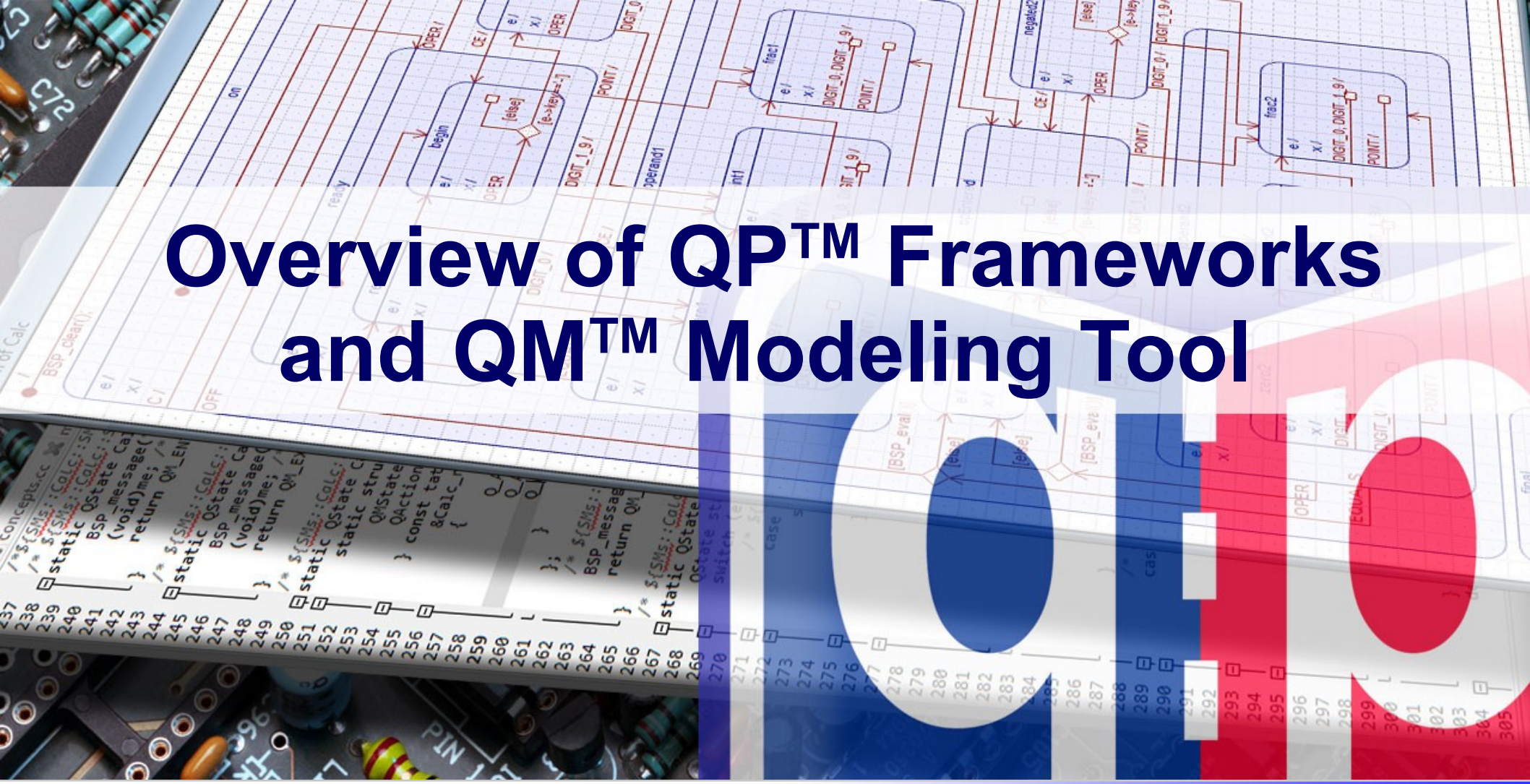
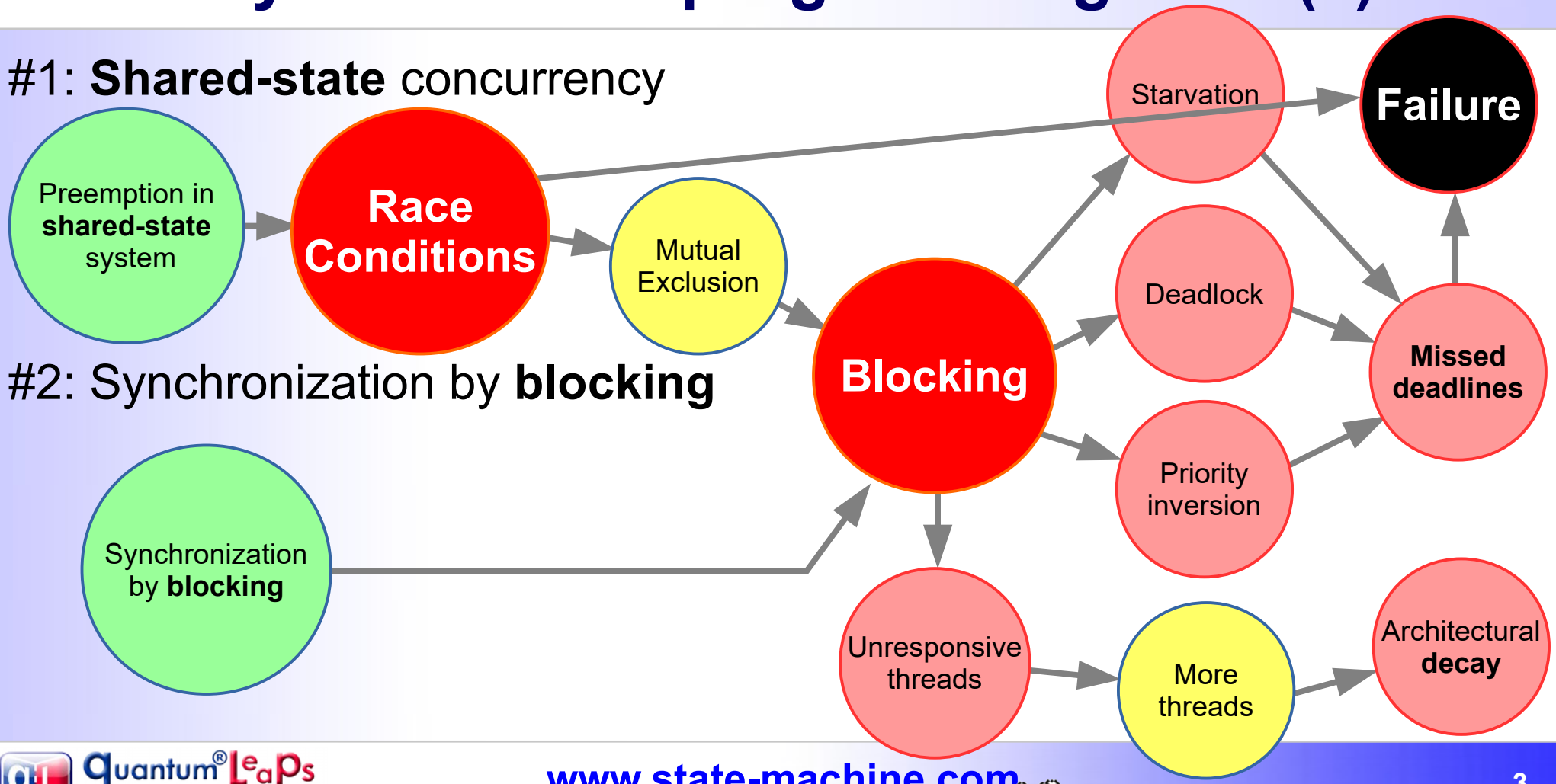


Overview of QP™ Frameworks and QM™ Modeling Tool

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

Why is real-time programming hard (1)?



