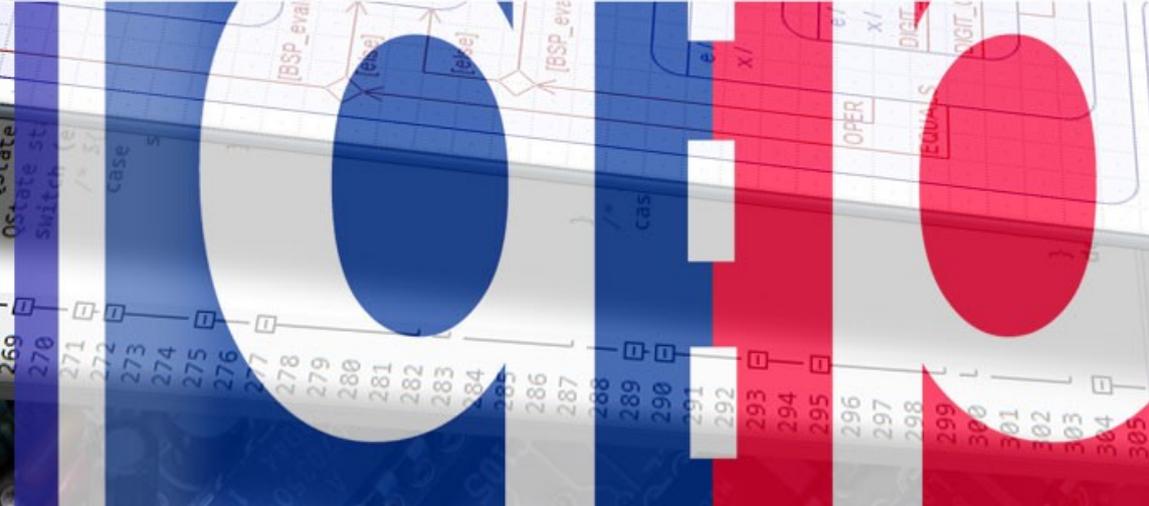


Overview of QP™ Frameworks and QM™ Modeling Tool

```
237 /* Concepts.ccc
238 /* $$(SMs):Calc::S
239 static QState C
240 BSP_message Ca
241 (void)me; /
242 return QM_EN
243 }
244 /* $$(SMs):Calc::S
245 static QState C
246 BSP_message Ca
247 (void)me; /
248 return QM_EX
249 }
250 /* $$(SMs):Calc::S
251 static QState C
252 BSP_message Ca
253 (void)me; /
254 return QM_EX
255 }
256 static stru
257 QMState
258 QAction
259 const tab
260 &Calc_
261 {
262 }
263 };
264 /* $$(SMs):Calc::S
265 BSP_message
266 return QM_
267 }
268 /* $$(SMs):Calc::S
269 static QState
270 BSP_message
271 return QM_
272 }
273 /* $$(SMs):Calc::S
274 BSP_message
275 return QM_
276 }
277 /* $$(SMs):Calc::S
278 BSP_message
279 return QM_
280 }
281 /* $$(SMs):Calc::S
282 BSP_message
283 return QM_
284 }
285 /* $$(SMs):Calc::S
286 BSP_message
287 return QM_
288 }
289 /* $$(SMs):Calc::S
290 BSP_message
291 return QM_
292 }
293 /* $$(SMs):Calc::S
294 BSP_message
295 return QM_
296 }
297 /* $$(SMs):Calc::S
298 BSP_message
299 return QM_
300 }
301 /* $$(SMs):Calc::S
302 BSP_message
303 return QM_
304 }
305 /* $$(SMs):Calc::S
306 BSP_message
307 return QM_
308 }
```

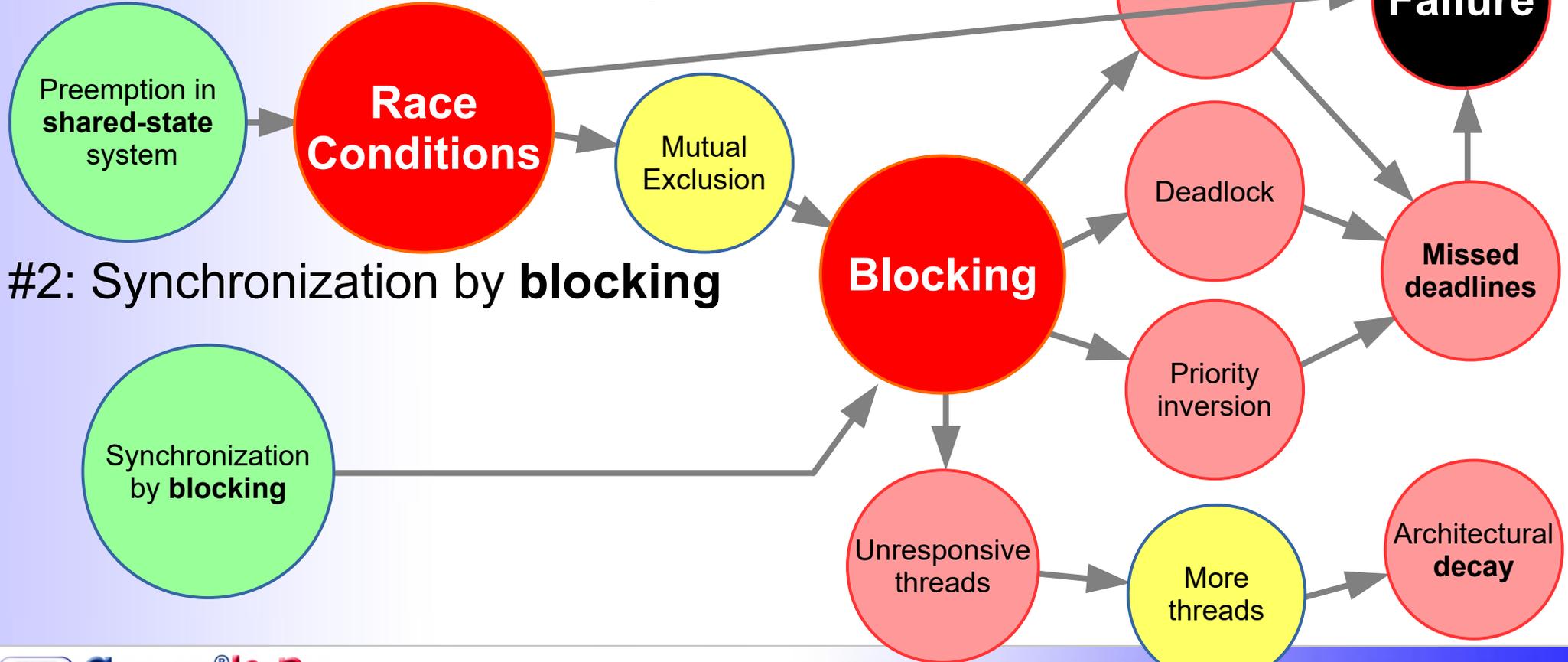


Presentation Outline

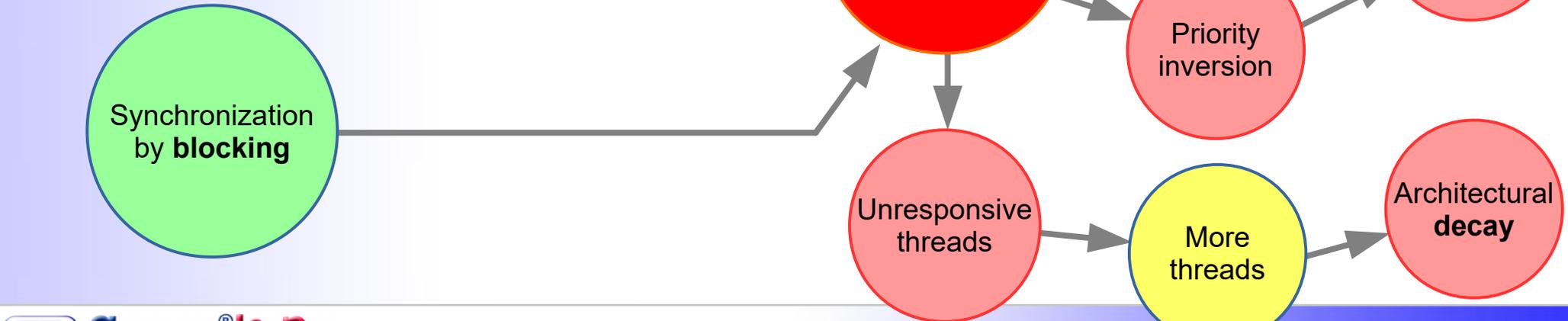
- Why is RTE programming so hard and what can we do about it?
- QP™ real-time embedded frameworks (RTEFs)
- QM™ graphical model-based design and code generating tool

Why is real-time programming hard (1)?

