Real-Time Software: Time as Important as Value

Implemented on Resource-Constrained Microcontrollers
Time modeled arithmetically

Time in seconds
Can add, subtract, multiply, and divide time intervals

0ms 50ms 100ms 150ms
Time modeled arithmetically

Time is quantized; quantum not user-visible

Quantum might be 1 MHz, 16 MHz, etc.

Integer timestamps thwart Zeno
Time modeled arithmetically

Time is quantized; quantum not user-visible

Program thinks processor is infinitely fast: execution a sequence of zero-time instants (hence “synchronous”)

Every instruction that runs in an instant sees the same timestamp

0ms  50ms  100ms  150ms
Time modeled arithmetically

Program thinks processor is infinitely fast: execution a sequence of zero-time instants (hence “synchronous”)

Time is quantized;
quantum not user-visible

Nothing happens in most instants (hence “sparse”)

---

0ms  50ms  100ms  150ms
```
blink led =
  loop
    after ms 50,
    led ← not (deref led)
  wait led

led = 0
```
blink led =
  loop
    after ms 50,
    led ← not (deref led)
  wait led

led = 0
blink led =

  loop
  after ms 50,
    led ← not (deref led)
  wait led

*led* is mutable; can be scheduled

Infinite loop
blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led

led is mutable; can be scheduled
Infinite loop
Schedule a future update

led = 0
blink led =
    loop
        after ms 50,
        led ← not (deref led)
    wait led

led is mutable; can be scheduled
Infinite loop
Schedule a future update

0ms  50ms  100ms  150ms

led ← 1

led = 0
blink led =
    loop
        after ms 50,
            led ← not (deref led)
    wait led

led is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

0ms

50ms

led ← 1

100ms

150ms

led = 0
blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led

*led* is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
blink led =
    loop
        after ms 50,
            led ← not (deref led)
        wait led

led is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

0ms  led ← 1
     50ms
     100ms
     150ms

led _______
`blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led`

*led* is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
led is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

```plaintext
blink led =
  loop
    after ms 50,
      led ← not (deref led)
    wait led
```

0ms

50ms

100ms

150ms

led
blink led =
loop
  after ms 50,
    led ← not (deref led)
  wait led

*led* is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

0ms

50ms

100ms

150ms

led ← 1
blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led

`led` is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
blink led =
  loop
    after ms 50,
    led ← not (deref led)
  wait led

*led* is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
blink led =
  loop
    after ms 50,
      led ← not (deref led)
    wait led

*led* is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led

*led* is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable
blink led =

loop

    after ms 50,
    led ← not (deref led)

wait led

*led* is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
blink led =
  loop
    after ms 50,
    led ← not (deref led)
  wait led

led is mutable; can be scheduled
Infinite loop
Schedule a future update
Wait for a write on a variable

0ms  50ms  led ← 0
      100ms

led
blink led =
  loop
    after ms 50,
      led ← not (deref led)
  wait led

_ led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable
blink led =

loop

after ms 50,

led ← not (deref led)

wait led

---

*led* is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable

---
`blink led =`

`loop`

`after ms 50,`

`led ← not (deref led)`

`wait led`

*LED is mutable; can be scheduled*

*Infinite loop*

*Schedule a future update*

*Wait for a write on a variable*
blink led =

\[
\text{loop}
\]

after ms 50,

\[
\text{led } \leftarrow \text{ not (deref led)}
\]

\[
\text{wait led}
\]
blink led =
  loop
    after ms 50,
    led ← not (deref led)
  wait led

*led* is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable

<table>
<thead>
<tr>
<th>0ms</th>
<th>50ms</th>
<th>100ms</th>
<th>150ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>led ← 1</td>
</tr>
</tbody>
</table>

led
Concurrent Code Executes in Syntactic Order for Determinism

add2 \( x \) = \( x \leftarrow \text{deref} \, x + 2 \)  // Add 2 as a side-effect

mult4 \( x \) = \( x \leftarrow \text{deref} \, x \times 4 \)  // Multiply by 4 as a side-effect
Concurrent Code Executes in Syntactic Order for Determinism

\[
\text{add2 } x = x \leftarrow \text{deref } x + 2 \quad \text{// Add 2 as a side-effect}
\]

\[
\text{mult4 } x = x \leftarrow \text{deref } x \times 4 \quad \text{// Multiply by 4 as a side-effect}
\]

main =
    \text{let } a = \text{new } 1 \quad \text{// Allocate a new mutable variable}
Concurrent Code Executes in Syntactic Order for Determinism

```plaintext
add2 x = x <- deref x + 2  // Add 2 as a side-effect

mult4 x = x <- deref x * 4  // Multiply by 4 as a side-effect

main =
  let a = new 1  // Allocate a new mutable variable
  par add2 a     // Runs first: a ← 1 + 2 = 3
      mult4 a    // Runs second: a ← 3 × 4 = 12
```

