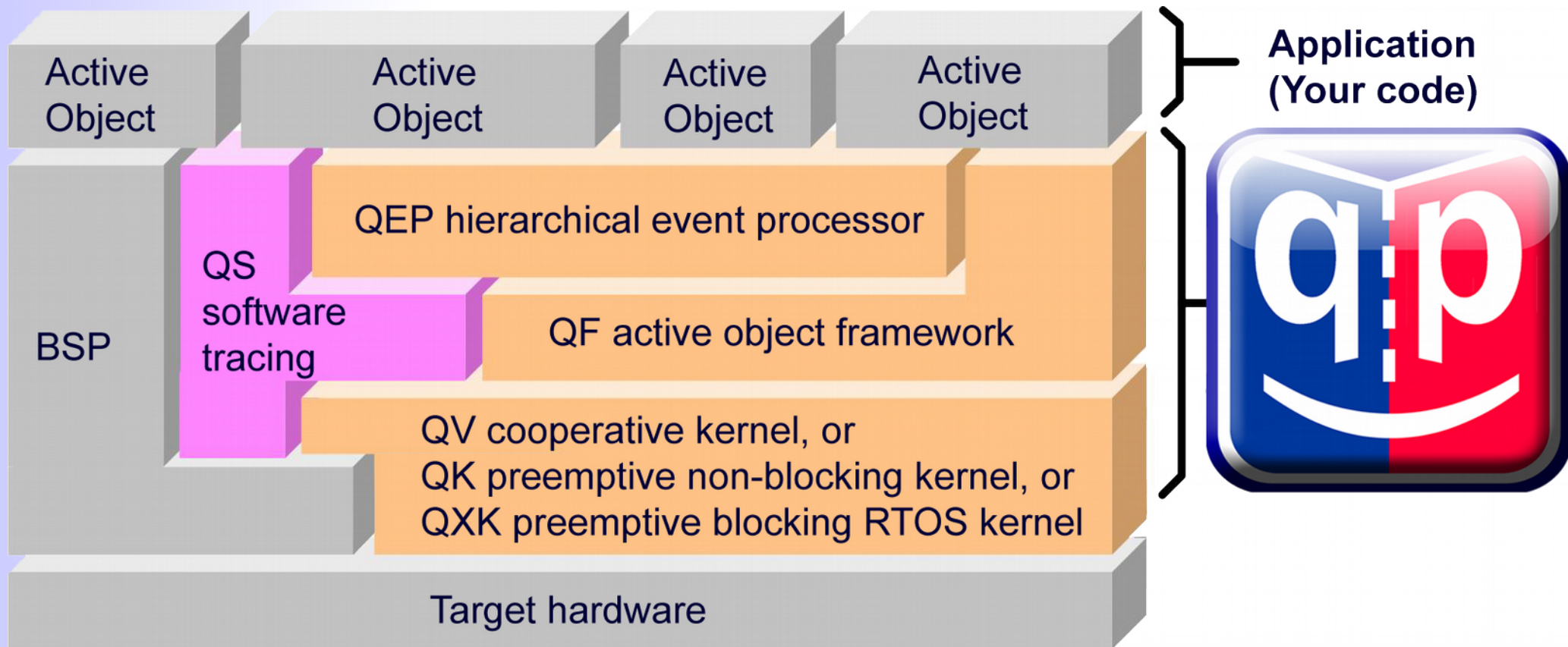
Traditional FSMs “explode”
due to **repetitions**