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)**

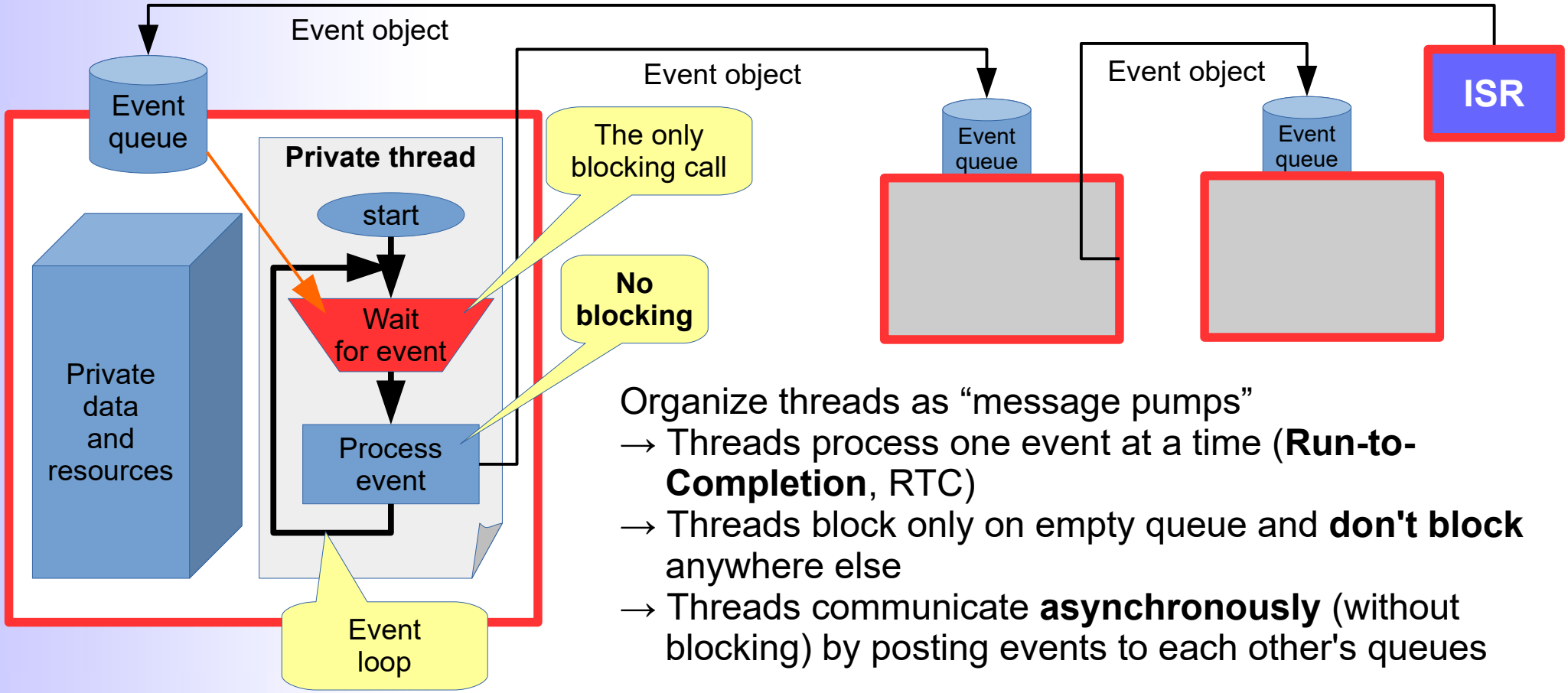
(*) Herb Sutter "Prefer Using Active Objects Instead of Naked Threads"

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

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

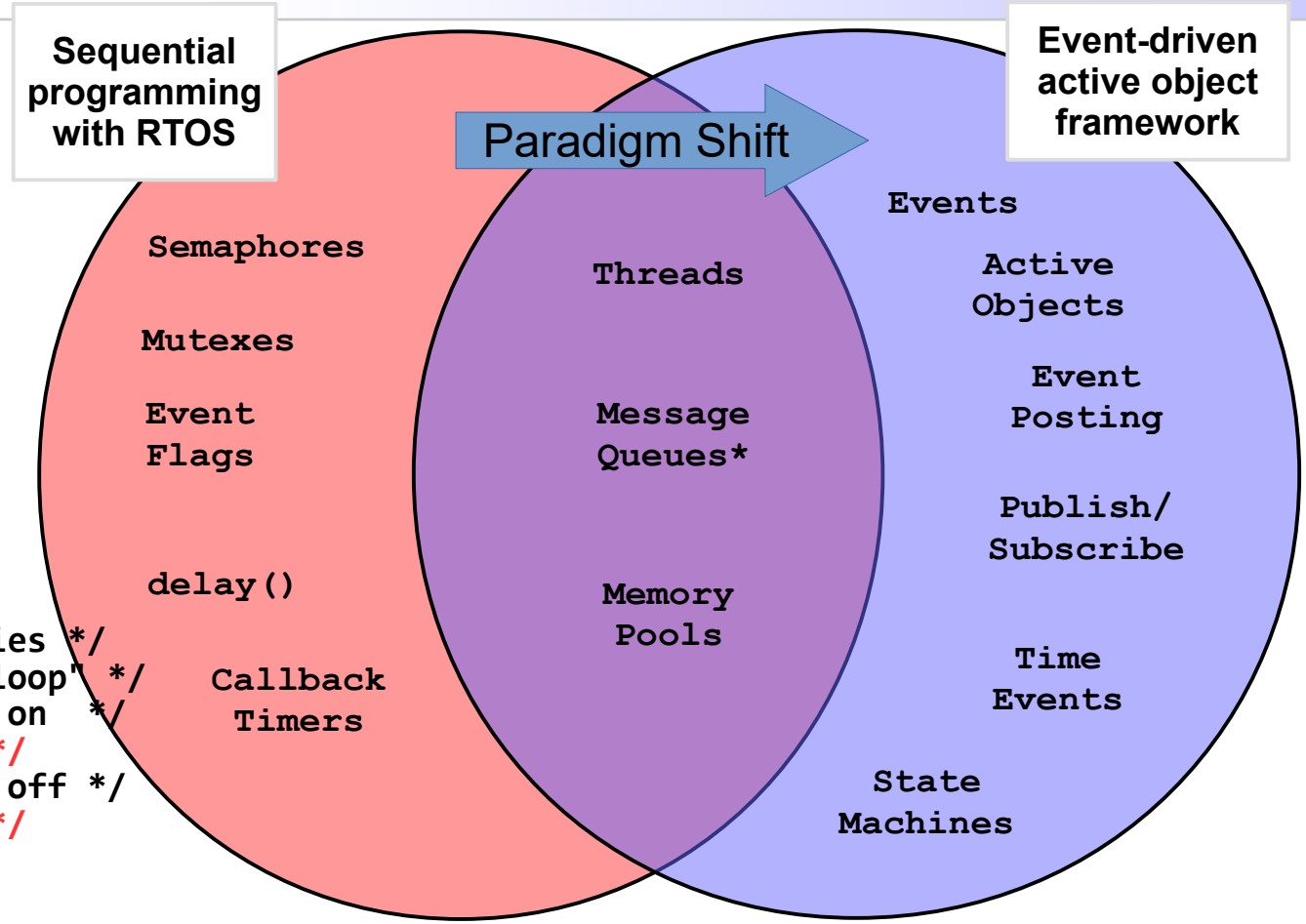
A Better Way: Active Object Framework

- 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)

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!!! */
}
```



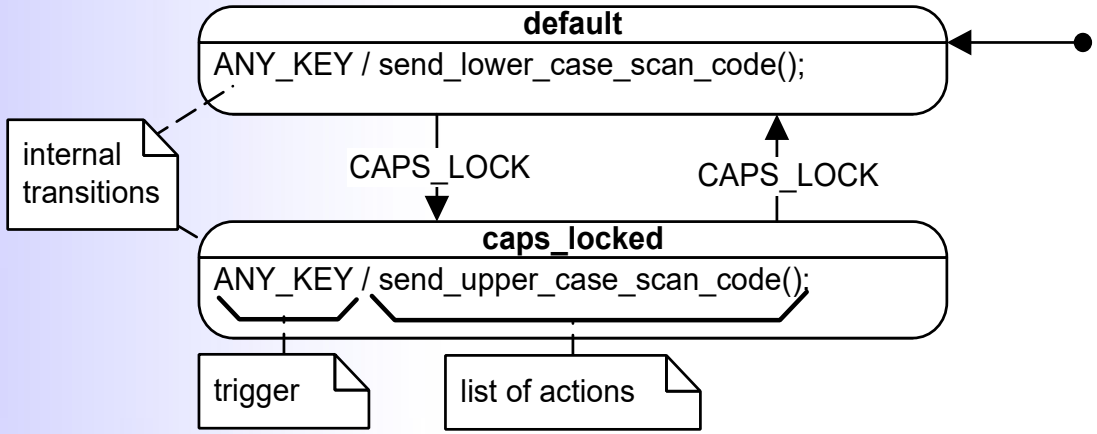
Why is real-time programming hard (2)?

- 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**

```
1  void operator_click(int Index) {
2      char TempReadout[READOUT_LEN];
3      strcpy(TempReadout, Readout);
4      if (strcmp(LastInput, "NUMS") == 0) {
5          NumOps++;
6      }
7      switch (NumOps) {
8          case 0:
9              if ((Operator[Index].Caption[0] == '-')
10                 && (strcmp(LastInput, "NEG") == 0))
