REALTALK: Language Support for Long-Latency Operations in Embedded Devices

Alexandre Bergel
Hasso-Plattner-Institut

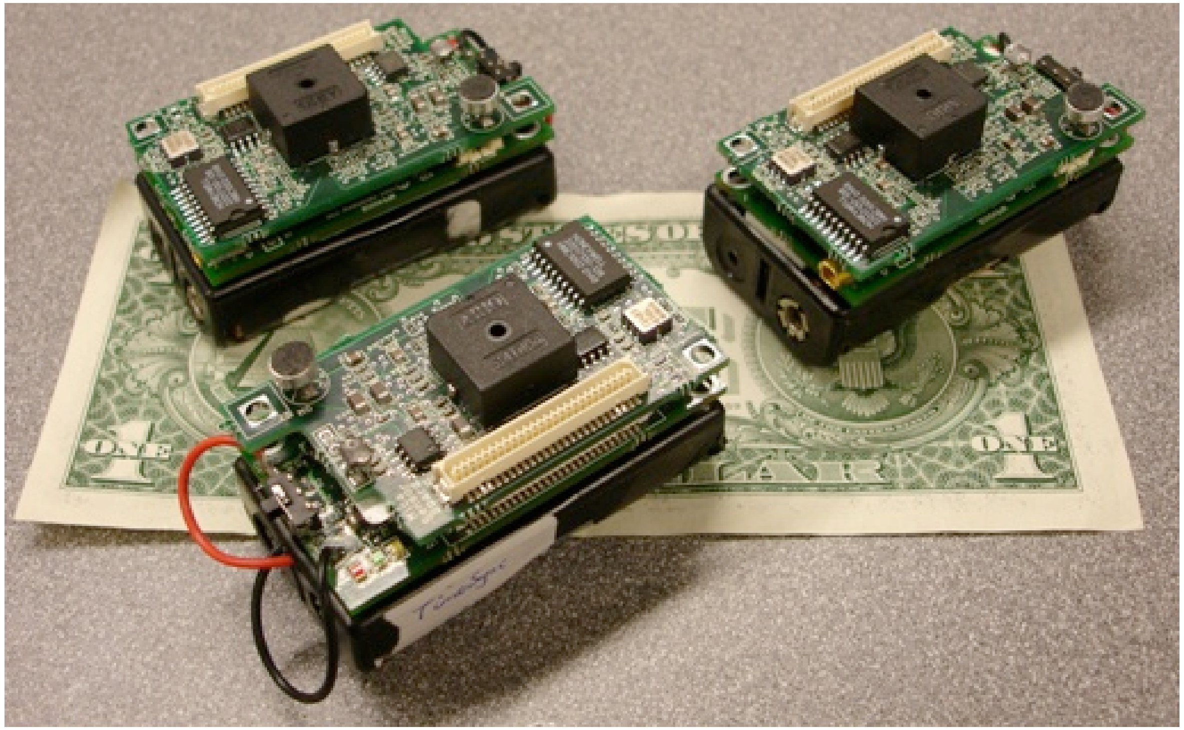
HPI Hasso Plattner Institut

Alexandre.Bergel@hpi.uni-potsdam

Outline

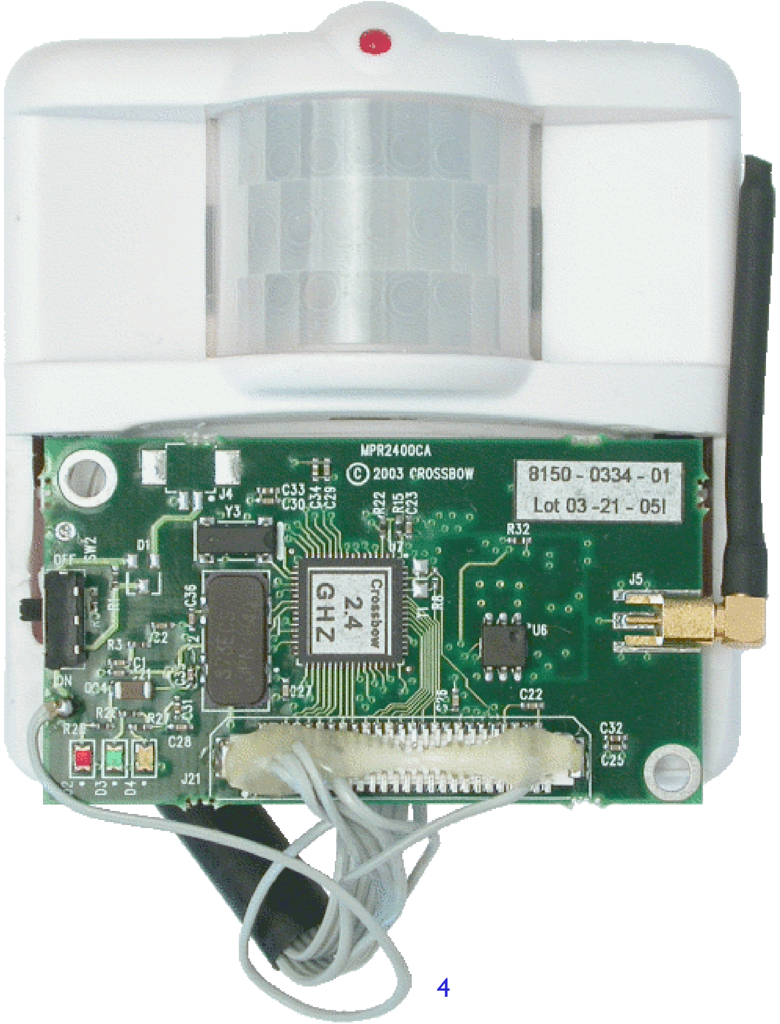
1. Wireless embedded sensor device
2. The Realtalk programming language
3. Benefits of Realtalk
4. Implementation based on TinyOS
5. Conclusion