State hierarchy eliminates repetitions
→ programming-by-difference

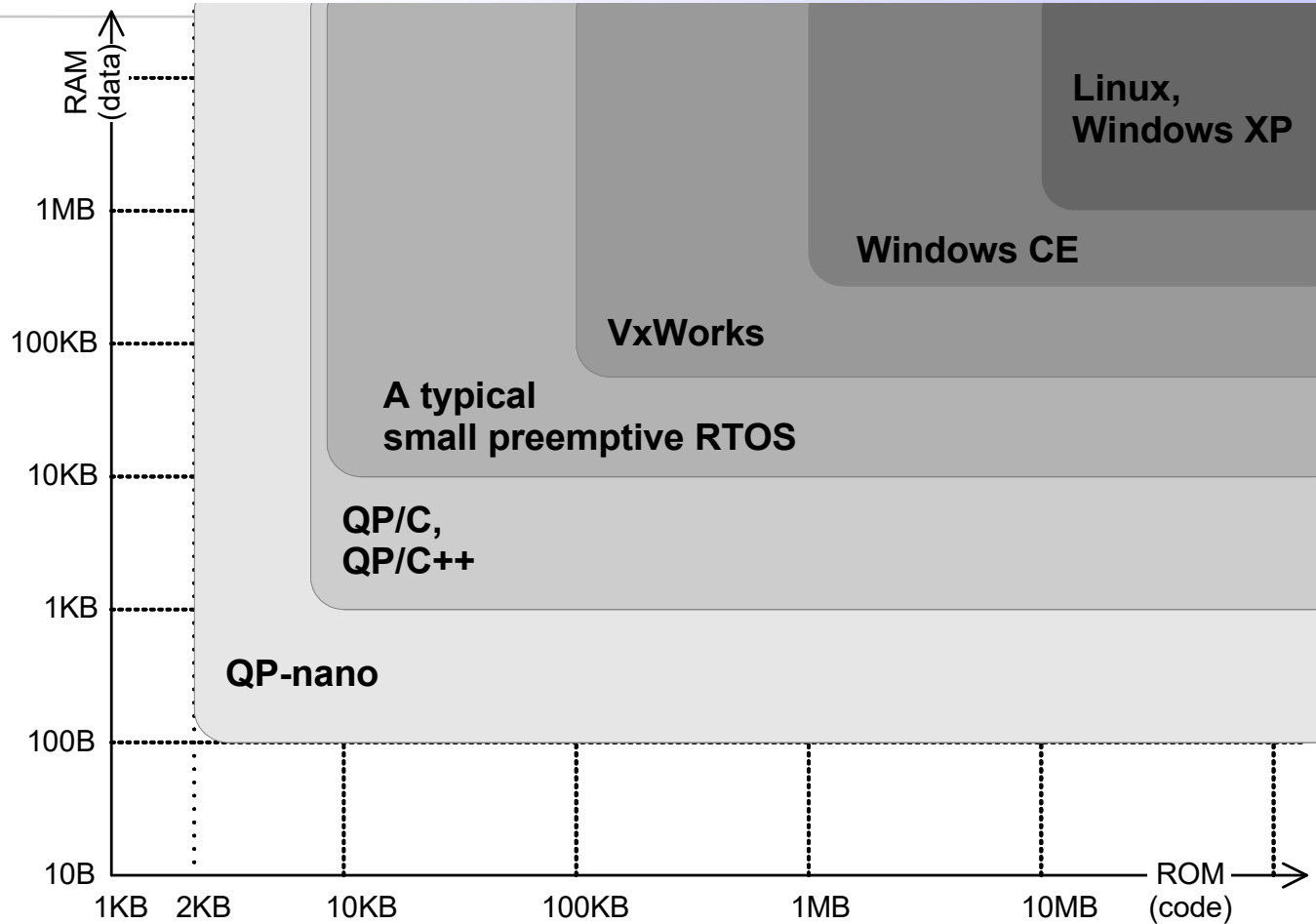


AO Frameworks for Deeply Embedded Systems

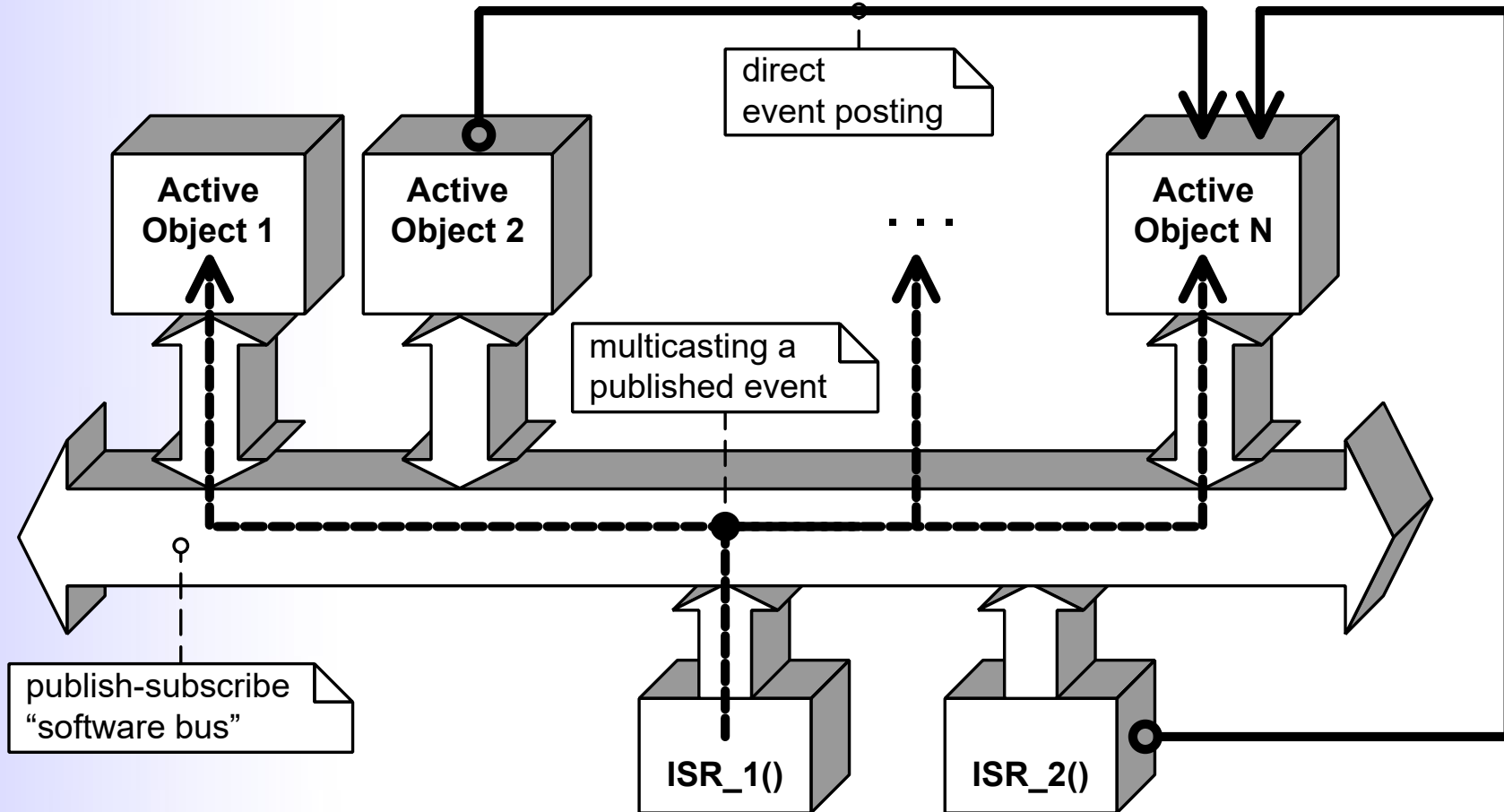