11             {
12                 strcpy(&Readout[1], &Readout[0]);
13                 Readout[0] = '-';
14                 LastInput = "NEG";
15             }
16             break;
17         case 1:
18             Opt1 = Readout;
19             if ((Operator[Index].Caption[0] == '-')
20                 && (strcmp(LastInput, "NUMS") != 0)
21                 && (OpFlag != '='))
22             {
23                 Readout[0] = '-';
24                 LastInput = "NEG";
25             }
26             break;
27         case 2:
28             Op2 = TempReadout;
29             switch (OpFlag) {
30                 case '+':
31                     Op1 = CDbl(Op1) + CDbl(Op2);
32                     break;
33                 case '-':
34                     Op1 = CDbl(Op1) - CDbl(Op2);
35                     break;
36                 case '*':
37                     Op1 = CDbl(Op1) * CDbl(Op2);
38                     break;
39                 case '/':
40                     if (Op2 == 0) {
41                         Error("Division by zero");
42                     }
43                     Op1 = CDbl(Op1) / CDbl(Op2);
44                     break;
45             }
46             Readout = Op1;
47             NumOps = 1;
48             break;
49     }
50     if (strcmp(LastInput, "NEG") == 0) {
51         strcpy(LastInput, "OPS");
52         OpFlag = Operator[Index].Caption;
53     }
54 }
```

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

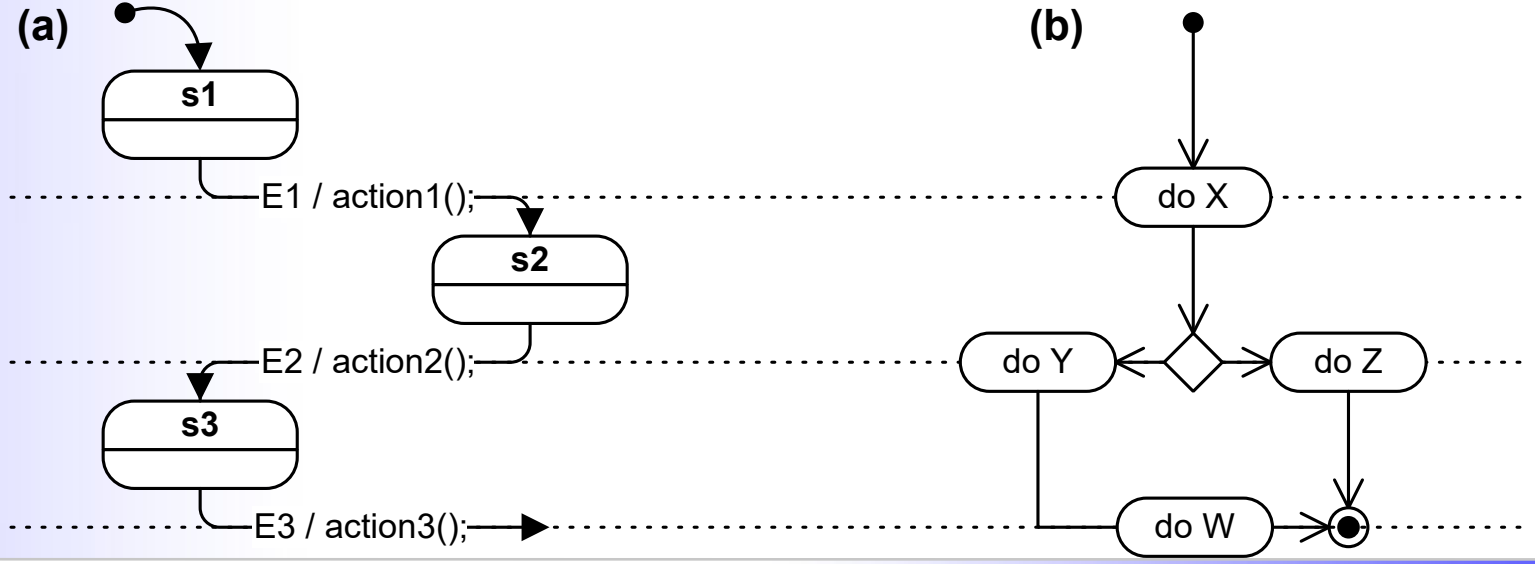


Paradigm Shift: Sequential → Event-Driven (2)

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"

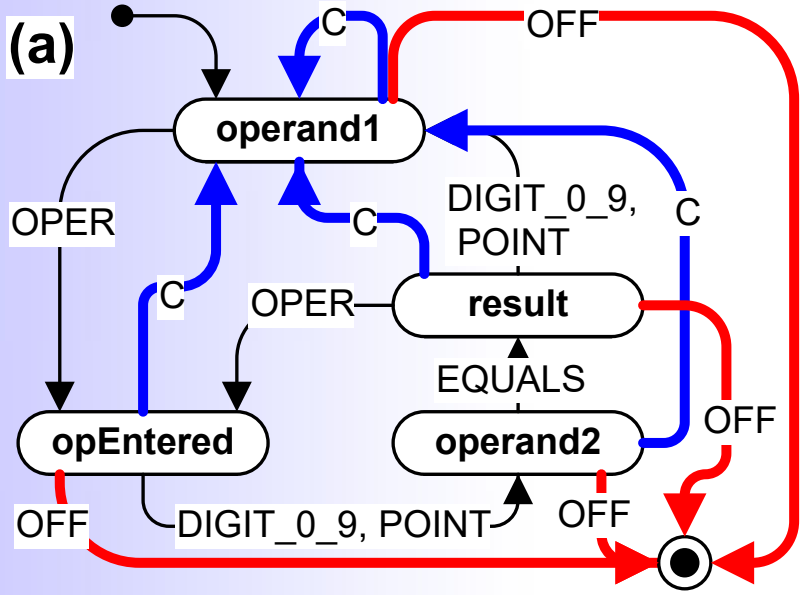
Flowchart (sequential)
→ represents stages of processing in a system
→ gets from node to node upon completion
→ processing happens in nodes
→ progresses from start to finish

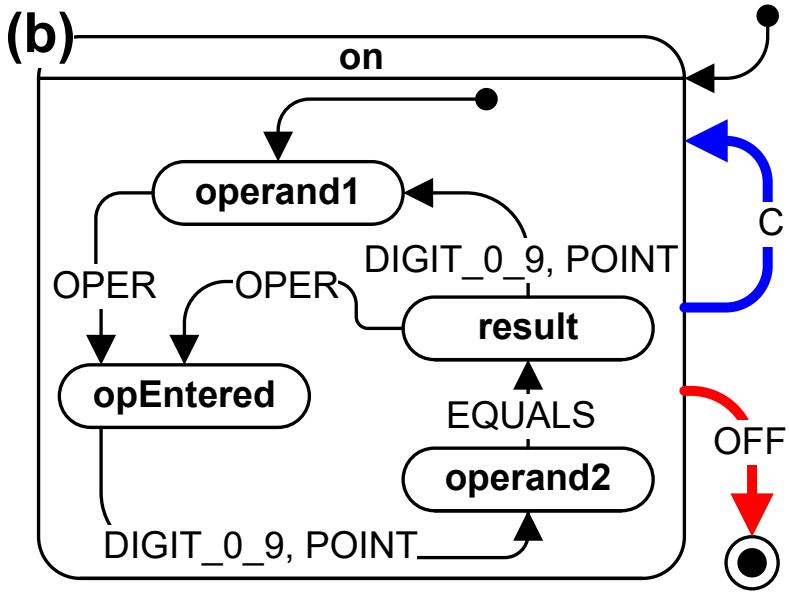


Hierarchical State Machines

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