#1: Shared-state concurrency



#2: Synchronization by **blocking**



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

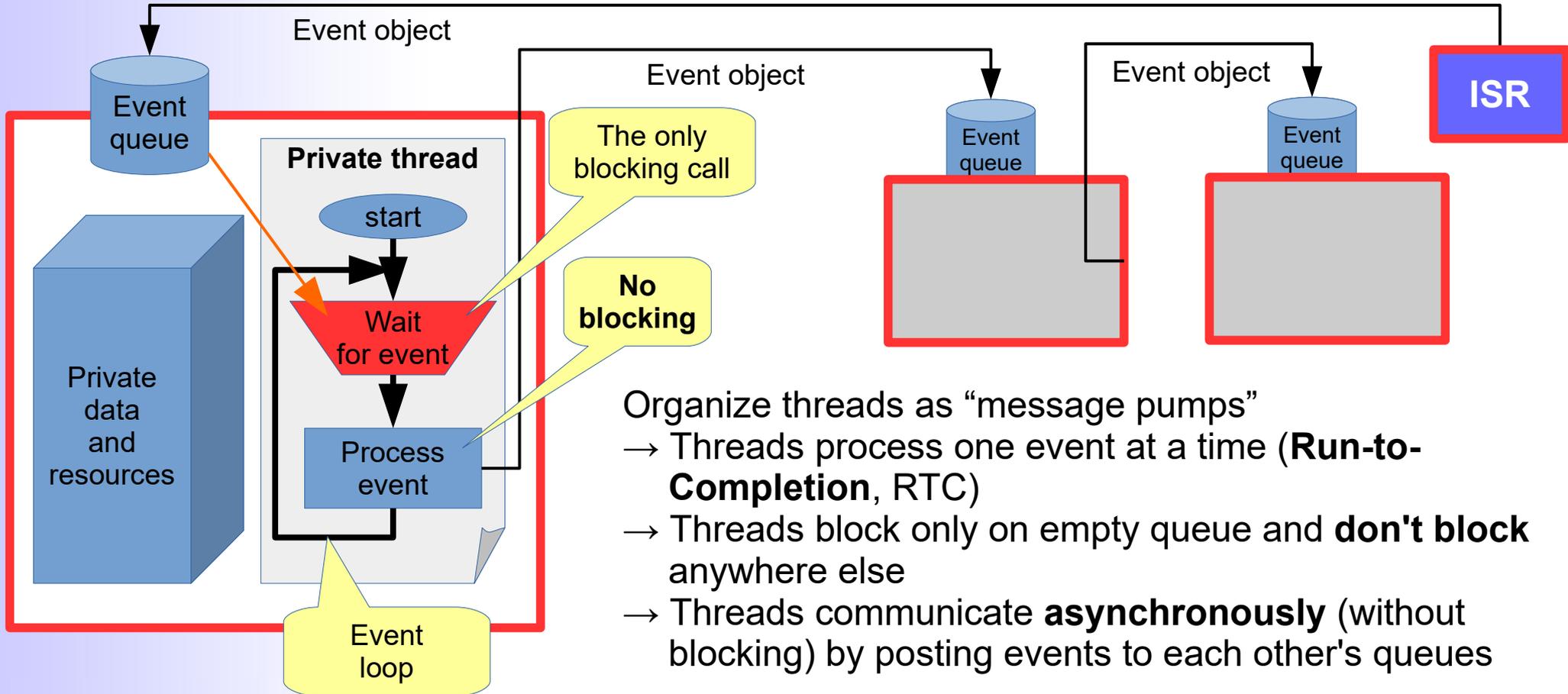
- **Active Object* (Actor*)** is an event-driven, **strictly encapsulated** software object running in its **own thread** and communicating **asynchronously** by means of **events**.
 - Not a real novelty. The concept known from 1970s, adapted to real-time in 1990s (ROOM actor), and from there into the 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



