Beyond the RTOS
Modern Embedded Software Architecture
Miro Samek, Ph.D.
Quantum Leaps, LLC
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”

// 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() { // RTOS thread routine
    pinMode(LED_PIN, OUTPUT); // setup: set the LED 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_alrm() { // 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
    }
}
void thread_blink() {
    pinMode(LED_PIN, OUTPUT);
    While (1) {
        // endless loop
        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

RTOS kernel

context switch

Thread-A runs

Thread-A blocked

Thread-A runs

Thread-B blocked

Thread-B runs

Thread-B blocked

priority
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

Synchronization by blocking

Shared-state concurrency

Race conditions

Mutual Exclusion

Blocking

Starvation

Deadlock

Priority inversion

Missed deadlines

Unresponsive threads

More threads

Architectural decay

Failure

Quantum Leaps
Modern Embedded Software

state-machine.com
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”
Best Practices: RTOS Implementation

Event queue
Private thread
start
Wait for event
Process event
Event queue
ISR

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!
        ...
    }
}

Paradigm Shift: Sequential → Event-Driven

- No blocking
  → No use for most RTOS mechanisms!

Sequential programming with RTOS

- Semaphores
- Mutexes
- Event Flags
- delay()
- Callback
- Timers

Paradigm Shift

Event-driven active object framework

- Threads
- Message Queues*
- Memory Pools
- Event Posting
- Publish/Subscribe
- Time Events
- State Machines

State-machine.com
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

```
default
ANY_KEY / send_lower_case_scan_code();

internal transitions

caps_locked
ANY_KEY / send_upper_case_scan_code();

trigger
list of 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”

(a)

```
+----+
| s1 |
+----+
    E1 / action1();
```

(b)

```
+----+
| s2 |
+----+
        do X

+----+
| s3 |
+----+
    E2 / action2();

+----+
| s4 |
+----+
    E3 / action3();
```

**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

```
+----+
| do Y |
+----+
    do Z
```

```
+----+
| do W |
+----+
```

Quantum® Leaps
Modern Embedded Software

state-machine.com
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)

**Source:** http://users.ece.cmu.edu/~koopman/lectures/ece642/04_modalstatechart.pdf
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

Direct event posting
Multicasting a published event

Publish-subscribe "software bus"

ISR_1() ISR_2()
QState Calc_on(Calc * const me, QEvt const *e) {
    QState status;
    switch (e->sig) {
        case Q_ENTRY_SIG: /* entry action */
            . . .
            status = Q_HANDLED();
            break;
        case Q_EXIT_SIG:  /* 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

priority = n-1

priority = 1

idle processing

e = queue.get();

dispatch(e);

e = queue.get();

dispatch(e);

... e = queue.get();

dispatch(e);
Preemptive, Non-Blocking Kernel (QK)

Synchronous Preemption:

- Priority: low priority task | high priority task
- Time: 0, 5, 10, 15, 20, 25
- Task Preempted: (1)
- Function Call: (3)
- Interrupt Entry/Exit: (2)
- RTC Scheduler: (4)
- Interrupt Call: (5)
- Interrupt Return: (6)

Asynchronous Preemption:

- Priority: low priority task | high priority task
- Time: 0, 5, 10, 15, 20, 25
- Task Preempted: (1)
- Function Call: (7)
- Interrupt Entry/Exit: (8)
- RTC Scheduler: (9)
- Interrupt Call: (10)
- Interrupt Return: (11)
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!**