State hierarchy eliminates repetitions → 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

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++

Who is using QP™?

professional

open source

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

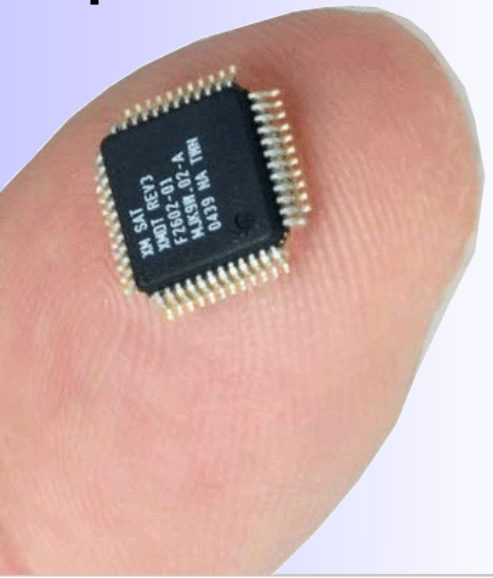
→ Consumer electronics
→ Medical devices
→ Defense
→ Industrial controls
→ Communication & IoT
→ Robotics
→ Semiconductor IP
→ … (see online)

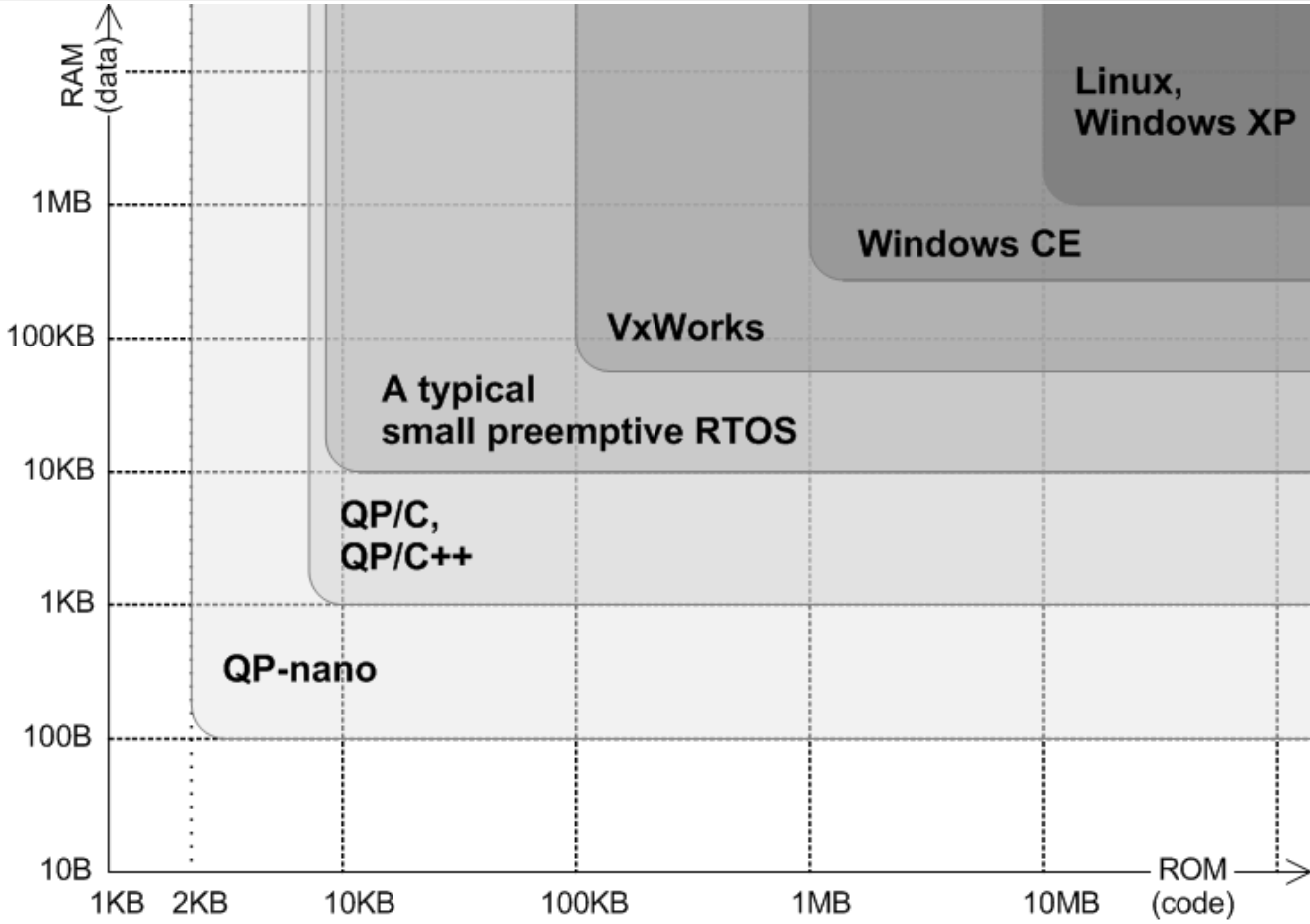
QP™ Framework Family Features

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	✓	✓	×

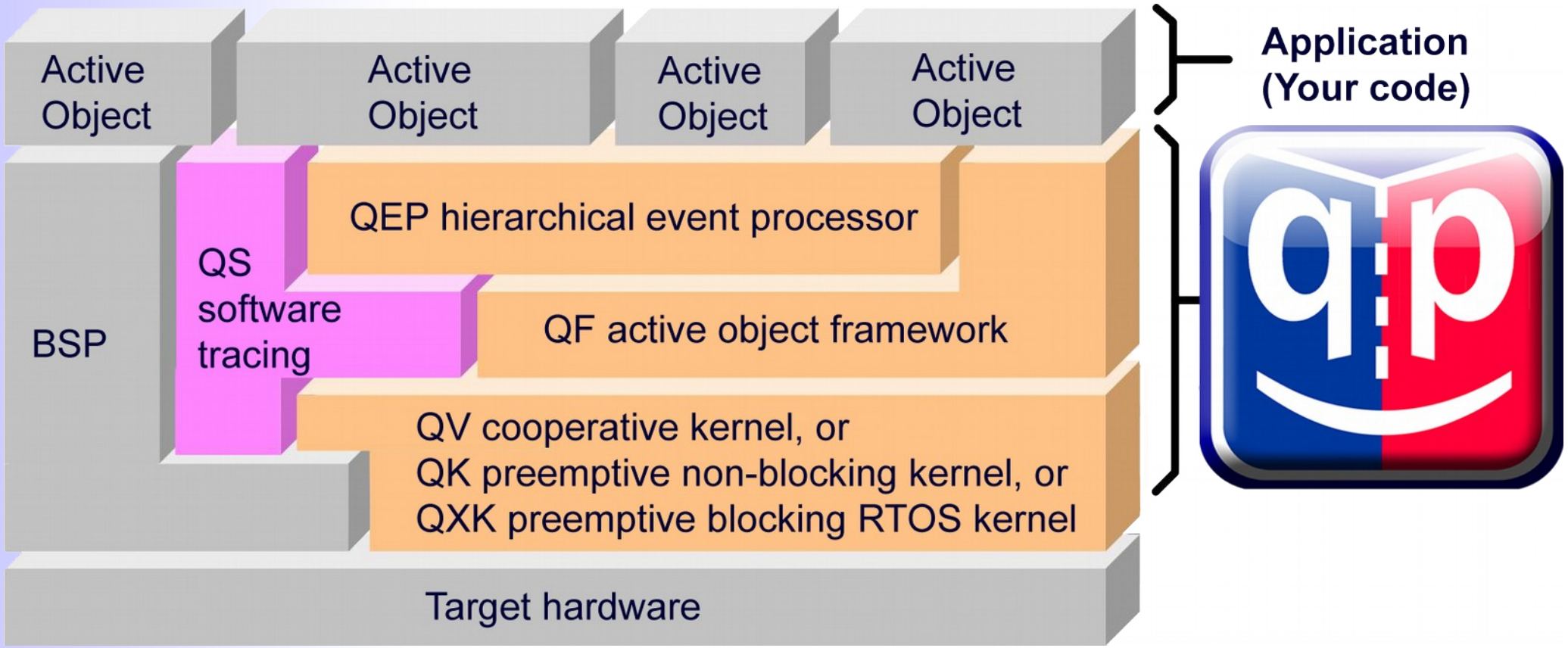
QP™ vs. RTOS Memory Footprint

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

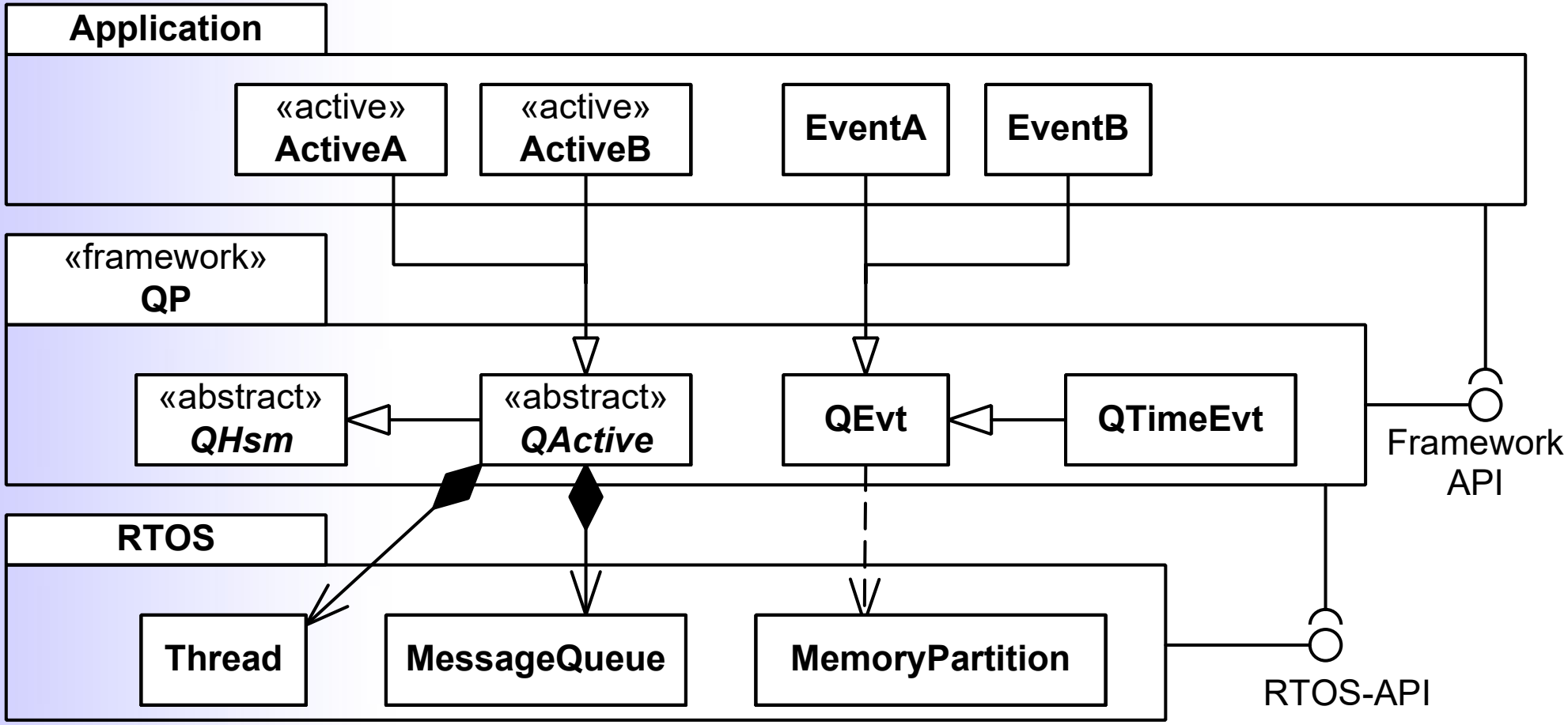




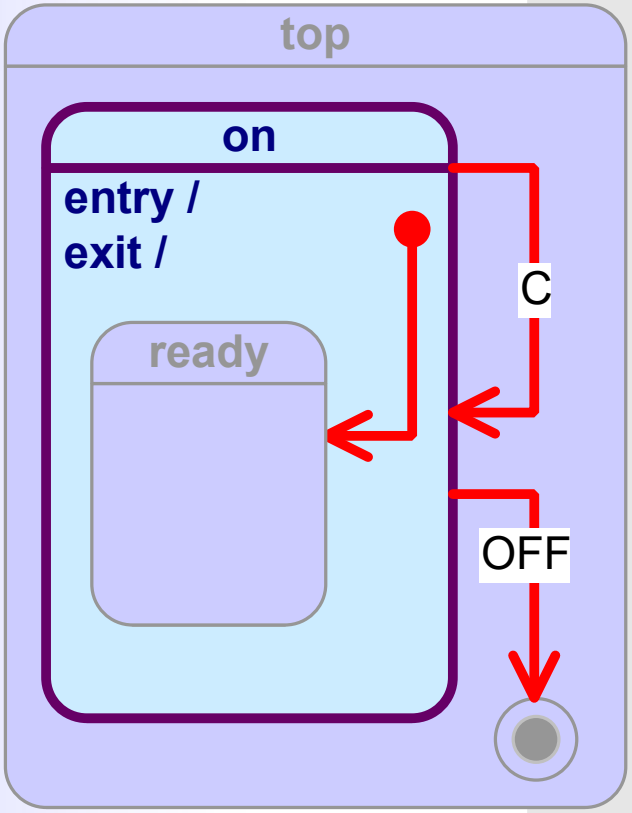
QP™ Components and Layers



QP™ Package and Class View

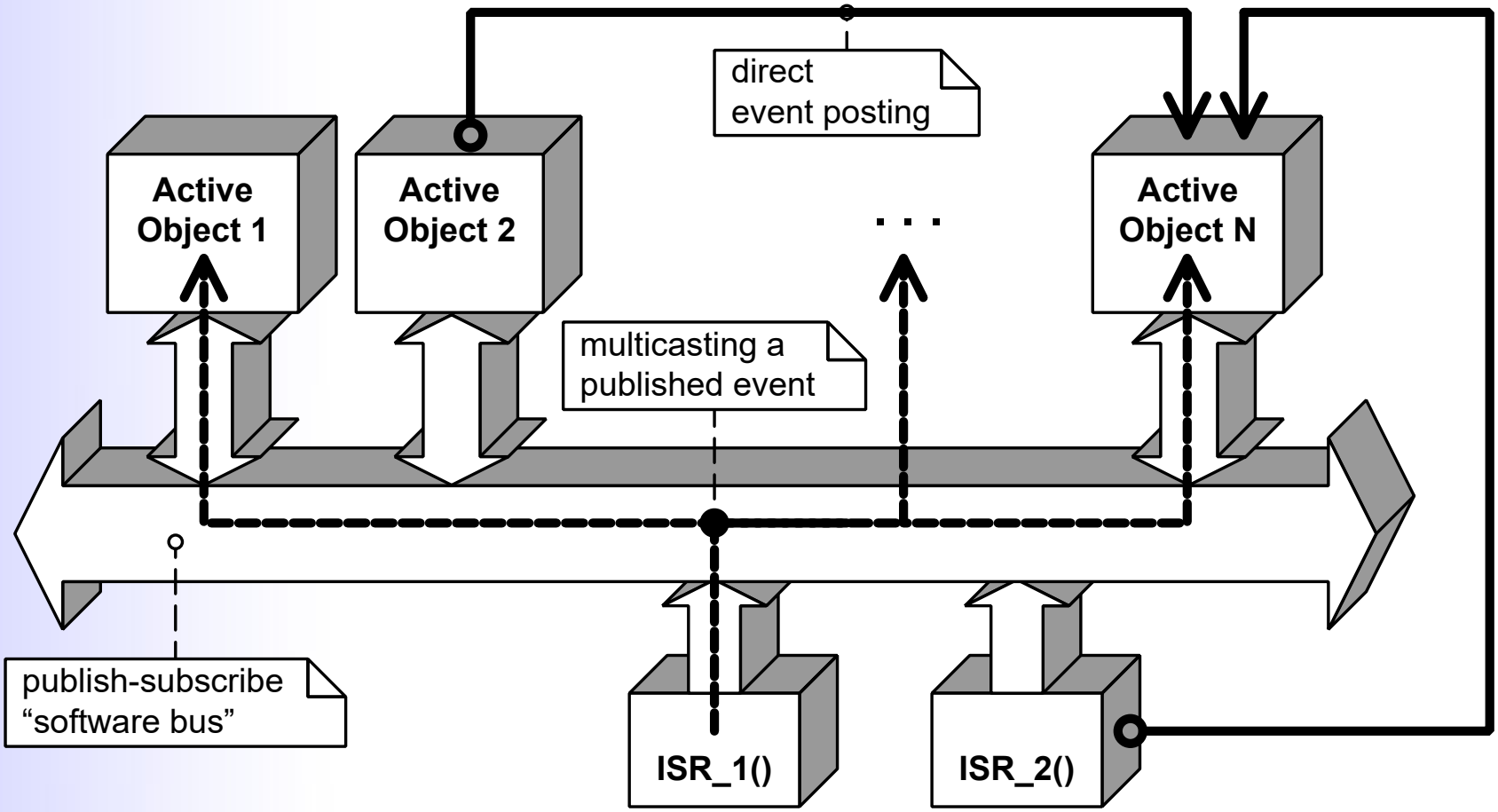


QEP Hierarchical Event Processor

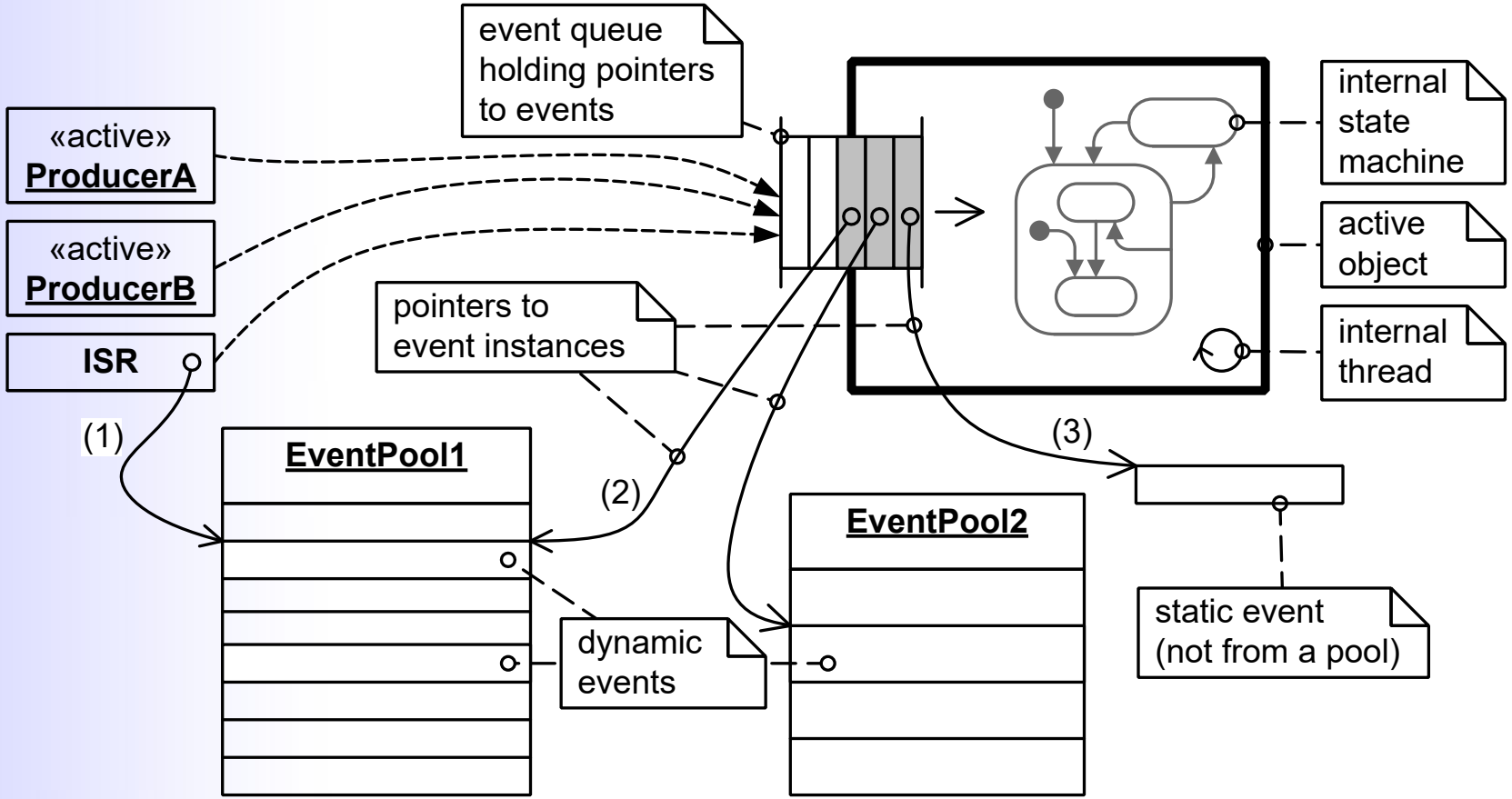



```