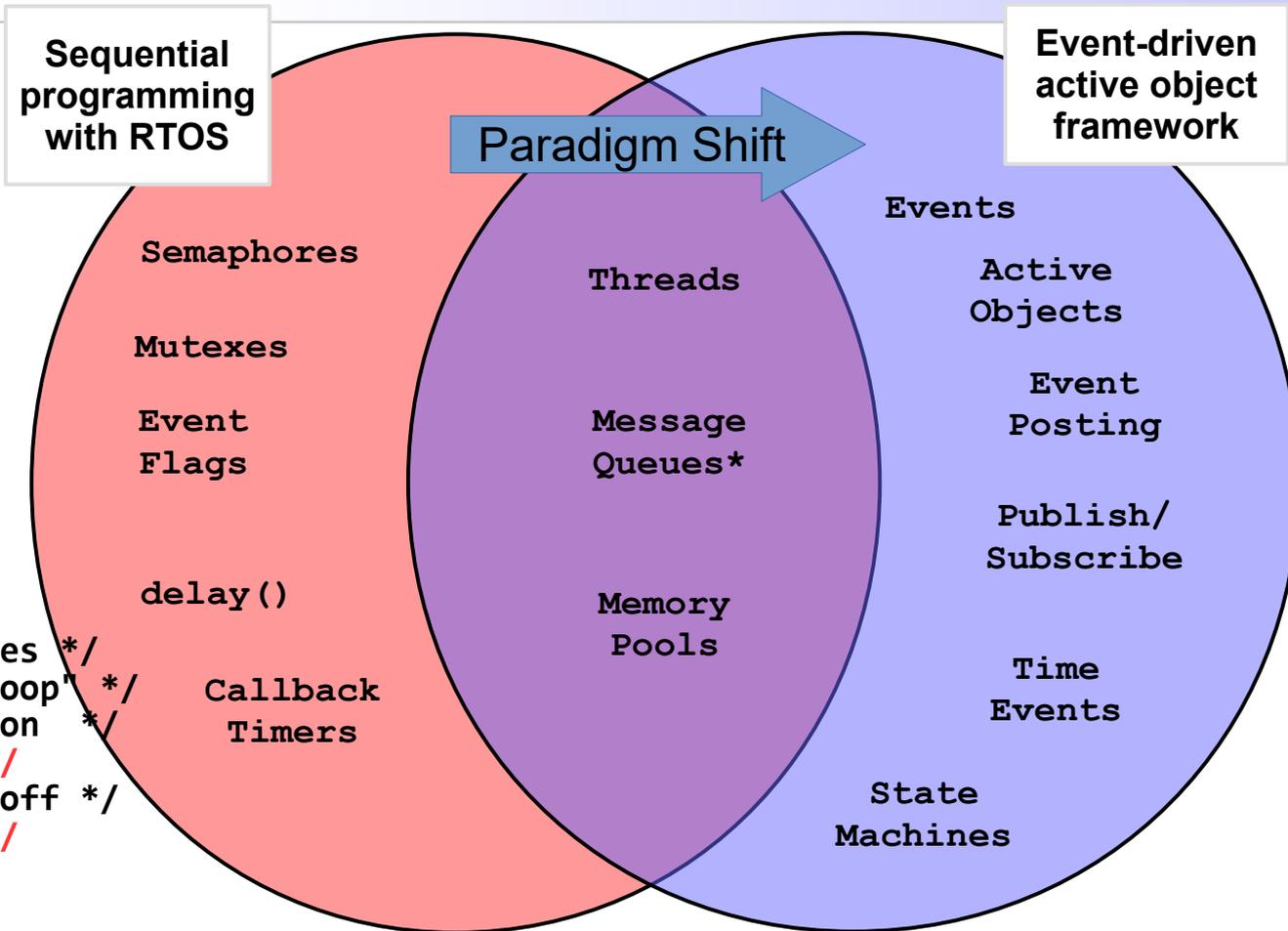
A Better Way: Active Object Framework

- 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!!! */  
}
```



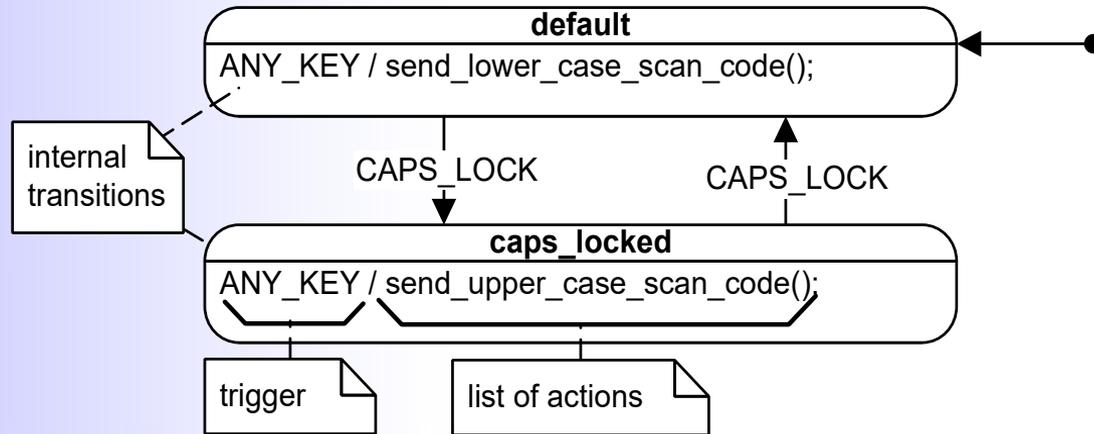
Why is real-time programming hard (2)?

- 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     Op1 = 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 = CDb1(Op1) + CDb1(Op2);
32         break;
33       case '-':
34         Op1 = CDb1(Op1) - CDb1(Op2);
35         break;
36       case '*':
37         Op1 = CDb1(Op1) * CDb1(Op2);
38         break;
39       case '/':
40         if (Op2 == 0) {
41           Error("Division by zero");
42         }
43         Op1 = CDb1(Op1) / CDb1(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

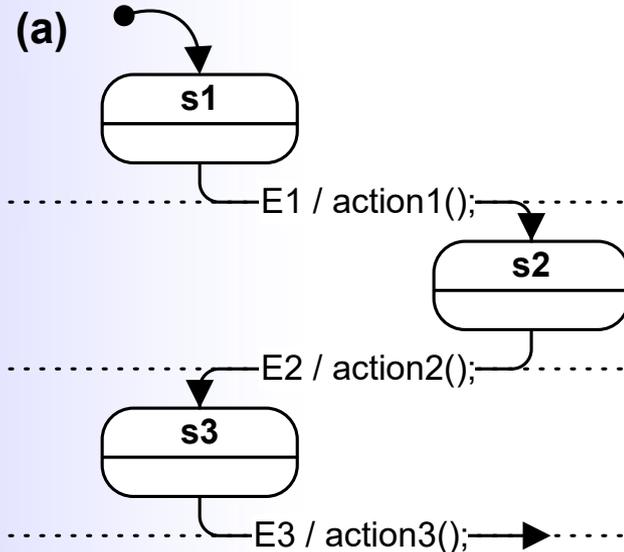


Paradigm Shift: Sequential → Event-Driven (2)

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

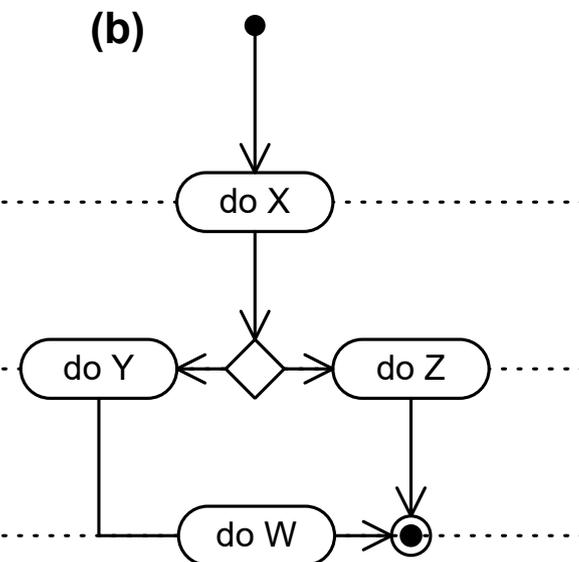
Statechart (event-driven)

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



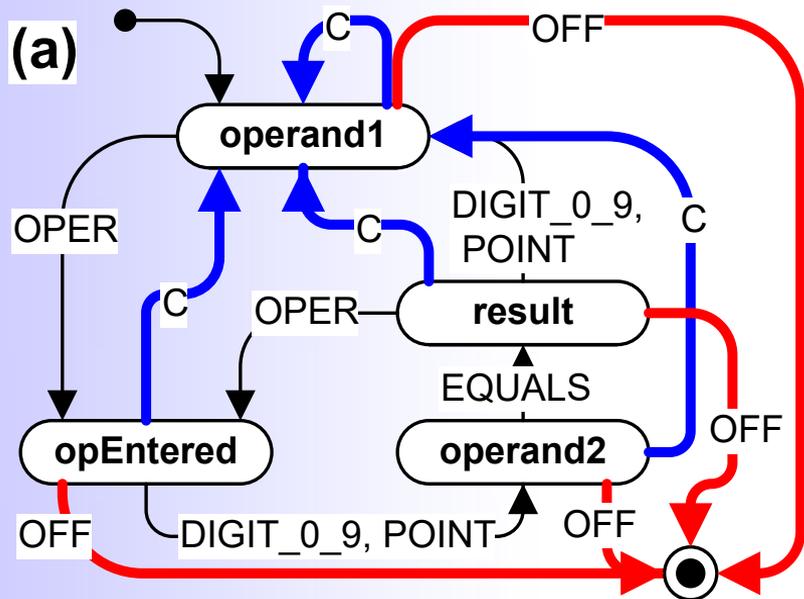
Flowchart (sequential)

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

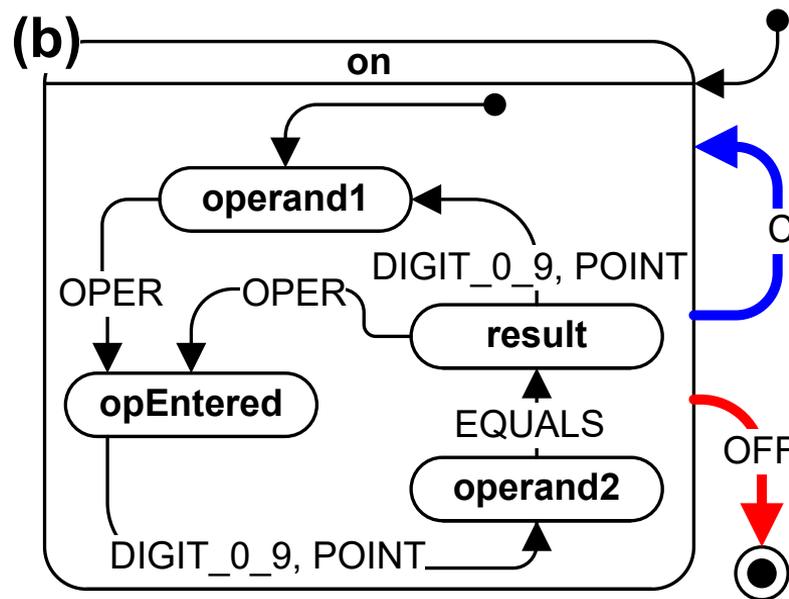


Hierarchical State Machines

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 embedded frameworks: 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)

ABB ABB Switzerland	AC PROPULSION AC Propulsion	AcuityBrands Acuity Brands	altia Alta Digital Monitoring	Amway Amway	dynamic Dynamic Controls (repeat customer)	EATON Eaton Power Business Division	Electrolux Electrolux Sweden	EMERSON Emerson Network Power	EMERSON Emerson Process Management	SIEMENS Siemens Schweiz (repeat customer)	TRW TRW Poland