AO Frameworks vs. RTOS kernels

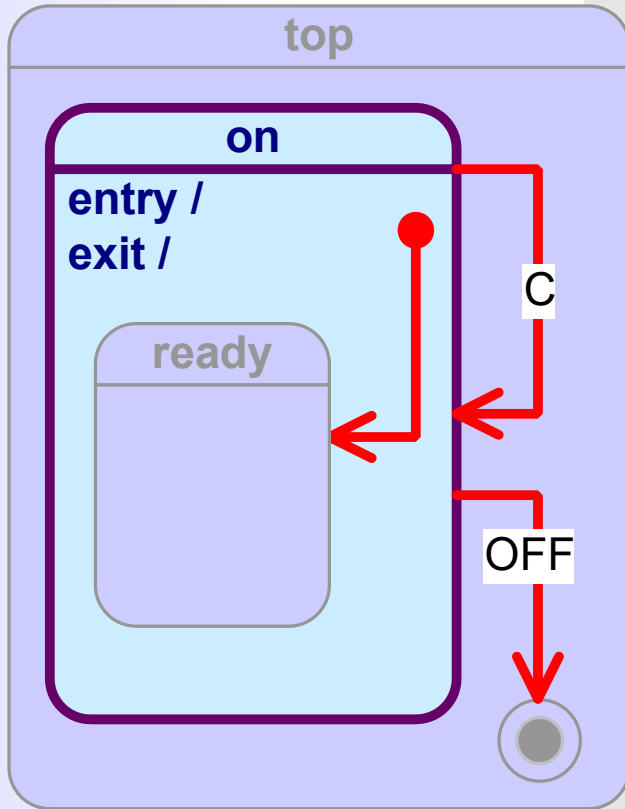
AO Frameworks can be **smaller** than RTOS kernels, because they don't need blocking