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG:   /* entry action */
            BSP_message("on-ENTRY");
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG:    /* exit action */
            BSP_message("on-EXIT");
            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;
}
```

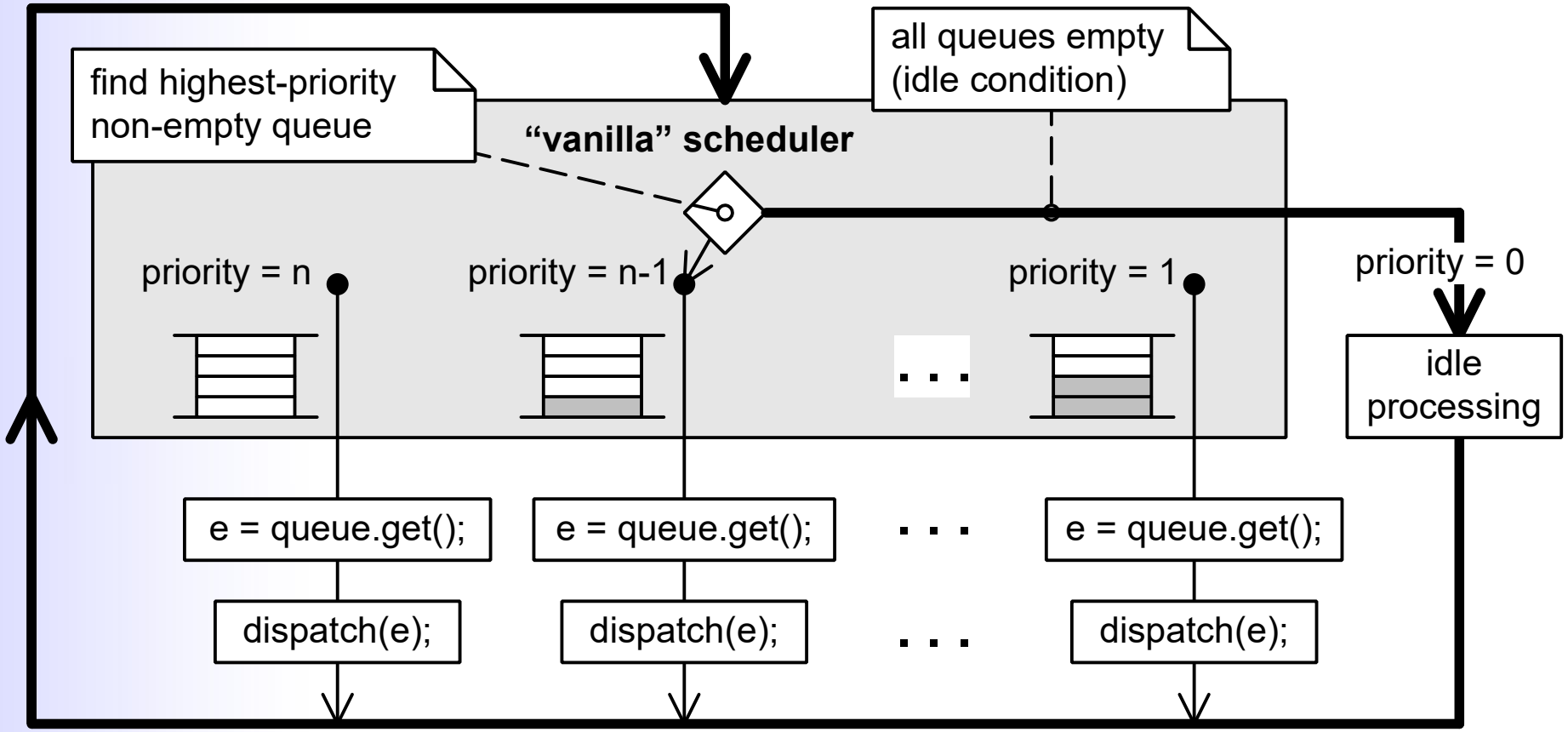
QF AO Framework – “Software Bus”



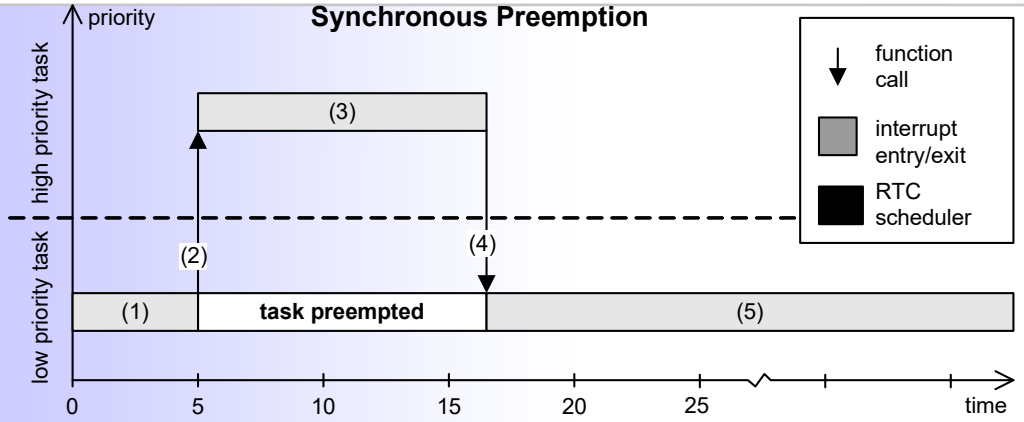
QF AO Framework – "Zero Copy" Event Delivery

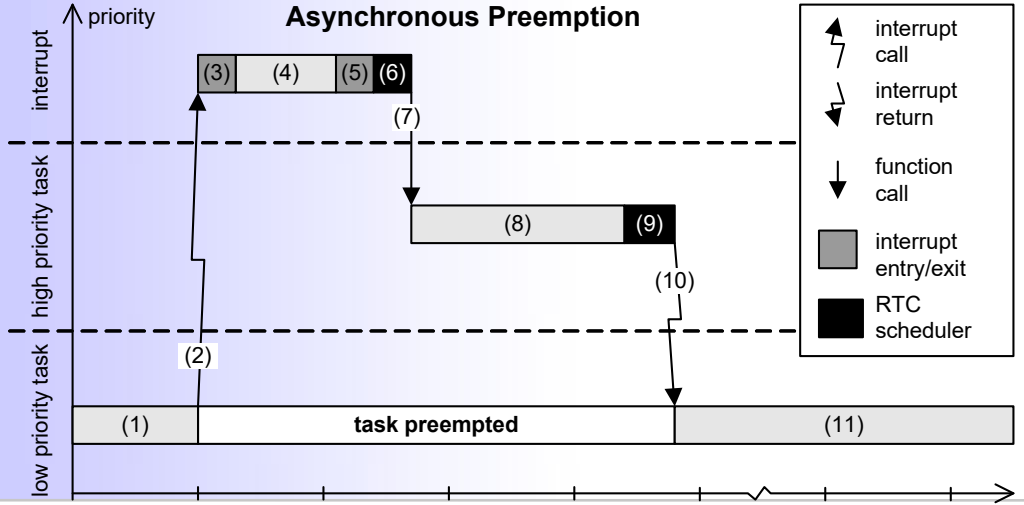


QV Cooperative Kernel



QK Preemptive, Non-Blocking Kernel

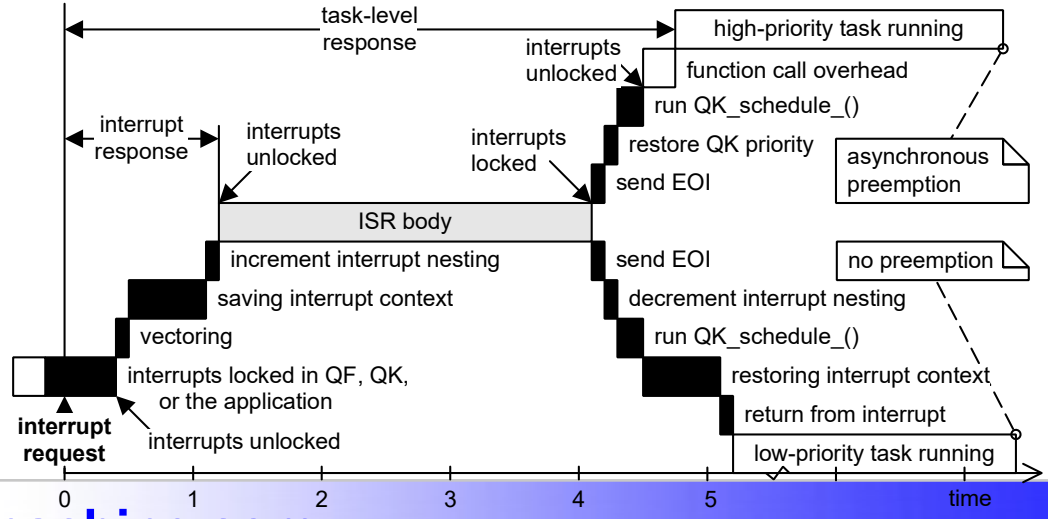




- Preemptive priority-based kernel
- Meets all requirements of Rate Monotonic Analysis (RMA)
- Run-to-Completion Kernel

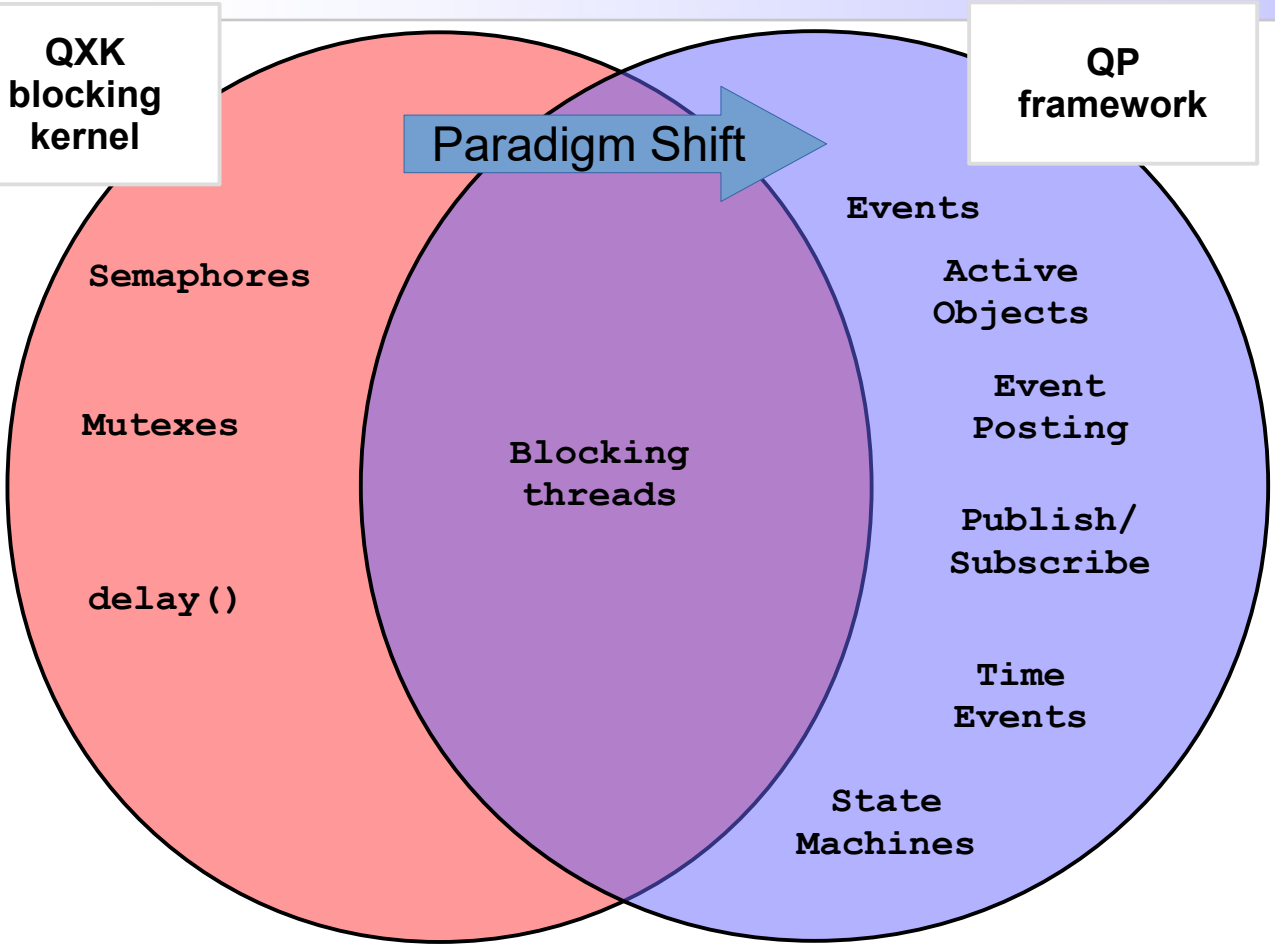
→ **Cannot block** in-line

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



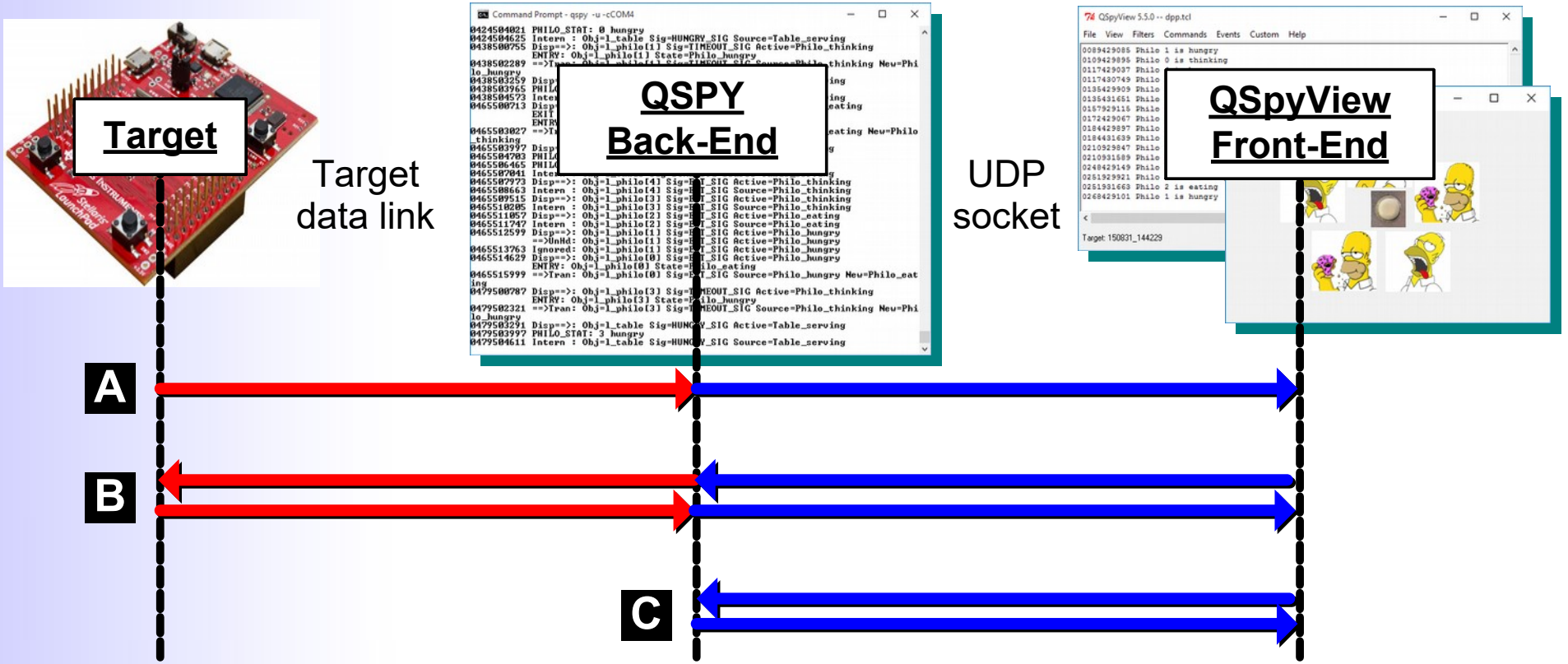
QXK Preemptive, Blocking Kernel

- 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.