AMX AMX (repeat customer)	ANALOG DEVICES Analog Devices	APC American Power Conversion	ARRIS ARRIS (Motorola Mobility)	Artesis Artesis Turkey	G-Tech G-Tech New Zealand	ATSIO ATSIO (repeat customer)	GE GE Consumer & Industrial	GENERAL DYNAMICS General Dynamics	HACH Hach (multiple divisions)	SunTech Medical SunTech Medical	WHOOPT Whoop
BIO-RAD Bio-Rad	BOSCH Bosch	BRAND Brand Hydraulics	CAMERON Cameron	CardinalHealth Cardinal Health	HAEMONETICS Haemonetics (Baxter Management Company)	hansen Hansen Medical	HEX HEX Microsystems	HITACHI Hitachi	Honeywell Honeywell (multiple divisions)	TOMIRA Tomra	Starkey Starkey Laboratories
CATERPILLAR Trimble Caterpillar Trimble	Corindus Corindus Vascular Robotics	CORNING Corning	CUBIC Cubic Defense	CUBIC Cubic Transportation	ホシザキ電機 Hosizaki Group	HsinPlas Hsin-Chin Machinery Co.	IBM lenovo IBM (Lenovo)	Imagination Imagination Technologies	INFIMED Infimed	VERATHON Verathon	tellabs Tellabs (multiple divisions)
CYBERCOM GROUP Cybercom Group	DDD DDD Diagnostic	digitalSTROM DigitalSTROM	DRS DRS Tactical Systems	DWG MORI DWG MORI	INSITU Insitu (repeat customer)	intel Intel	iRhythm iRhythm	JBL JBL	JOHN DEERE John Deere Forestry	St. JUDE MEDICAL St. Jude Medical	TYMPHANY Tymphany
Kodak alaris Kodak-Alaris (repeat customer)	communications L3 MAG	LINAK Linak	Land Transport & Machinery LTA Singapore	lytx Lytx	sonova Phonak Sonova	PHYSIO CONTROL Physio Control	proteus Proteus	QinetiQ QinetiQ (for NSA)	Qwizdom Qwizdom	TATA TATA Technologies	wirelessSEISMIC Wireless Seismic
Wantowoc Wantowoc	Medtronic Medtronic (various divisions)	METTLER TOLEDO Mettler Toledo	SMSC Microchip DMC	micronics Micronics	Raytheon Anschütz Raytheon- Anschütz	Resodyn Resodyn	RINGLY Ringly	rittmeier Rittmeyer	Roche Roche	TOSHIBA Toshiba (repeat customer)	STRATOS Stratos
