Modern Embedded Software

Overview of QP™ Real-Time Embedded Frameworks and QM™ Model-Based Design Tool
Presentation Outline

- Why is RTE programming so hard and what can we do about it?
- QP™ real-time embedded frameworks (RTEFs)
- QM™ model-based design (MBD) and code generation tool
Why is real-time programming hard (1)?

#1: **Shared-state** concurrency

- Preemption in shared-state system
- Race Conditions
- Mutual Exclusion

#2: Synchronization by **blocking**

- Synchronization by blocking
- Blocking
- Starvation
- Deadlock
- Priority inversion
- Missed deadlines
- Architectural decay
- More threads
- Unresponsive threads
- Architectural decay

https://www.state-machine.com/
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

https://www.state-machine.com/
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: Real-Time Embedded 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

```c
/* 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 event-driven 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
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

```java
ANY_KEY / send_lower_case_scan_code();
```

```java
ANY_KEY / send_upper_case_scan_code();
```

https://www.state-machine.com/
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
Input-driven state machines are NOT driven by events:
→ combination of polling for events and state machine logic
→ often called from “superloops” (while(1) loops)
→ transitions have only guard conditions (if(guard) statements in the code)

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™ model-based design (MBD) and code generation tool
QP™ Real-Time Embedded Frameworks

- Family of frameworks for deeply embedded real-time systems: QP/C, QP/C++, QP-nano
  - Combines 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++

https://www.state-machine.com/
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

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

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

https://www.state-machine.com/
QP™ Sub-Components

- Active Object
- QP hierarchical event processor
- QF active object framework
- QV cooperative kernel, or
  QK preemptive non-blocking kernel, or
  QXK preemptive blocking RTOS kernel
- Target hardware

Application (Your code)

https://www.state-machine.com/
QP™ Package and Class View

Application

«active»
ActiveA

«active»
ActiveB

EventA

EventB

«framework»
QP

«abstract»
QHsm

«abstract»
QActive

QEvt

QTimeEvt

RTOS

Thread

MessageQueue

MemoryPartition

Framework API

RTOS-API

https://www.state-machine.com/
QEP Hierarchical Event Processor

```c
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 Framework – “Software Bus”

Active Object 1

Active Object 2

Active Object N

multicasting a published event

direct event posting

publis-publish-subscribe‘software bus’

ISR_1()

ISR_2()
QF Framework – “Zero Copy” Event Delivery

- "active" ProducerA
- "active" ProducerB
- ISR

EventPool1
- event queue holding pointers to events
- pointers to event instances

EventPool2
- dynamic events
- static event (not from a pool)

https://www.state-machine.com/
find highest-priority non-empty queue

“vanilla” scheduler

priority = n

priority = n-1

priority = 1

all queues empty (idle condition)

priority = 0

idle processing

e = queue.get();
dispatch(e);

e = queue.get();
dispatch(e);

...
QM™ Preemptive, Non-Blocking RT-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 RT-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.

https://www.state-machine.com/
QS/Spy™ 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

https://www.state-machine.com/
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™ model-based design (MBD) and code generation 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 complexity to support other languages)
- Optimized for QP™ (no complexity to support other frameworks)
- Forward-engineering only (no complexity 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**
Physical Design (Directories / Files)

File Generated on Disk

File Template in QM

Expanded $define()
Code-Generation Directive

Expanded $declare()
Code-Generation Directive

https://www.state-machine.com/
Extending QM™ with Command-Line Tools

https://www.state-machine.com/
Welcome to the 21st Century!

- Experts avoid shared-state concurrency and blocking
- Experts use the event-driven Active Object design pattern
- Experts use hierarchical state machines instead of “spaghetti code”
- Event-driven active objects and state machines require a paradigm shift from sequential to event-driven programming
- QP™ real-time embedded frameworks (RTEFs) provide a lightweight, reusable architecture based on the AO pattern and hierarchical state machines for deeply embedded systems, such as single-chip MCUs
- QM™ model-based design tool eliminates manual coding of HSMs