QS/QSPY Software Tracing System

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



QUTest Unit Testing Harness

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

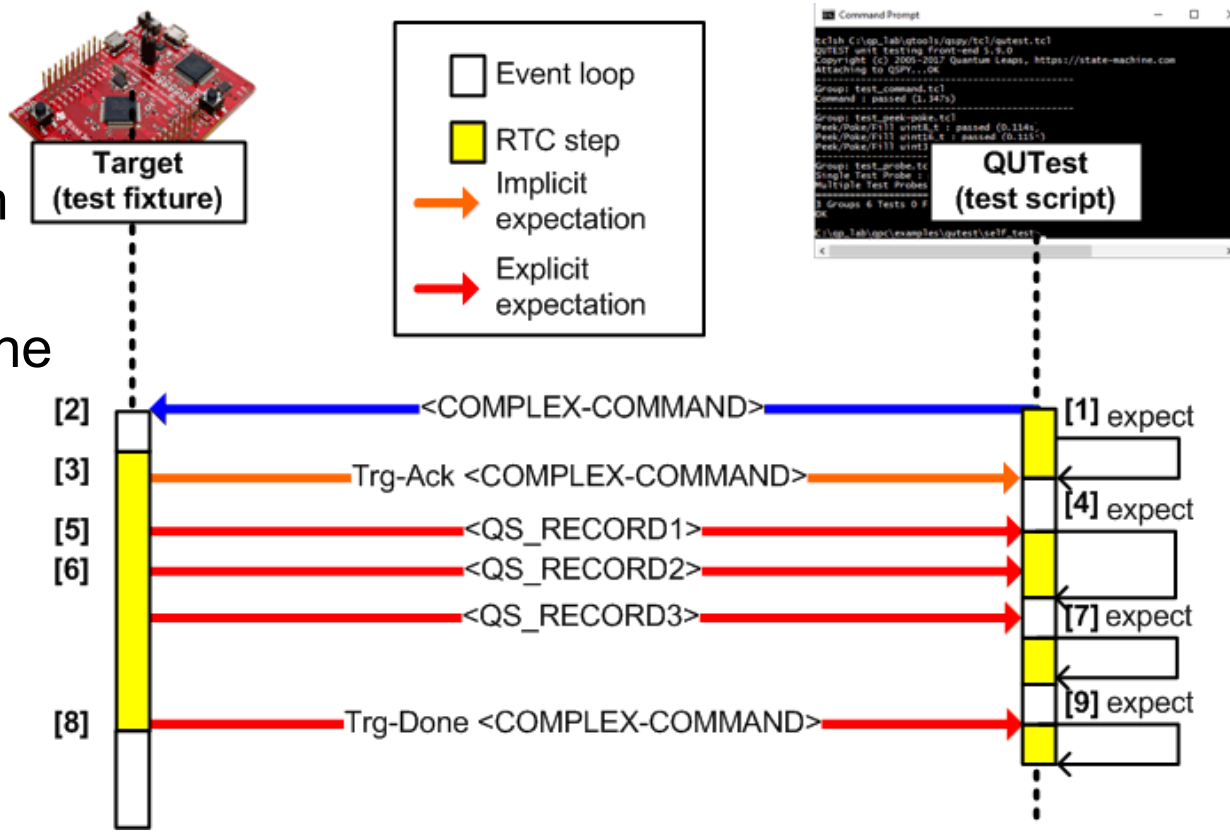
→ Separates CUT execution from checking the test assertions

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

→ Driving the tests and checking correctness on the Host

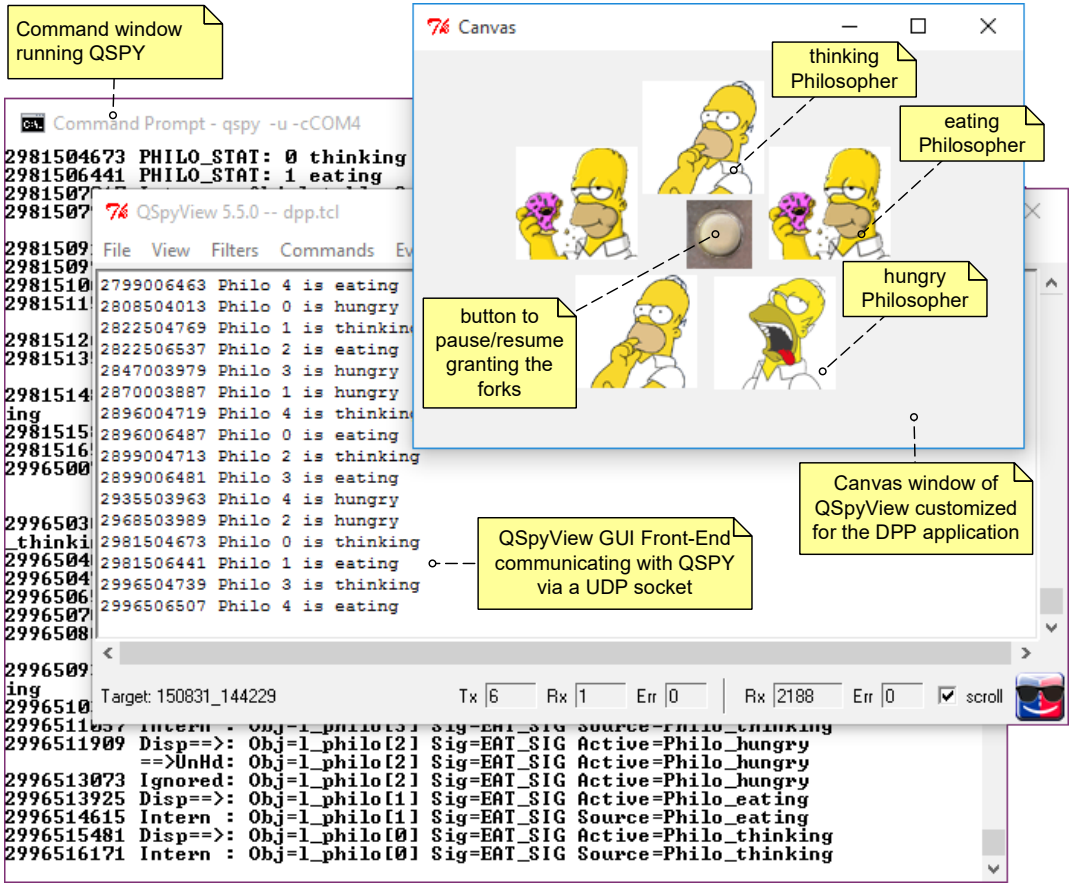
→ **Python** and Tcl test scripting

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



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



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

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

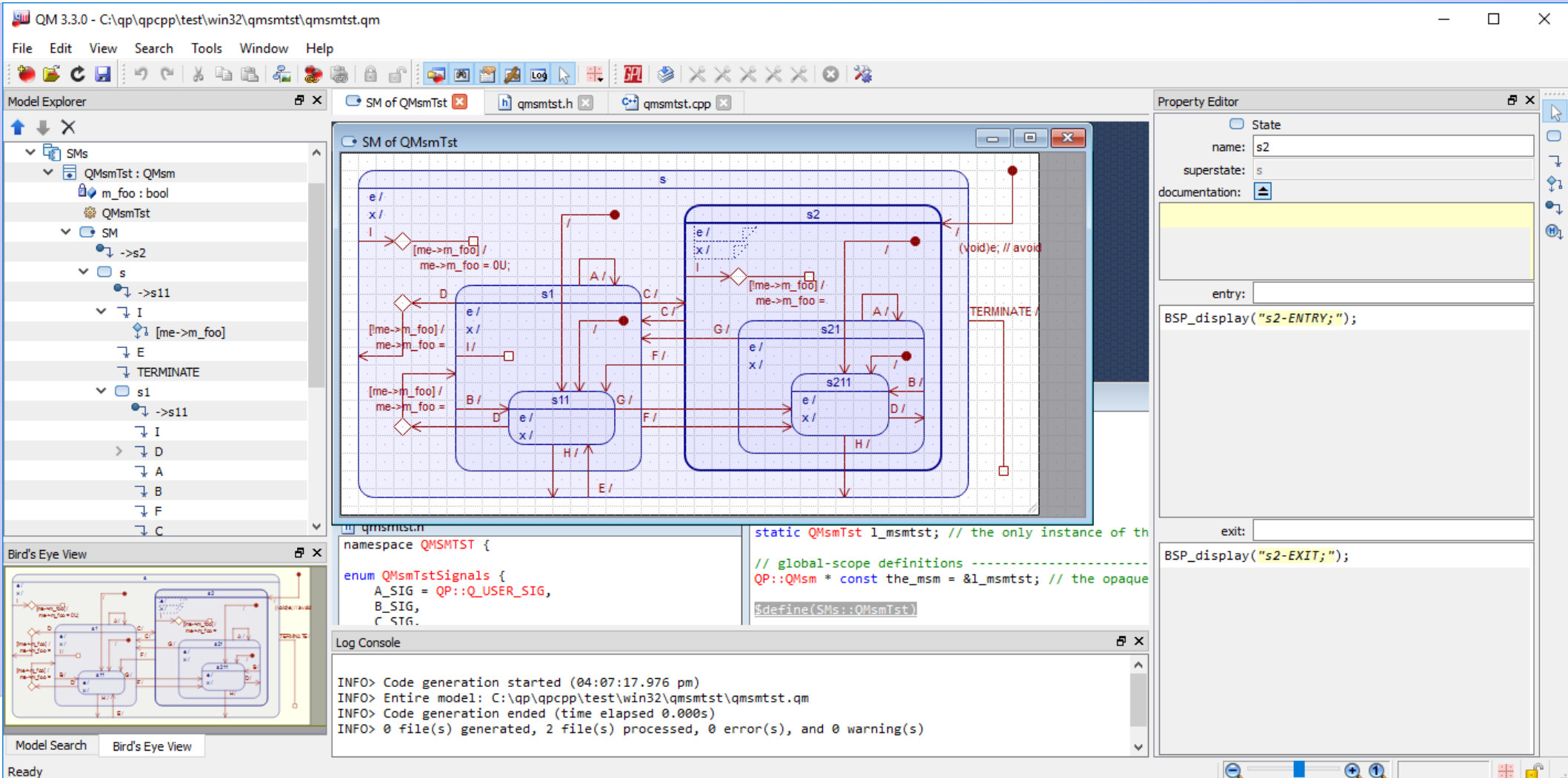
QM™ Model-Based Design Tool