MOOG Moog Medical Device Group	MOOG FERNAU Moog Fernau (repeat customer)	MOTOROLA SOLUTIONS Motorola Solutions	MOTOROLA Motorola (various divisions)	berth Berth	Rockwell Collins Rockwell Collins	RÖHDE & SCHWARZ Rohde Schwarz	SAIC Saic	Sandia National Laboratories Sandia National Laboratories	Schneider Electric Schneider Electric	VISURAY Visuray	tm4 TM4 (Electrodynamics Systems)
NANO MR NanoMR (DNA Electronics)	NASA NASA	NETGEAR Netgear	nielsen Nielsen (Nielsen)	NOV NOV	SEA SEA Schloss Systeme	SECOM Secom Japan	SECURITON Securiton	Sound Sound Environmental Products	STEP STEP Electric Shanghai	STANLEY Stanley Healthcare	SAFRAN SAFRAN Vestronix
Naval Research Laboratory Naval Research Laboratory	NTREPID Ntrepid	OCEANEERING Oceaneering	PATTON Patton Insp	PHILIPS Philips Healthcare (repeat customer)	SIEMENS Siemens Schweiz (repeat customer)	St. JUDE MEDICAL St. Jude Medical	STANLEY Stanley Healthcare	Starkey Starkey Laboratories	STRATOS Stratos	TeleCommunication Systems TeleCommunication Systems	Whisper Whisper Sweden

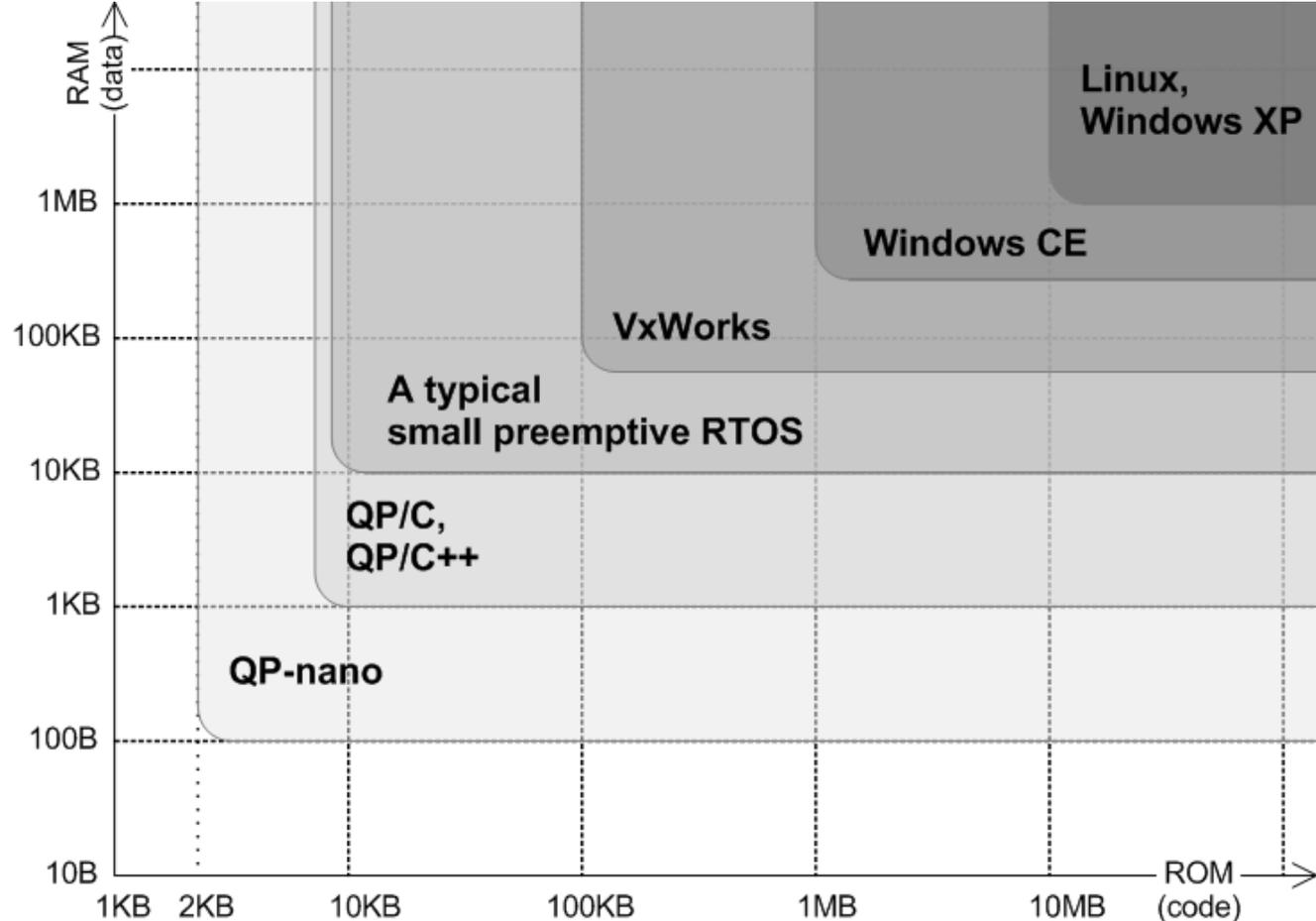
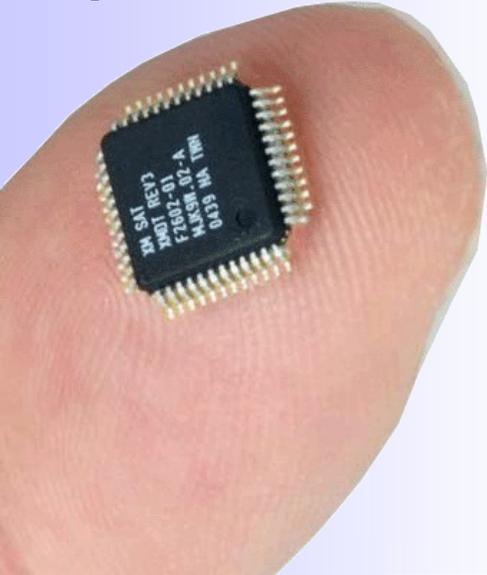


QP™ Framework Family Features

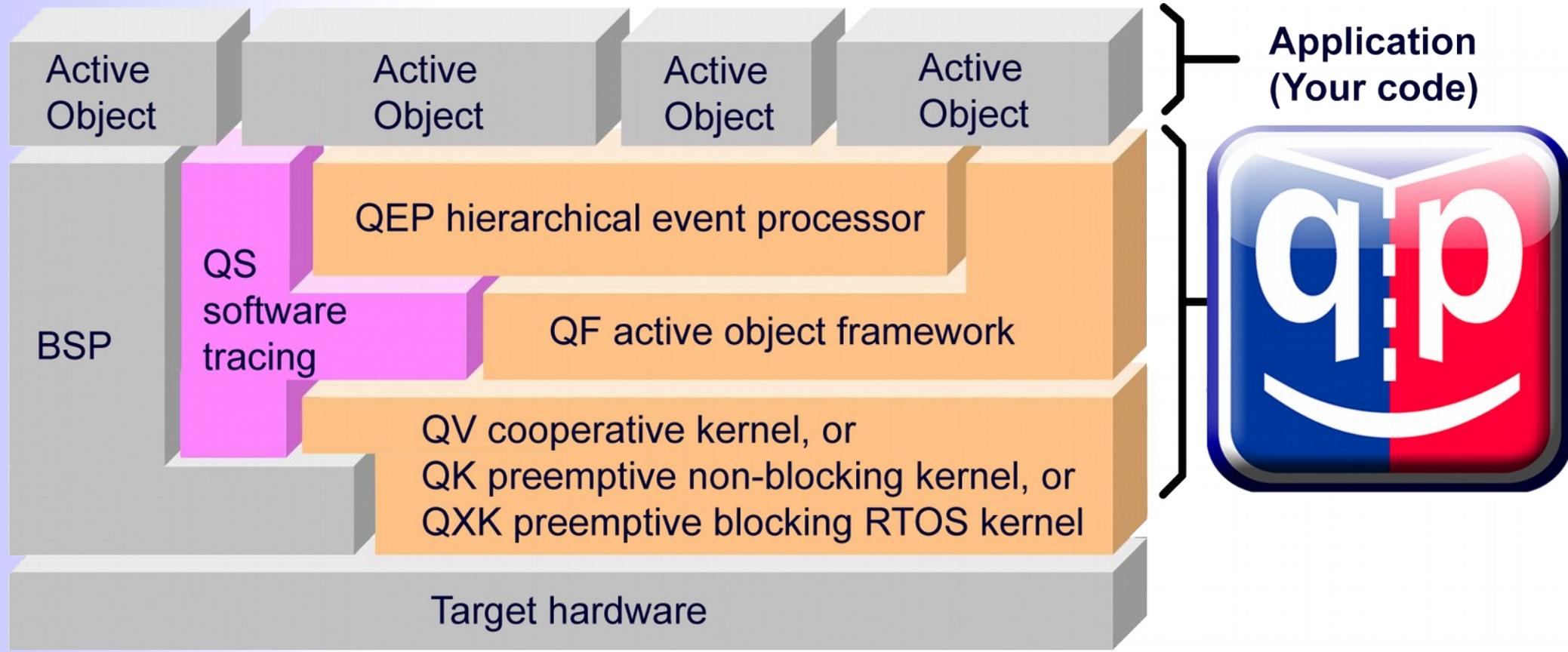
Feature	QP/C	QP/C++	QP-nano
Code (ROM) / Data (RAM) footprint	4KB / 1KB	5KB / 1KB	2KB / 0.5KB
Maximum number of active objects	64	64	8
Hierarchical state machines	✓	✓	✓
Events with arbitrary parameters	✓	✓	32-bits
Event pools and automatic event recycling	✓	✓	✗
Direct event posting	✓	✓	✓
Publish-Subscribe	✓	✓	✗
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 3 rd -party RTOS	✓	✓	✗

QP™ vs. RTOS Memory Footprint

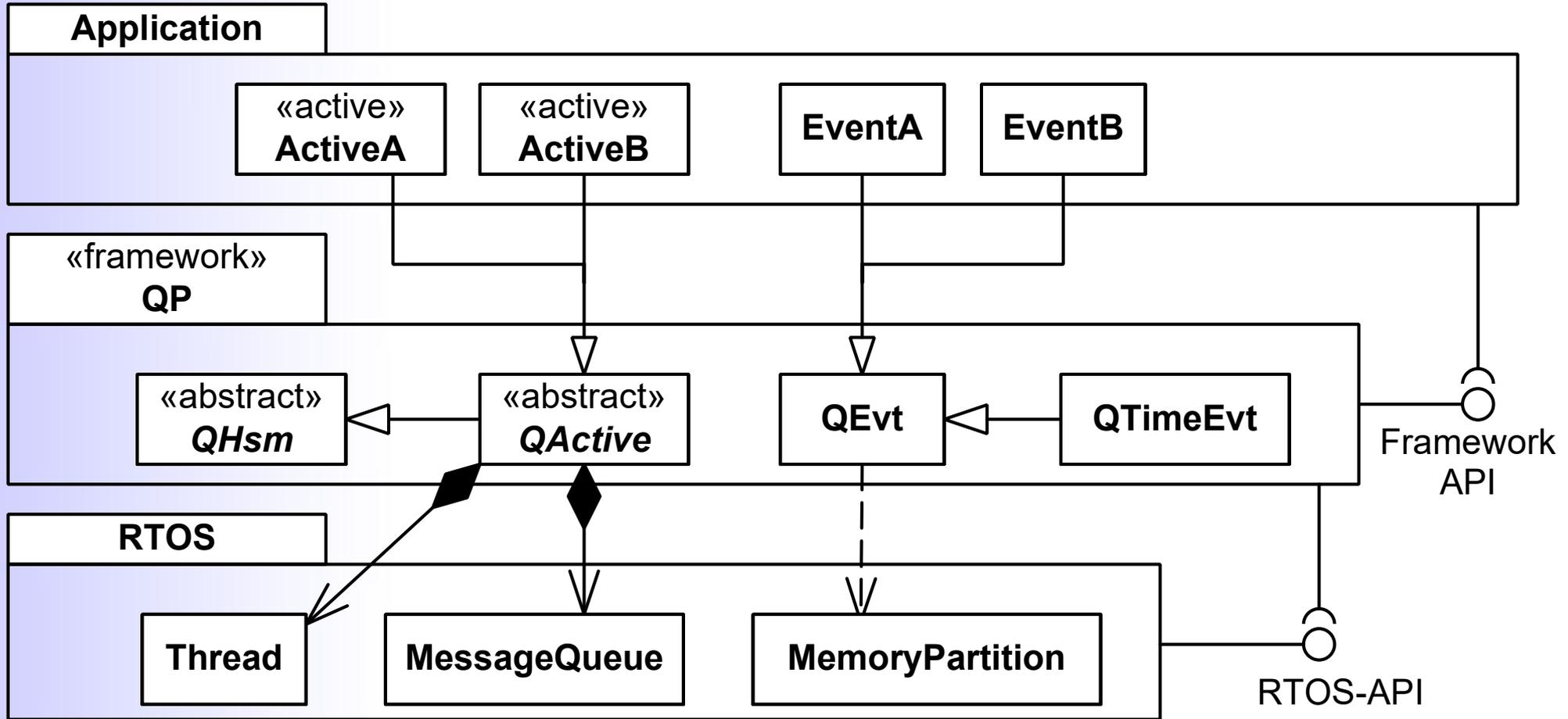
QP frameworks fit into smaller RAM, because event-driven programming style uses less **stack space**



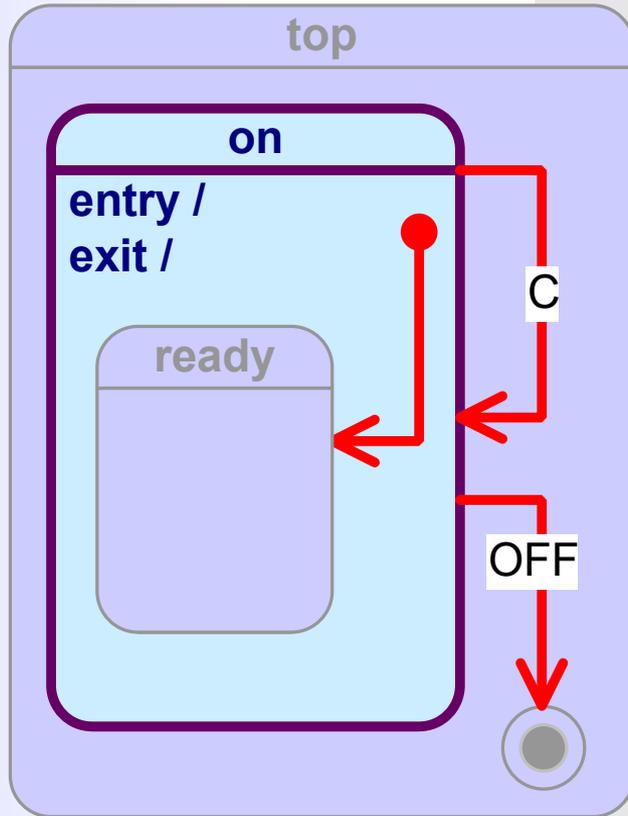
QP™ Components and Layers



QP™ Package and Class View

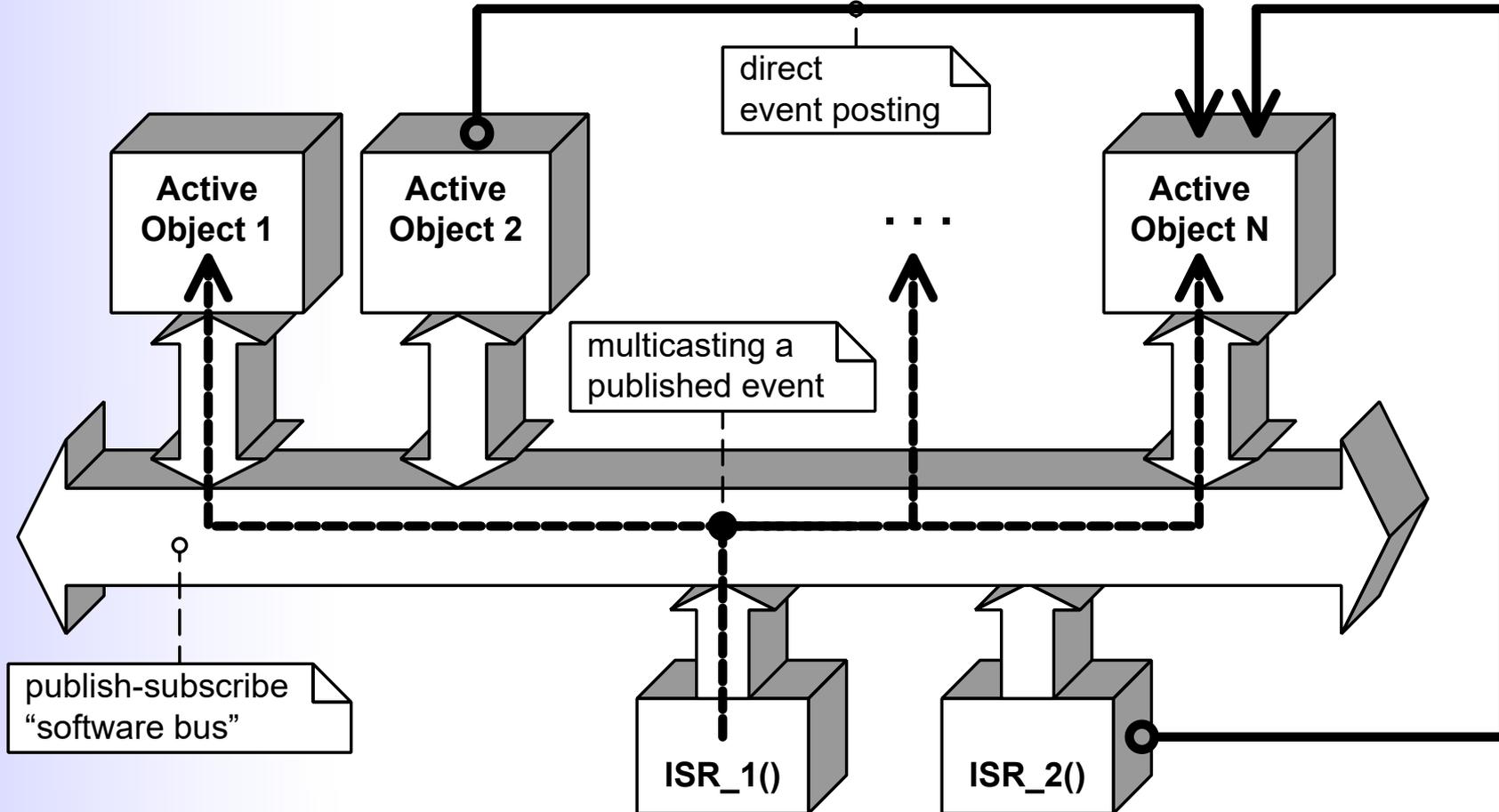


QEP Hierarchical Event Processor

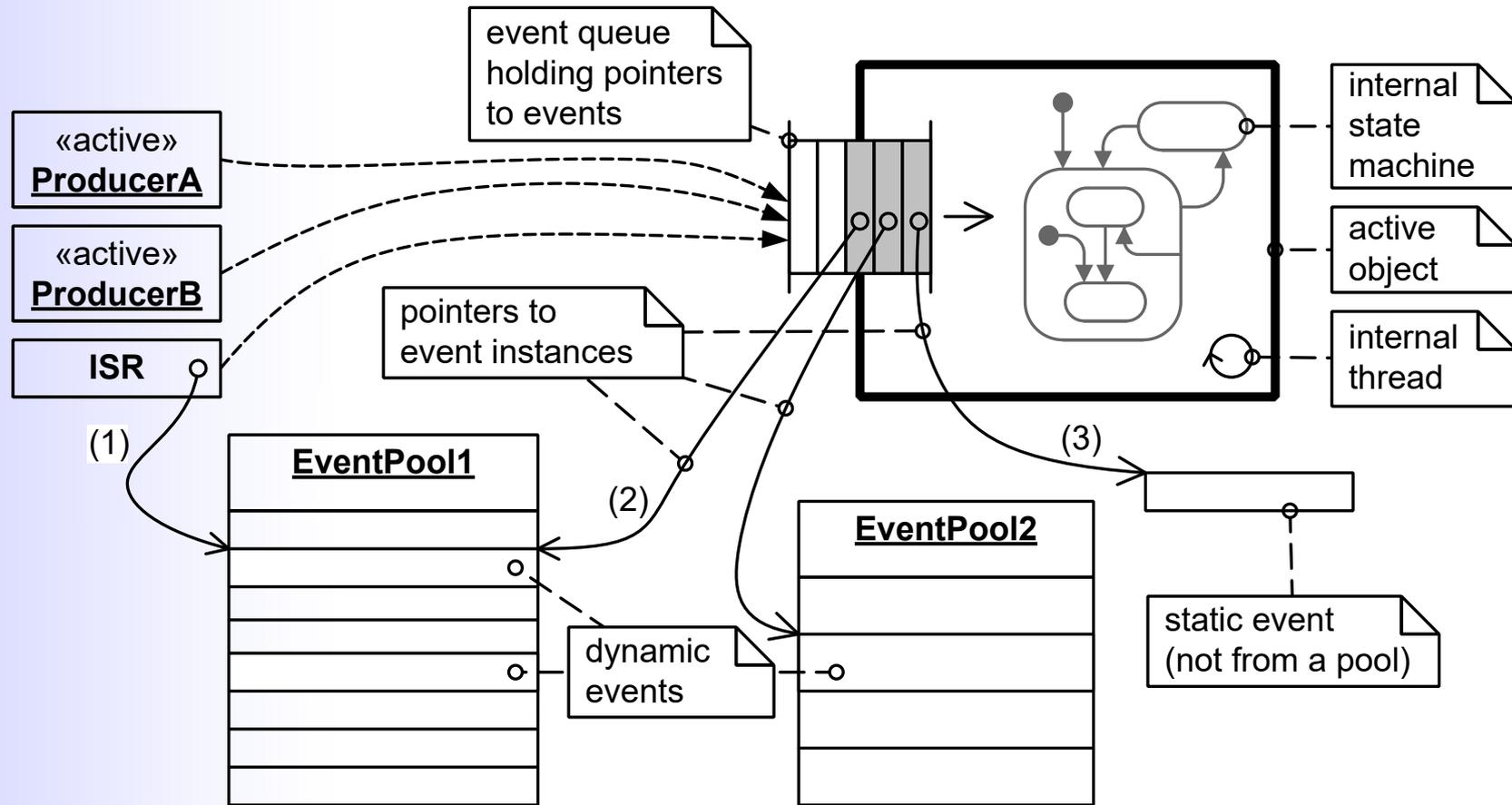