AO Framework – “Software Bus”

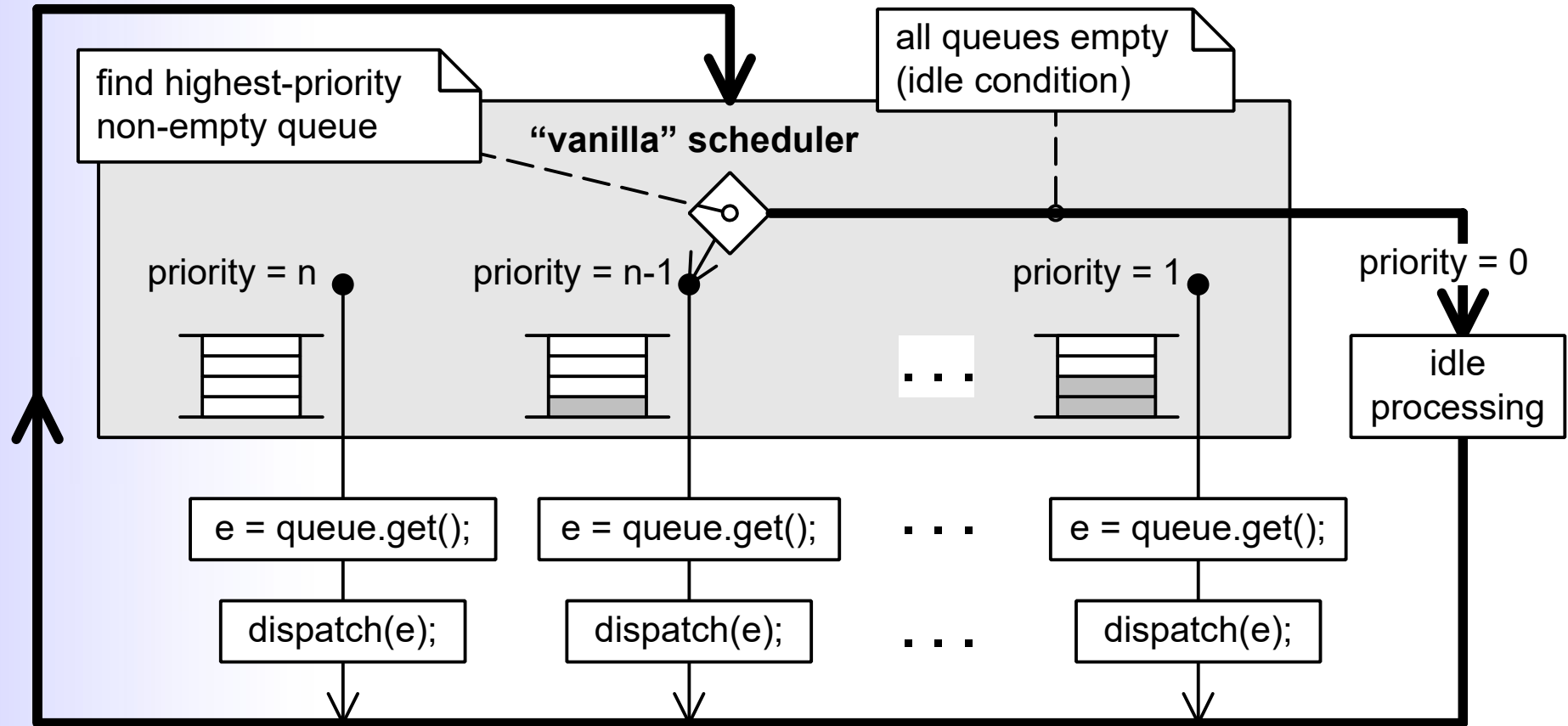


Coding Hierarchical State Machines

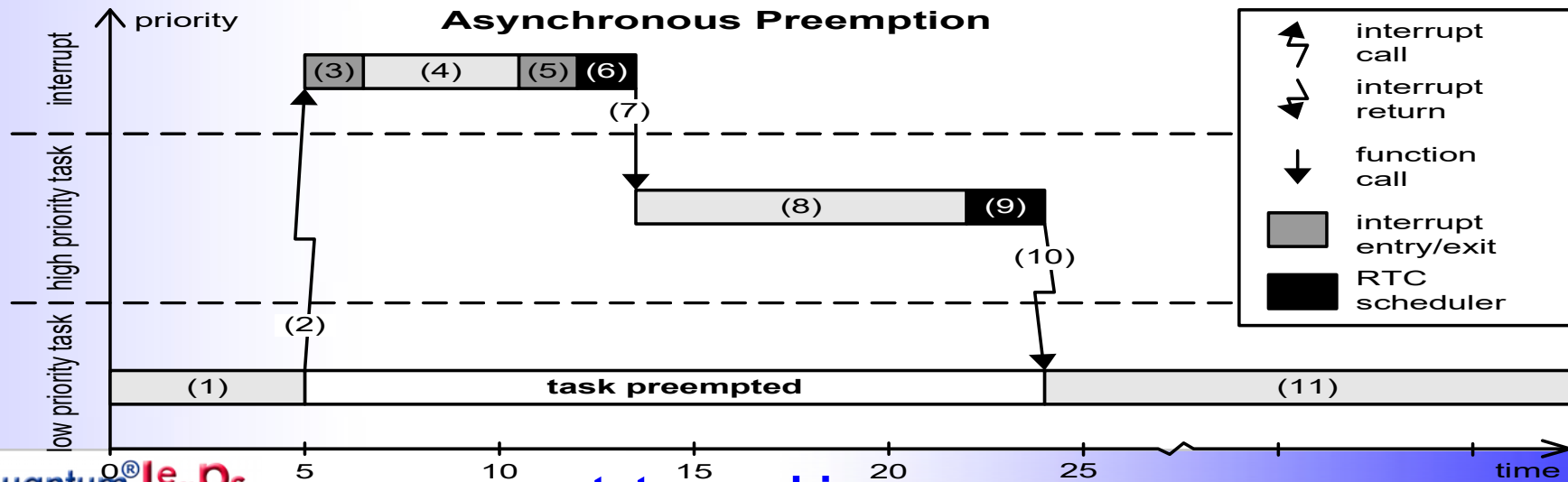
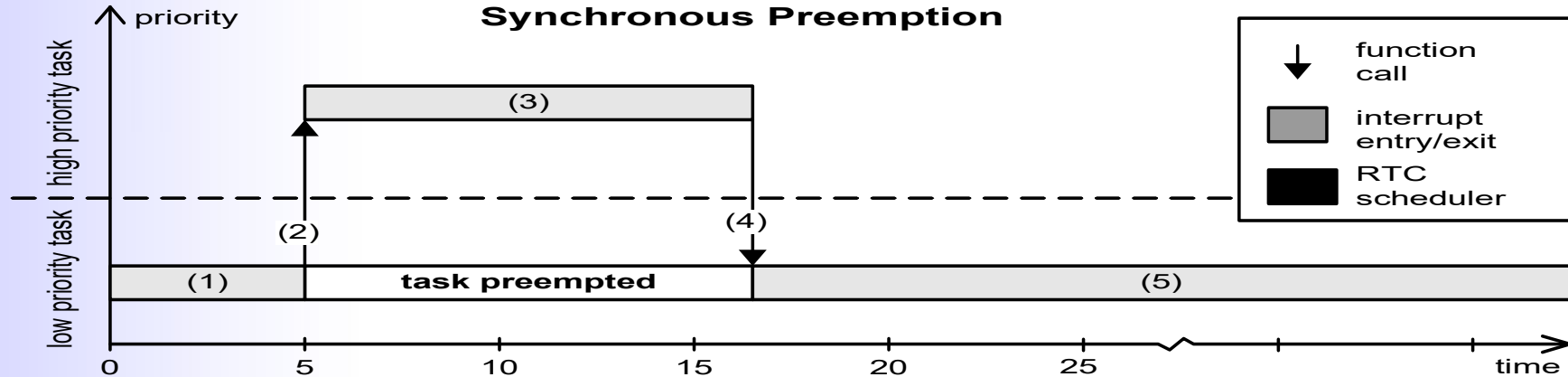


```
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG:    /* entry action */
            . . .
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG:    /* exit action */
            . . .
            status = Q_HANDLED();
            break;
        case Q_INIT_SIG:    /* initial transition */
            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;
}
```