- 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

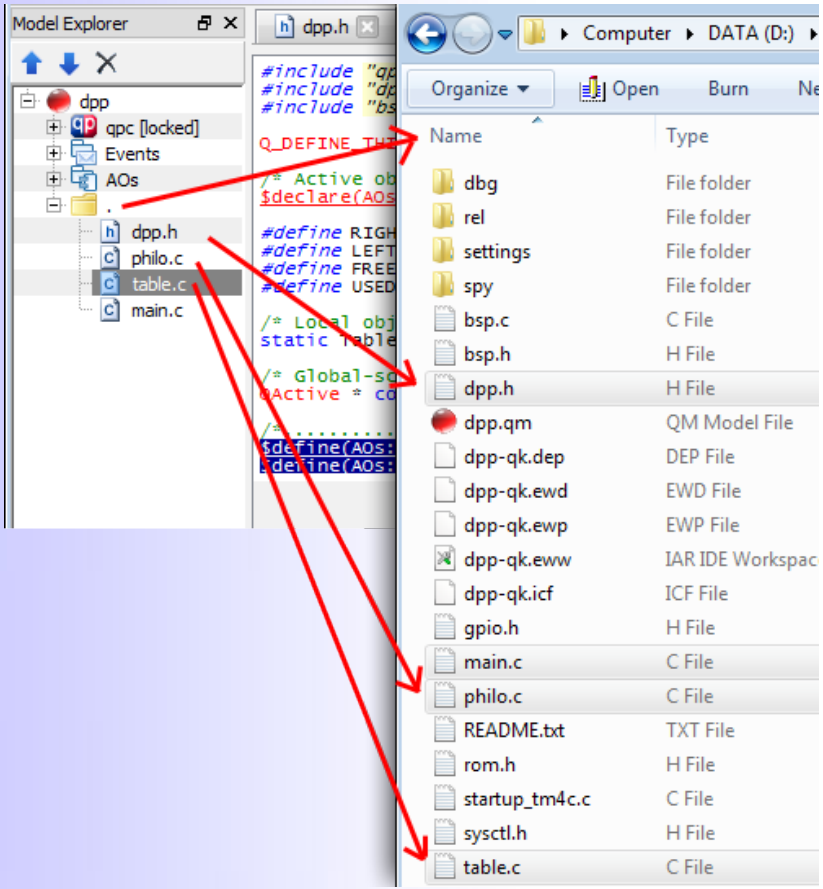
QM™ Design Philosophy

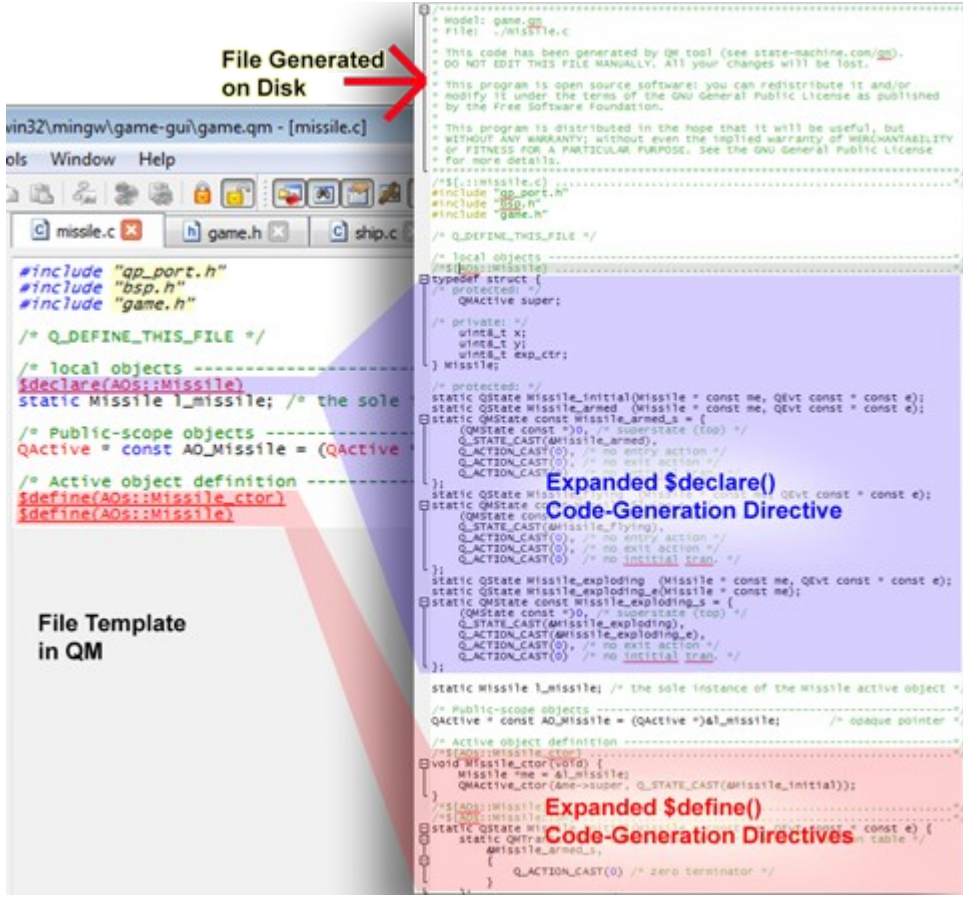
- "Low ceremony", code-centric tool (no PIM, PSM, action-languages,…)
  → Not appropriate if you need these features (80% of benefits for 20% of costs)
- Optimized for C and C++, (no attempts to support other languages)
- Optimized for QP™ (no attempts to support other frameworks)
- Forward-engineering 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**

Logical Design (Packages/Classes/State Machines)

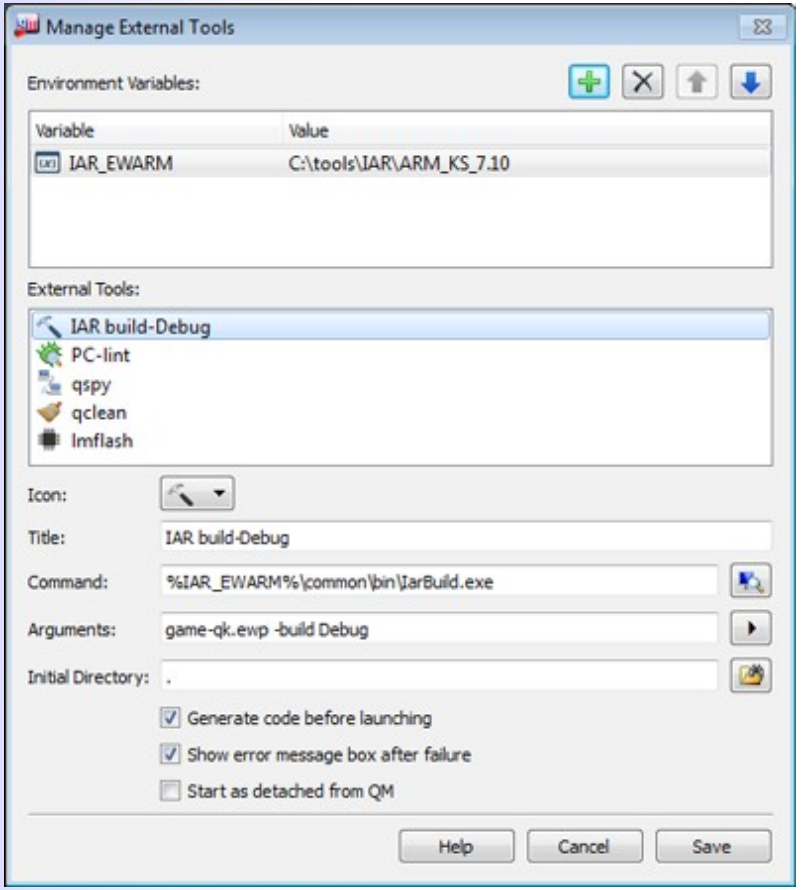


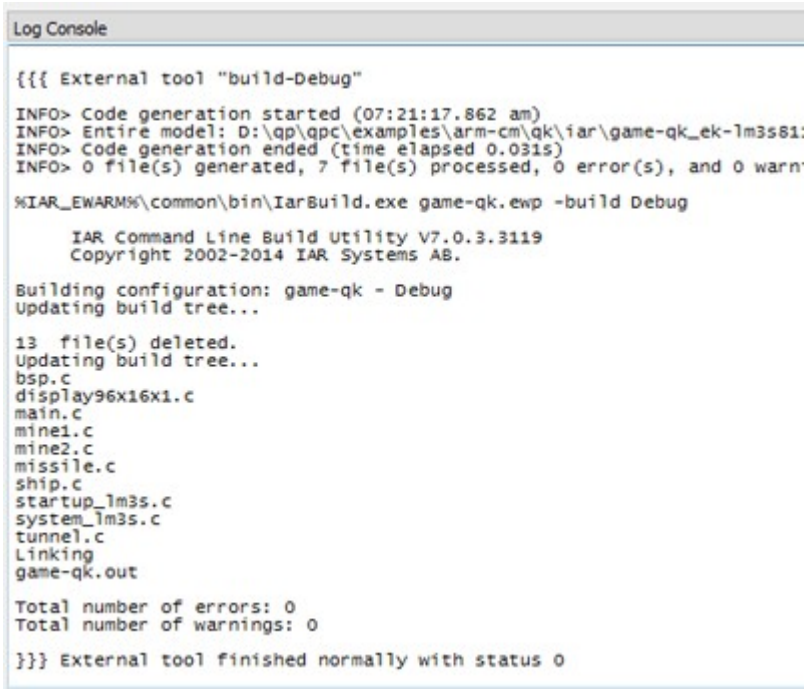
Physical Design (Directories / Files)





Extending QM™ with Command-Line Tools





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