```
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG:    /* entry action */
            BSP_message("on-ENTRY");
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG:     /* exit action */
            BSP_message("on-EXIT");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

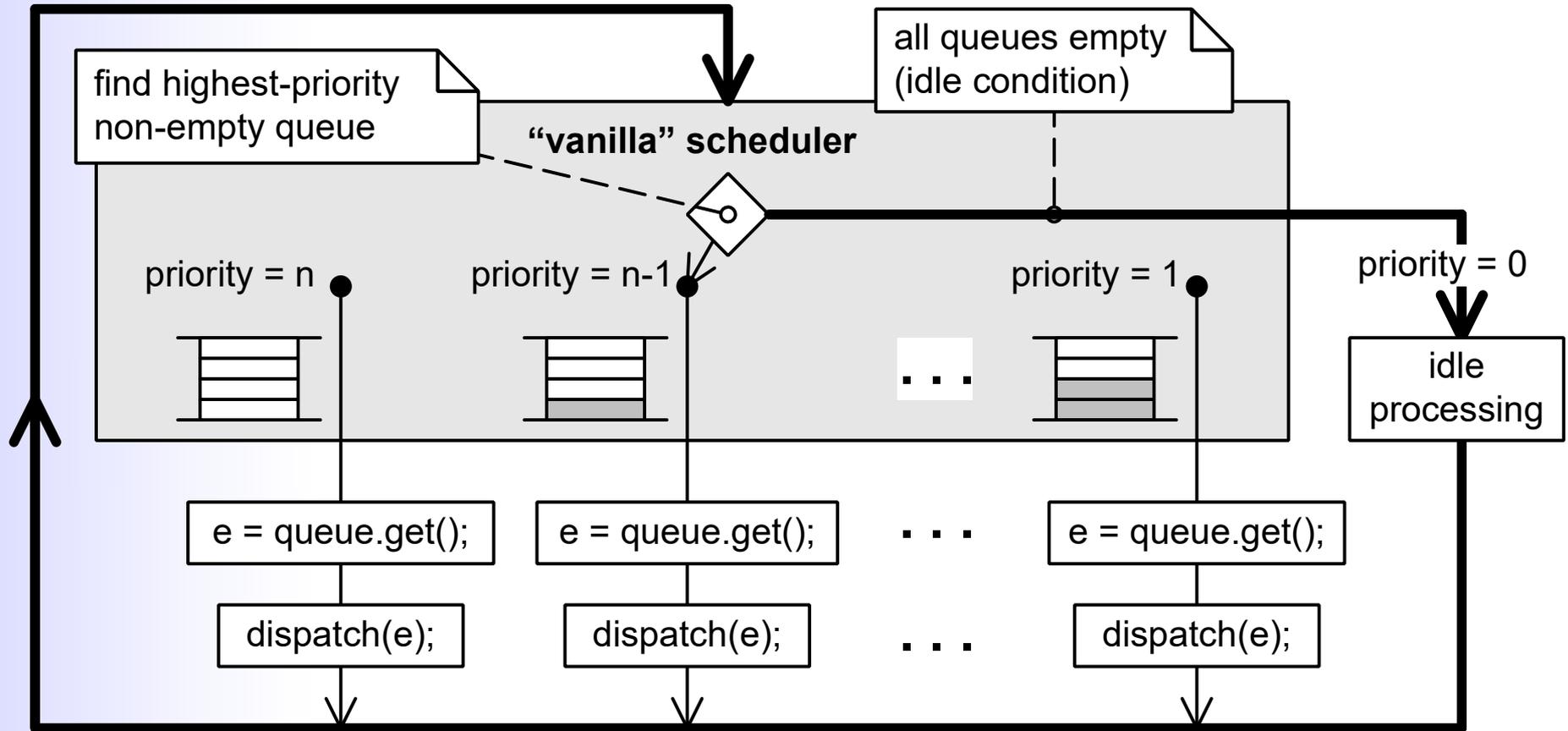
QF AO Framework – “Software Bus”



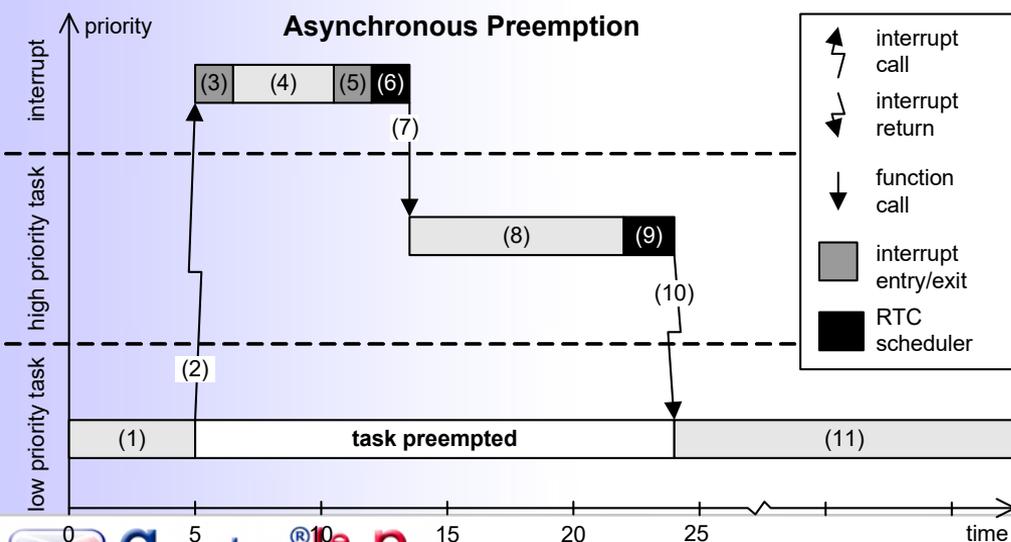
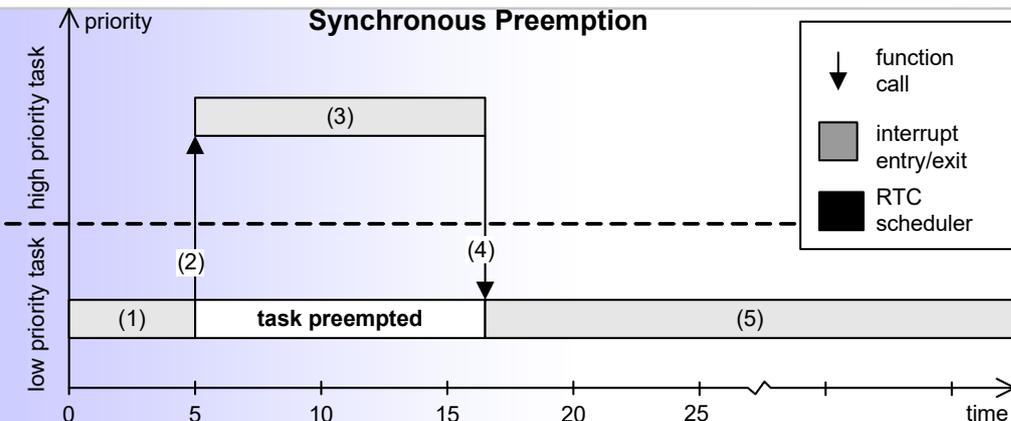
QF AO Framework – “Zero Copy” Event Delivery