Cooperative Kernel (QV)

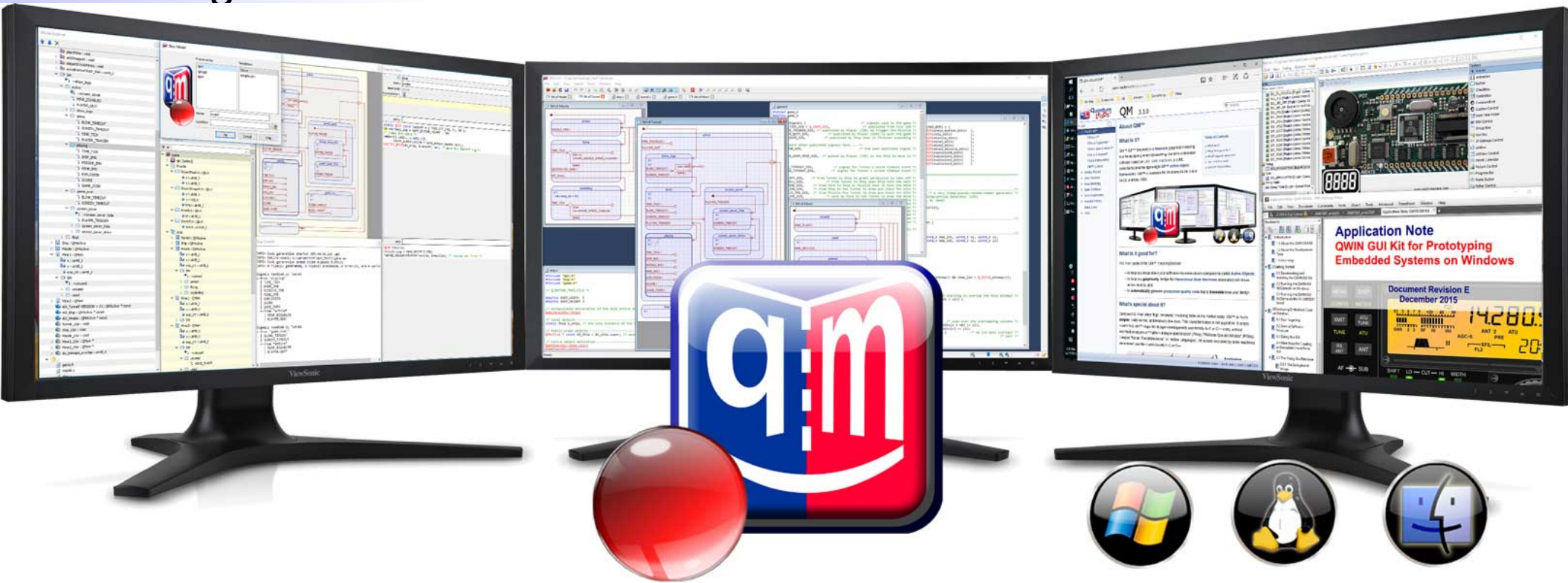


Preemptive, Non-Blocking Kernel (QK)



Graphical Modeling and Code Generation

- Active Objects enable you to effectively apply UML modeling
- A modeling tool needs an AO framework as a target for automatic code generation