 Concurrent Code Executes in Syntactic Order for Determinism

\[\text{add2 } x = x \leftarrow \text{deref } x + 2 \quad \text{// Add 2 as a side-effect}\]

\[\text{mult4 } x = x \leftarrow \text{deref } x \times 4 \quad \text{// Multiply by 4 as a side-effect}\]

\text{main =}
\begin{align*}
\text{let} & \quad a = \text{new } 1 \quad \text{// Allocate a new mutable variable} \\
\text{par} & \quad \text{add2 } a \quad \text{// Runs first: } a \leftarrow 1 + 2 = 3 \\
& \quad \text{mult4 } a \quad \text{// Runs second: } a \leftarrow 3 \times 4 = 12 \\
\text{par} & \quad \text{mult4 } a \quad \text{// Runs third: } a \leftarrow 12 \times 4 = 48 \\
& \quad \text{add2 } a \quad \text{// Runs fourth: } a \leftarrow 48 + 2 = 50
\end{align*}
Concurrent Code May Block on \textit{wait}

\begin{verbatim}
blink led period =
  let timer = new ()                      // void/unit scheduled variable
  loop
    led <- not (deref led)                 // Toggle led now
    after period, timer <- ()             // Wait for the period
    wait timer

main led =
  par blink led (ms 50)
    blink led (ms 30)
    blink led (ms 20)                      // led toggles three times at time 600
\end{verbatim}
Basic trick: Two priority queues

First queue for scheduled variable update events

Second queue for code to be executed in the current instant

A wait statement reminds the variable that something is waiting on it

When a variable is written, it schedules the waiting code in the second queue
FDL 2020: C API for SSM Runtime

// Routine activation record management
rar_t *enter(size_t size, void (*step)(rar_t *), rar_t *caller,
             uint32_t priority, uint8_t depth)
void call(rar_t *rar)
void fork(rar_t *rar)
void leave(rar_t *rar, size_t size)

// Variable management
void initialize_type(cv_type_t *var, type val)            // new
void assign_type(cv_type_t *var, uint32_t priority, type val) // <-
void later_type(cv_type_t *var, uint64_t time, type val)   // after
bool event_on(cv_t *var)

// Trigger management (for wait statements)
void sensitize(cv_t *var, trigger_t *trigger)
void desensitize(trigger_t *trigger)
FDL 2020: C API Example

```c
rar_examp_t *enter_examp(rar_t *caller, uint32_t priority, unit8_t depth, cv_int_t *a) {
    rar_examp_t *rar = (rar_examp_t *)
    enter(sizeof(rar_examp_t), step_examp, caller, priority, depth);
    rar->a = a;             // Store pass-by-reference argument
    rar->trig1.rar = (rar_t *) rar; // Initialize our trigger
    return rar;
}

void step_examp(rar_t *gen_rar) {
    rar_examp_t *rar = (rar_examp_t *) gen_rar;
    switch (rar->pc) {
    case 0:
        initialize_int(&rar->loc, 0);         // let loc = new 0
        sensitize((cv_t *) rar->a, &rar->trig1); // wait a
        rar->pc = 1; return;
    case 1:
        if (event_on((cv_t *) rar->a)) {      // if @a then
            desensitize(&rar->trig1);          // De-register our trigger
            assign_int(&rar->loc, rar->priority, 42); // loc <- 42
            later_int(rar->a, now+10000, 43);    // after 10ms, a <- 43
            rar->pc = 2;
            call((rar_t *) enter_foo((rar_t *) rar, rar->priority, rar->depth, 42, &rar->loc));
            return;
        } else return;
    case 2:                                  // Concurrent call: par foo 40 loc; bar 42
        { uint8_t new_depth = rar->depth - 1;
            uint32_t pinc = 1 << new_depth;
            uint32_t new_priority = rar->priority;
            fork((rar_t *) enter_foo((rar_t *) rar, new_priority, new_depth, 40, &rar->loc));
            new_priority += pinc;
            fork((rar_t *) enter_bar((rar_t *) rar, new_priority, new_depth, 42));
            rar->pc = 3; return;
        }
    case 3: ; }
    leave((rar_t *) rar, sizeof(rar_examp_t)); // Terminate
}
```

```c
examp a =
    let loc = new 0
    wait a
    loc <- 42
    after ms 10, a <- 43
    par foo 42 loc
    par foo 40 loc
    bar 42
```
sigGen :: (?out0 :: Ref GPIO) => Ref Word64 -> SSM ()
    sigGen hperiod = routine $ while true (do
        after (ns (deref hperiod)) ?out0 (not' (deref ?out0))
        wait ?out0)