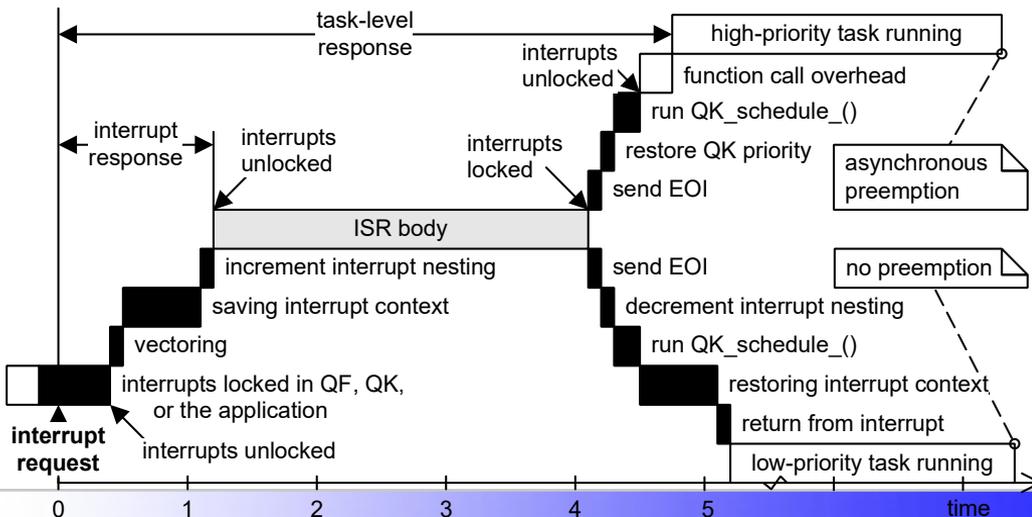
QV Cooperative Kernel



QK Preemptive, Non-Blocking Kernel

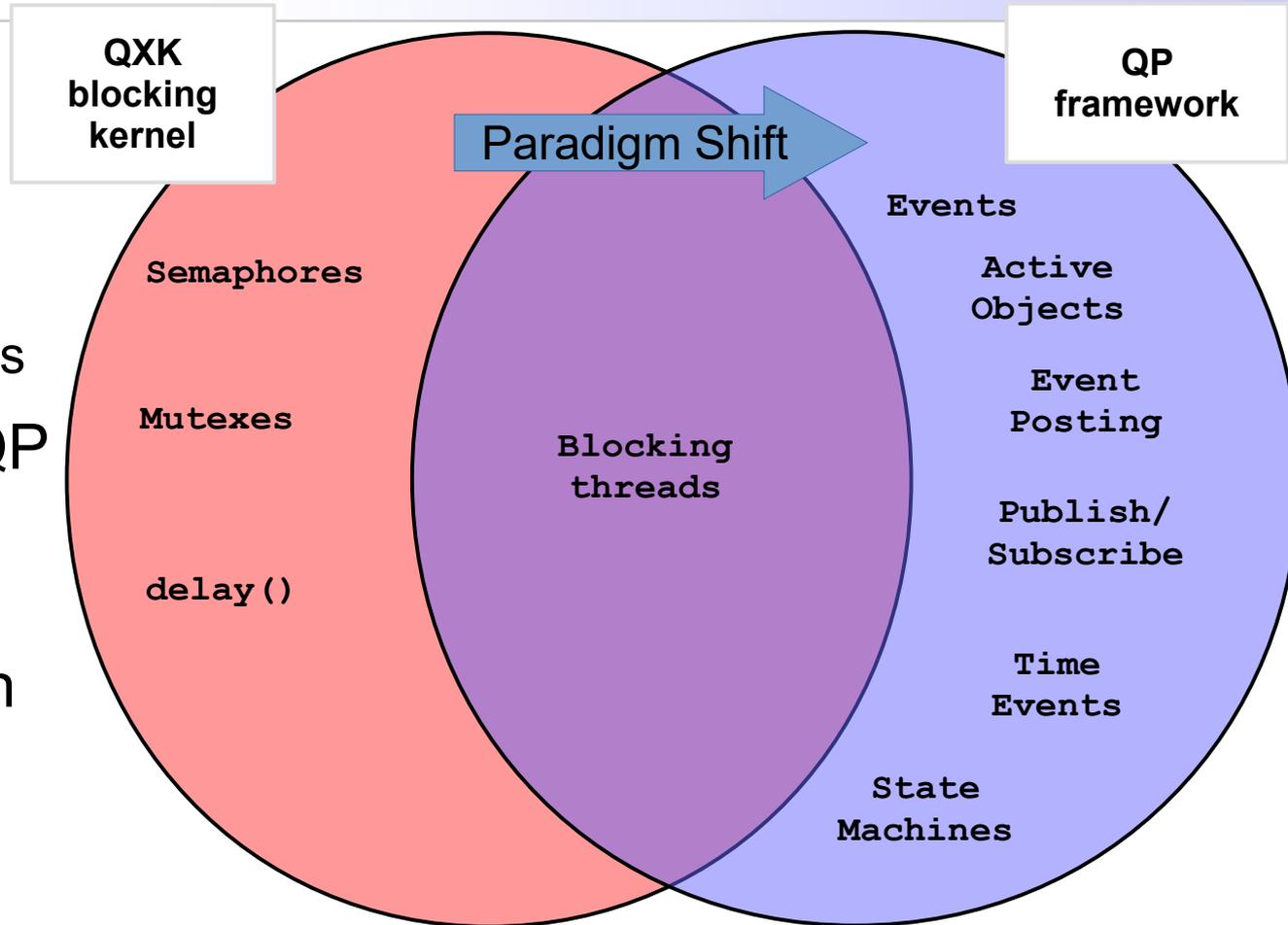


- Preemptive priority-based kernel
 - Meets all requirements of Rate Monotonic Analysis (RMA)
 - Run-to-Completion Kernel
- **Cannot block in-line**
- **Single stack operation (like ISRs)**



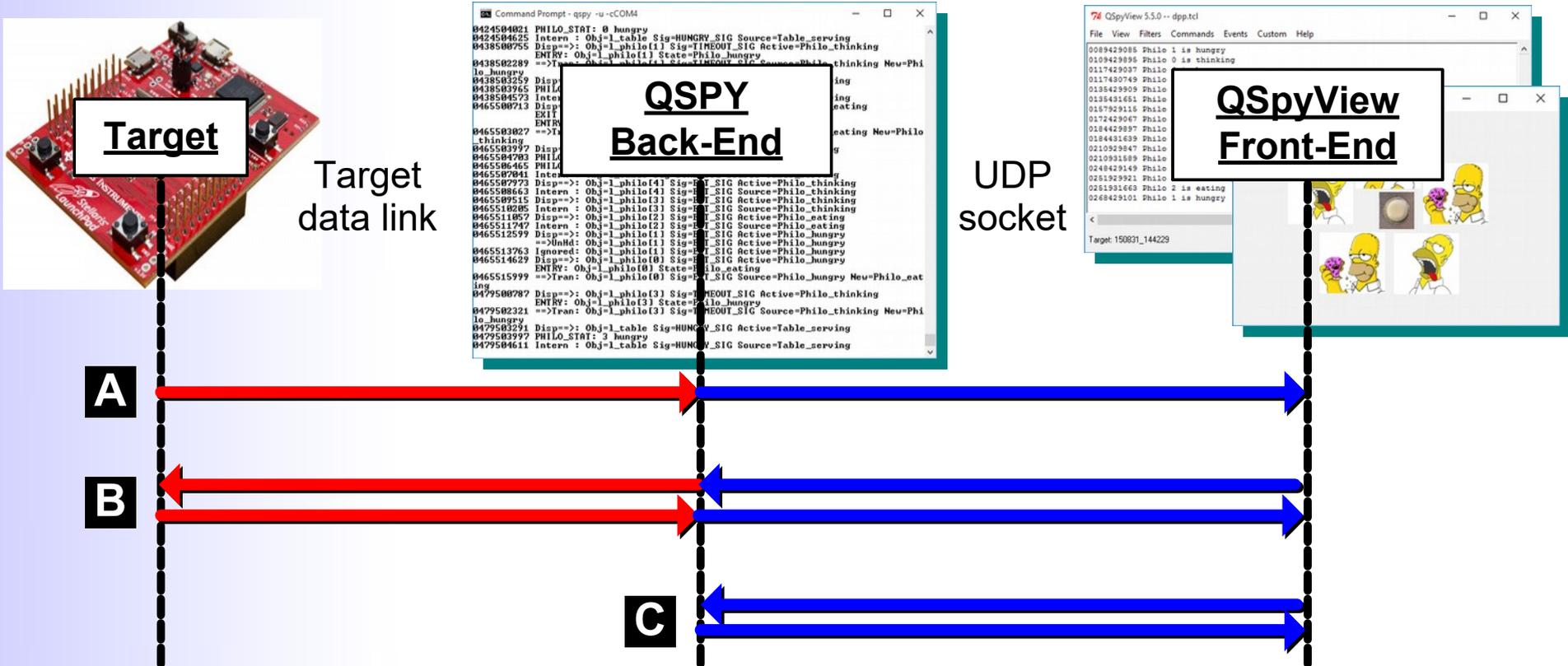
QXK Preemptive, Blocking Kernel

- 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

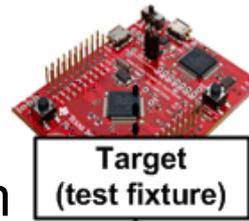
- You need to observe system **live**, not stopped in a debugger