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!**

Demo: Blinky on Arduino

QM 3.3.0 - C:\Presentations\Beyond_the_RTOS\blinky_arduino\blinky.qm - [SM of Blinky]

File Edit View Search Tools Window Help

Model Explorer

- blinky
 - qpn [locked]
 - AOs
 - Blinky : QActive
 - SM
 - >off
 - off
 - Q_TIMEOUT
 - on
 - Q_TIMEOUT

blinky.ino

Property Editor

Initial Transition

target: ->off

action:

```
QActive_armX((QActive *)me, 0U,  
             BSP_TICKS_PER_SEC, BSP_TICKS_PER_SEC/4)
```

Model Search

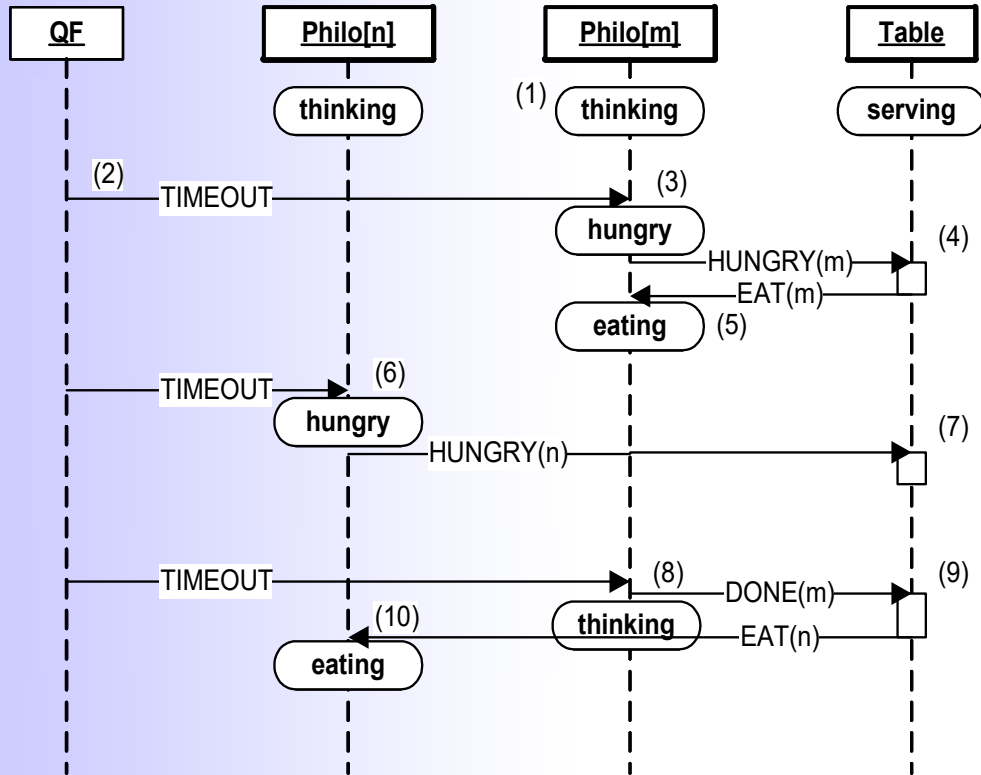
Log Console

```
INFO> Entire model: C:\Presentations\Beyond_the_RTOS\blinky_ardu:  
INFO> Code generation ended (time elapsed 0.001s)  
INFO> 0 file(s) generated, 1 file(s) processed, 0 error(s), and 0  
"  
====> OK (up to date)  
}}}  
External tool finished normally with status 0
```

Demo: PELICAN on Arduino



Demo: Dining Philosophers with Q-SPY



The screenshot shows the Q-SPY tool interface. The main window displays a log of system events, including messages like **ENTRY**, **Disp**, **Inter**, **Ignor**, **UnHd**, and **EXIT** for various objects and states. A **Canvas** window is overlaid on the log, displaying a cartoon illustration of Homer Simpson and his family, with Homer Simpson holding a fork and a knife, and the family members eating.

Demo: “Fly 'n' Shoot” game on Windows

