Beyond the RTOS
Reusable Event-Driven Architecture for Embedded Systems

Miro Samek
Quantum Leaps, LLC

www.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
In the beginning was the “Superloop”

```c
// 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
  }
}

```
void thread_blink() {
    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
    }
}

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); // wait for 100ms
    }
}

RTOS Multithreading: Multiple “Superloops”
```c
void thread_blink() {
    pinMode(LED_PIN, OUTPUT);
    while (1) {
        digitalWrite(LED_PIN, HIGH);
        RTOS_delay(1000);
        digitalWrite(LED_PIN, LOW);
        RTOS_delay(1000);
    }
}
```
Thread Blocking

- Thread makes a blocking call, e.g., RTOS_delay().
- RTOS kernel
- Thread-B blocked
- context switch
- Thread-B runs
- Thread-A blocked
- Clock tick interrupt
- ISR
- RTOS kernel
- context switch
- Thread-A runs
- Thread-B blocked
- Thread-A runs
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

- Shared-state concurrency
- Race conditions
- Mutual Exclusion
- Blocking
- Starvation
- Deadlock
- Priority inversion
- Missed deadlines
- Unresponsive threads
- More threads
- Architectural decay

Synchronization by blocking

Race conditions can lead to Mutual Exclusion, which in turn can lead to Starvation and Deadlock. Priority inversion can also lead to Missed deadlines, which can result in Failure. More threads can lead to Unresponsive threads, and architectural decay can occur.
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”
void thread_handler(AO_Type *ao) {
    // AO thread routine
    // setup
    while (1) {
        // event loop
        Event e = RTOS_queuePend(ao->queue); // pend on the event queue (BLOCKING!)
        ao->handle(e); // handle event (NON-BLOCKING!)
    }
}
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**.


- 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**

```c
void thread_handler(AO_Type *ao) {
  // AO thread routine
  ...
  // setup
  while (1) { // event loop
    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
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

![Diagram of a state machine with transitions and actions]
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

BSP

QS software tracing

QEP 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)
AO Frameworks vs. RTOS kernels

AO Frameworks can be **smaller** than RTOS kernels, because they don't need blocking.
AO Framework – “Software Bus”

- Active Object 1
- Active Object 2
- Active Object N
- ISR_1()
- ISR_2()

Direct event posting
Multicasting a published event
Publish-subscribe “software bus”
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
    case Q_ENTRY_SIG: /* entry action */
        /* entry action */
        status = Q.Handled();
        break;
    case Q_EXIT_SIG: /* exit action */
        /* 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)

- Find highest-priority non-empty queue
- "vanilla" scheduler
- All queues empty (idle condition)

priority = n
- e = queue.get();
- dispatch(e);

priority = n-1
- e = queue.get();
- dispatch(e);

priority = 1
- ...

priority = 0
- e = queue.get();
- dispatch(e);

idle processing
Preemptive, Non-Blocking Kernel (QK)

**Synchronous Preemption**

1. Low priority task
2. Time 0
3. High priority task
4. Time 15
5. Low priority task

**Asynchronous Preemption**

1. Low priority task
2. Time 0
3. Time 5
4. Time 7
5. Time 9
6. Time 11
7. Time 15
8. Time 17
9. Time 20
10. Time 22
11. Time 25

- Function call
- Interrupt entry/exit
- RTC scheduler
- Interrupt call
- Interrupt return
- Function call
- Interrupt entry/exit
- RTC scheduler
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 21\textsuperscript{st} century!
Demo: Blinky on Arduino
Demo: PELICAN on Arduino
Demo: Dining Philosophers with Q-SPY

QF

Philo[n]

(2)

thinking

TIMEOUT

hungry

HUNGRY(n)

TIMEOUT

eating

Philo[m]

(1)

thinking

(3)

hungry

EAT(m)

(5)

eating

Table

serving

(4)

(7)

DONE(m)

EAT(n)

(8)

(9)

TIMEOUT

Hungry

TIMEOUT

DONE(m)

EAT(n)

Hungry

TIMEOUT

Thought

EAT(n)

www.state-machine.com
Demo: “Fly 'n' Shoot” game on Windows

www.state-machine.com