remoteControl :: (?ble :: BLE) => Ref Word64 -> SSM ()
    remoteControl hperiod = routine $ do
        enableScan ?ble
        while true (do
            wait (scanref ?ble)
            if deref (scanref ?ble) ==. 0
                then hperiod <-~ deref hperiod * 2
                else hperiod <-~ max' (deref hperiod /. 2) 1)

entry :: (?ble :: BLE, ?out0 :: Ref GPIO) => SSM ()
entry = routine $ do
    hperiod <- var (time2ns (secs 1))
MEMOCODE 2022: Timer and Interrupts Drive the Runtime

System tick driver

Input ISR
External Inputs

Input ISR

Input queue

Tick

loop

Semaphore

set alarm

Timer

External Inputs

Event queue

Alarm ISR

post

wait

post

set alarm

1:24 1:27 1:35

1:36 1:42 1:49

Tick loop

schedule

post

1:2 7 1:2 4 1:3 6

1:4 2 1:4 9
local ssm = require("ssm")

function ssm.pause(d)
    local t = ssm.Channel {}
    t:after(ssm.msec(d), { go = true })
    ssm.wait(t)
end

function ssm.fib(n)
    if n < 2 then
        ssm.pause(1)
        return n
    end
    local r1 = ssm.fib:spawn(n - 1)
    local r2 = ssm.fib:spawn(n - 2)
    local rp = ssm.pause:spawn(n)
    ssm.wait { r1, r2, rp }
    return r1[1] + r2[1]
end

local n = 10

ssm.start(
    function ()
        local v = ssm.fib(n)
        print(("fib(%d) => %d"):format(n, v))
        print("Completed in %.2fms"):format(ssm.as_msec(ssm.now()))
    end
)
MEMOCODE 2023: The RP2040

2 ARM Cortex M0+ processor cores, 133 MHz
264K SRAM
Off-chip QSPI flash (e.g., 2 MB)
30 GPIO pins
2 Programmable I/O Blocks (PIO)
US$1 quantity 1
MEMOCODE 2023: A PIO Block

4 “State Machines”
32-instruction memory (shared)
9 instructions (jump, wait, in, out, etc.)
4 32-bit registers
Single-cycle execution

From TX FIFO
Out Shift
Scratch X
To RX FIFO
In Shift
Scratch Y
To instruction memory
PC
Clock Div
From instruction memory (or bus)
Control Logic
IRQ Set, Clear, Status
To GPIO
From GPIO
MEMOCODE 2023: Ssslagn on an RP2040

**Latency:** 10-20 $\mu$s  
**Accuracy:** 62.5 ns / 16 MHz
sleep delay =
  let timer = new ()
  after delay, timer <- ()
  wait timer

waitfor var value =
  while deref var != value
    wait var

debounce delay input press =
  loop
    waitfor input 0
    press <- ()
    sleep delay
    waitfor input 1
    sleep delay

pulse period press output =
  loop
    wait press
    output <- 1
    after period, output <- 0
    wait output

buttonpulse button led =
  let press = new ()
  par debounce (ms 10) button press
    pulse (ms 200) press led

21 µs Button-to-LED latency
MEMOCODE 2023: 100 µs pulse: C vs Sslang Latency

C: 1.80µs reaction time
Sslang: 13.8us reaction time
C falling edge:
1.41 $\mu$s late, 960 ns jitter

Sslang falling edge:
0 $\mu$s late, 62.6 ns jitter (16 MHz clock)