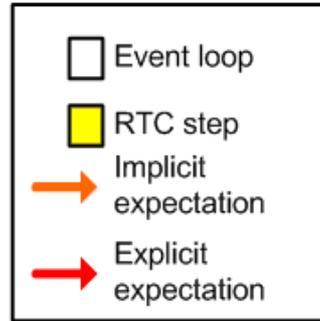
QUTest Unit Testing Harness

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

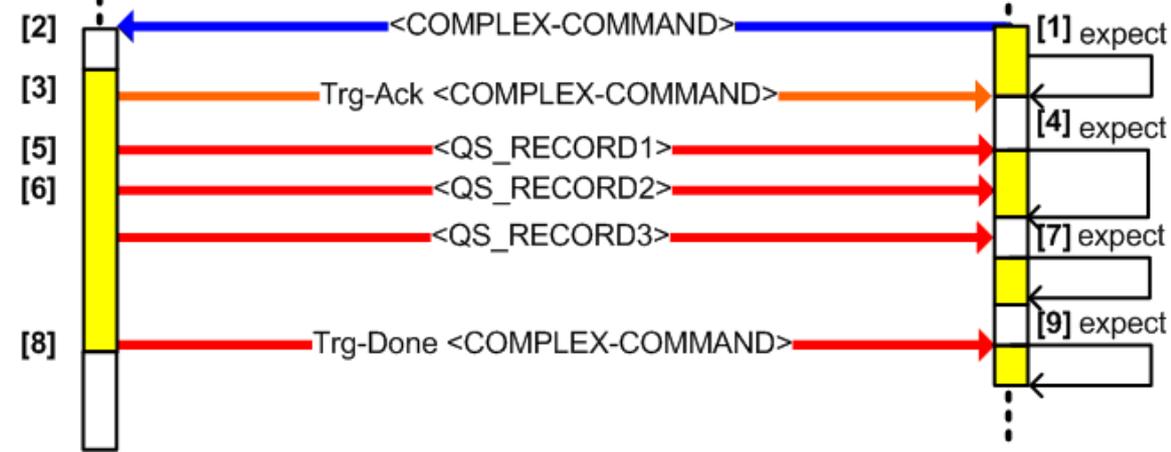
- Separates CUT execution from checking the test assertions
- Small, reusable test fixture in the Target (C or C++ code)
- Driving the tests and checking correctness on the Host
- **Python** and Tcl test scripting
- Specifically suitable for **event-driven** systems (simplifies “mocking”)



Target (test fixture)



QUTest (test script)



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

The screenshot displays the QSpyView GUI Front-End, which consists of two main windows:

- Command window running QSPY:** This window shows a terminal output of the QSPY application. The output includes status messages for five philosophers (PHILO_0 to PHILO_4) and their current states (thinking, eating, hungry). The output is as follows:

```
2981504673 PHILO_STAT: 0 thinking
2981506441 PHILO_STAT: 1 eating
2981507...
2981509...
2981509...
2981510...
2981511...
2981512...
2981513...
2981514...
2981515...
2981516...
2996500...
2996503...
2996504...
2996504...
2996506...
2996507...
2996508...
2996509...
2996510...
2996511...
2996511909 Disp==>: Obj=1_philo[2] Sig=EAT_SIG Active=Philo_hungry
2996513073 Ignored: Obj=1_philo[2] Sig=EAT_SIG Active=Philo_hungry
2996513925 Disp==>: Obj=1_philo[1] Sig=EAT_SIG Active=Philo_eating
2996514615 Intern : Obj=1_philo[1] Sig=EAT_SIG Source=Philo_eating
2996515481 Disp==>: Obj=1_philo[0] Sig=EAT_SIG Active=Philo_thinking
2996516171 Intern : Obj=1_philo[0] Sig=EAT_SIG Source=Philo_thinking
```
- Canvas window of QSpyView customized for the DPP application:** This window displays a graphical representation of the philosophers. It features five Homer Simpson characters in various states: thinking, eating, and hungry. A button is visible for pausing/resuming the granting of forks. The canvas window is titled "7% Canvas".

Annotations in the image provide additional context:

- "Command window running QSPY" points to the terminal window.
- "button to pause/resume granting the forks" points to a button in the canvas window.
- "QSpyView GUI Front-End communicating with QSPY via a UDP socket" points to the interface between the two windows.
- "Canvas window of QSpyView customized for the DPP application" points to the canvas window.

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)

The screenshot displays the QM 3.3.0 software interface. The main window shows a state machine diagram titled "SM of QMsmTst" with states s, s1, s2, s11, s21, and s211. Transitions are labeled with events and actions, such as "e / x /" and "[me->m_foo] / me->m_foo = 0U;".

The left sidebar shows the Model Explorer with a tree structure: SMs > QMsmTst: QMsm > m_foo: bool > QMsmTst > SM > ->s2 > s > ->s11 > I > [me->m_foo] > E > TERMINATE > s1 > ->s11 > I > D > A > B > F > C.

The bottom-left pane shows a Bird's Eye View of the state machine diagram.

The bottom-middle pane shows the code editor with the following C++ code:

```
namespace QMSMTST {  
enum QMsmTstSignals {  
    A_SIG = QP::Q_USER_SIG,  
    B_SIG,  
    C_SIG.  
};  
static QMsmTst l_msmstst; // the only instance of th  
// global-scope definitions -----  
QP::QMsm * const the_msm = &l_msmstst; // the opaque  
#define(SMs::QMsmTst)
```

The bottom-right pane shows the Log Console with the following output:

```
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)
```

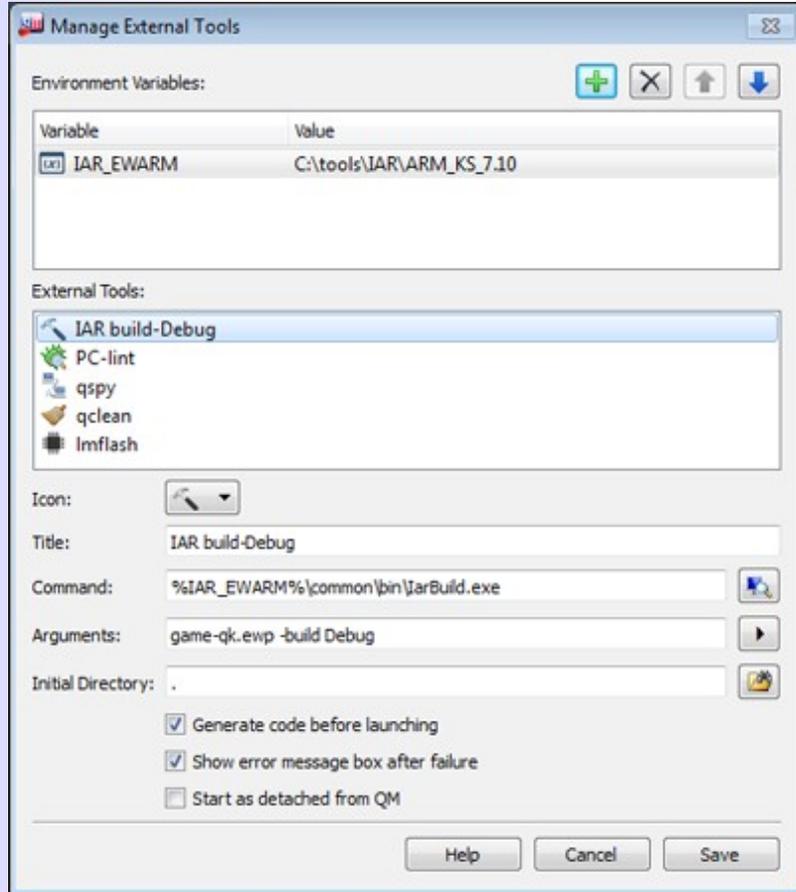
The right sidebar shows the Property Editor for state s2, with fields for name, superstate, documentation, entry, and exit.

Physical Design (Directories / Files)

The image shows two windows side-by-side. On the left is the 'Model Explorer' window showing a project tree for 'dpp'. The tree includes folders like 'Events' and 'AOs', and files like 'dpp.h', 'philo.c', 'table.c', and 'main.c'. On the right is a 'File Explorer' window showing the contents of the 'DATA (D:)' drive. It lists various files and folders such as 'dbg', 'rel', 'settings', 'spy', 'bsp.c', 'bsp.h', 'dpp.h', 'dpp.qm', 'dpp-qk.dep', 'dpp-qk.ewd', 'dpp-qk.ewp', 'dpp-qk.eww', 'dpp-qk.icf', 'gpio.h', 'main.c', 'philo.c', 'README.txt', 'rom.h', 'startup_tm4c.c', 'sysctl.h', and 'table.c'. Red arrows point from the 'dpp.h' file in the Model Explorer to the 'dpp.h' file in the File Explorer, and from the 'table.c' file in the Model Explorer to the 'table.c' file in the File Explorer.

The image shows a code editor window displaying generated code. A red arrow points to the top of the code with the text 'File Generated on Disk'. Another red arrow points to a specific line of code with the text 'Expanded \$declare() Code-Generation Directive'. A third red arrow points to another line of code with the text 'Expanded \$define() Code-Generation Directives'. The code includes comments and directives such as `Q_DECLARE_THIS_FILE`, `$declare(AOS::Missile)`, and `$define(AOS::Missile)`. The code is for a missile object in a QM (Quantum Machine) environment.

Extending QM™ with Command-Line Tools



```
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-1m3s81
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_1m3s.c
system_1m3s.c
tunnel.c
Linking
game-qk.out

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

}}}
```

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