Modern Embedded Systems Programming: Beyond the RTOS

Miro Samek
Quantum Leaps, LLC

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”

```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_alarm() {
    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() {
    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_blink() {
    pinMode(LED_PIN, OUTPUT);
    While (1) { // endless
        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()`
- Clock tick interrupt
- RTOS kernel
- Context switch

Timeline:
- Thread-A runs
- RTOS kernel
- Context switch
- Thread-B blocked
- Thread-A blocked
- Thread-A runs
- Thread-B runs
- Thread-B blocked
- Time
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
- Blocks
- Starvation
- Deadlock
- Priority inversion
- Missed deadlines
- Unresponsive threads
- More threads
- Architectural decay

Synchronization by blocking

Shared-state concurrency leads to race conditions, which may cause mutual exclusion. Blocking can lead to starvation, deadlock, priority inversion, and missed deadlines. This can result in unresponsive threads and more threads, leading to architectural decay.
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
        // pend on the event queue (BLOCKING!)
        Event e = RTOS_queuePend(ao->queue);
        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
        // 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
void thread_blink() {
    pinMode(LED_PIN, OUTPUT);
    while (1) {
        digitalWrite(LED_PIN, HIGH);
        RTOS_delay(1000); // NOT allowed!
        digitalWrite(LED_PIN, LOW);
        ...  
    }
}

Paradigm Shift: Sequential → Event-Driven

- No blocking
  → No use for most RTOS mechanisms!
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

(a)

```
\begin{align*}
E1 / \text{action1();} & \quad \rightarrow \text{s1} \\
E2 / \text{action2();} & \quad \rightarrow \text{s2} \\
E3 / \text{action3();} & \quad \rightarrow \text{s3}
\end{align*}
```

(b)

```
\begin{align*}
do X & \quad \rightarrow \text{do Y} \\
do Z & \quad \rightarrow \text{do W}
\end{align*}
```
Hierarchical State Machines

Traditional FSMs “explode” due to repetitions

State hierarchy eliminates repetitions → programming-by-difference
AO Frameworks for Deeply Embedded Systems

Active Object

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

multicasting a published event

direct event posting

ISR_1()

ISR_2()

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 implementation */
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG: /* exit action */
            /* exit action implementation */
            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)

- "vanilla" scheduler

- find highest-priority non-empty queue

- all queues empty (idle condition)

- idle processing

- priority = 0

- priority = 1

- priority = n-1

- priority = n

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

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

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

- . . .

- . . .

- . . .

- all queues empty (idle condition)

- find highest-priority non-empty queue
Preemptive, Non-Blocking Kernel (QK)

**Synchronous Preemption**

- Function call
- Interrupt entry/exit
- RTC scheduler

**Asynchronous Preemption**

- Function call
- Interrupt entry/exit
- RTC scheduler

Diagram showing the timeline of tasks and preemptive calls.
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
Demo: PELICAN on Arduino
Demo: Dining Philosophers with Q-SPY

QF

<table>
<thead>
<tr>
<th>Philo[n]</th>
<th>Philo[m]</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>thinking</td>
<td>thinking</td>
<td>serving</td>
</tr>
<tr>
<td></td>
<td>(2)</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>hungry</td>
</tr>
<tr>
<td></td>
<td>(4)</td>
<td>HUNGRY(m)</td>
</tr>
<tr>
<td></td>
<td>(5)</td>
<td>EAT(m)</td>
</tr>
<tr>
<td></td>
<td>(6)</td>
<td>hungry</td>
</tr>
<tr>
<td></td>
<td>(7)</td>
<td>HUNGRY(n)</td>
</tr>
<tr>
<td></td>
<td>(8)</td>
<td>DONE(m)</td>
</tr>
<tr>
<td></td>
<td>(9)</td>
<td>EAT(n)</td>
</tr>
<tr>
<td></td>
<td>(10)</td>
<td>thinking</td>
</tr>
</tbody>
</table>

TIMEOUT

TIMEOUT

TIMEOUT

TIMEOUT

Hungry

HUNGRY

EAT

DONE

EATING

HUNGRY

EAT

DONE

EATING
Demo: “Fly 'n' Shoot” game on Windows