Wireless sensor device



Hasso Plattner Institut

Wireless sensor device



Programming wireless sensor devices is difficult

- Limited resources (energy, memory)
- Asynchronism between electronic elements (sensors, micro-controller)

Programming wireless sensor devices is difficult

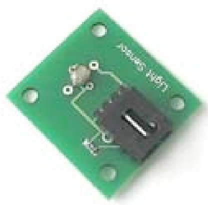
- Limited resources (energy, memory)

- Asynchronism between electronic elements (sensors, micro-controller)

Signal

Request

for a sampling



Program execution

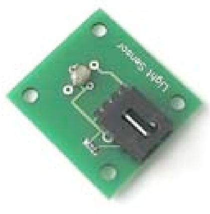
Hasso Plattner Institut

Programming wireless sensor devices is difficult

- Limited resources (energy, memory)

- Asynchronism between electronic elements (sensors, micro-controller)

Program execution



Environment being sampled

Hasso Plattner Institut

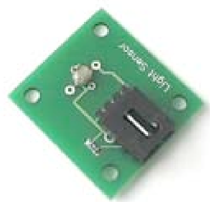
Programming wireless sensor devices is difficult

- Limited resources (energy, memory)
- Asynchronism between electronic elements (sensors, micro-controller)

Signal

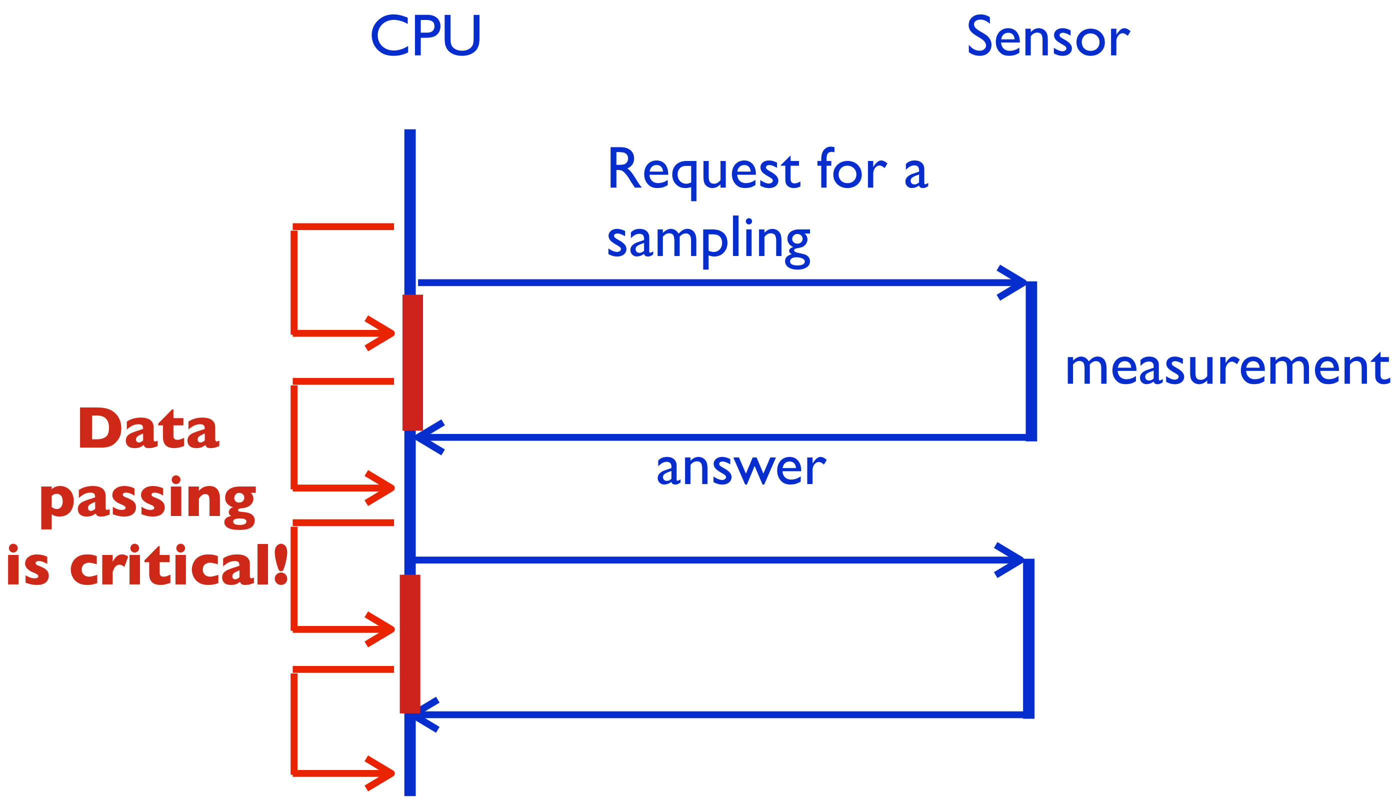
Sampling

transmitted



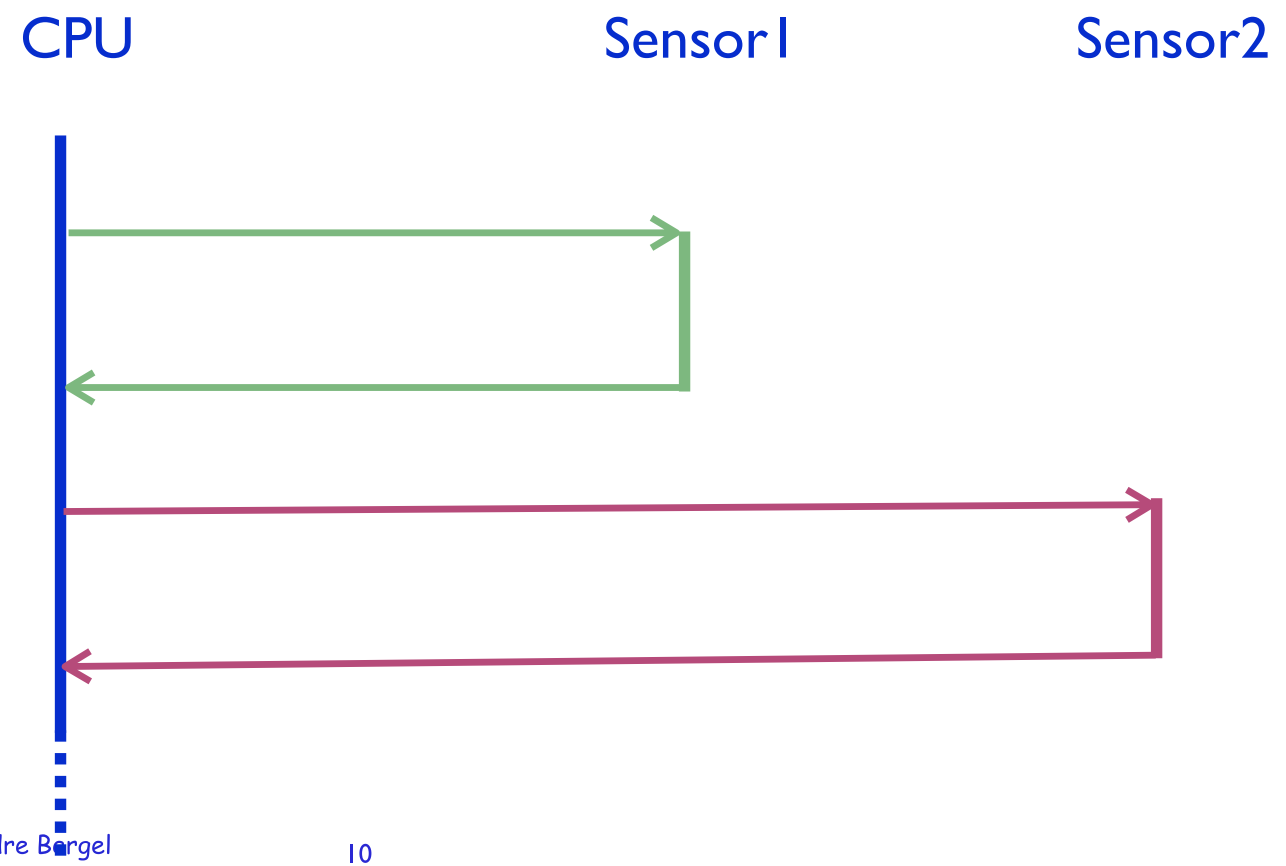
Program execution

Disruption in the control flow: data passing



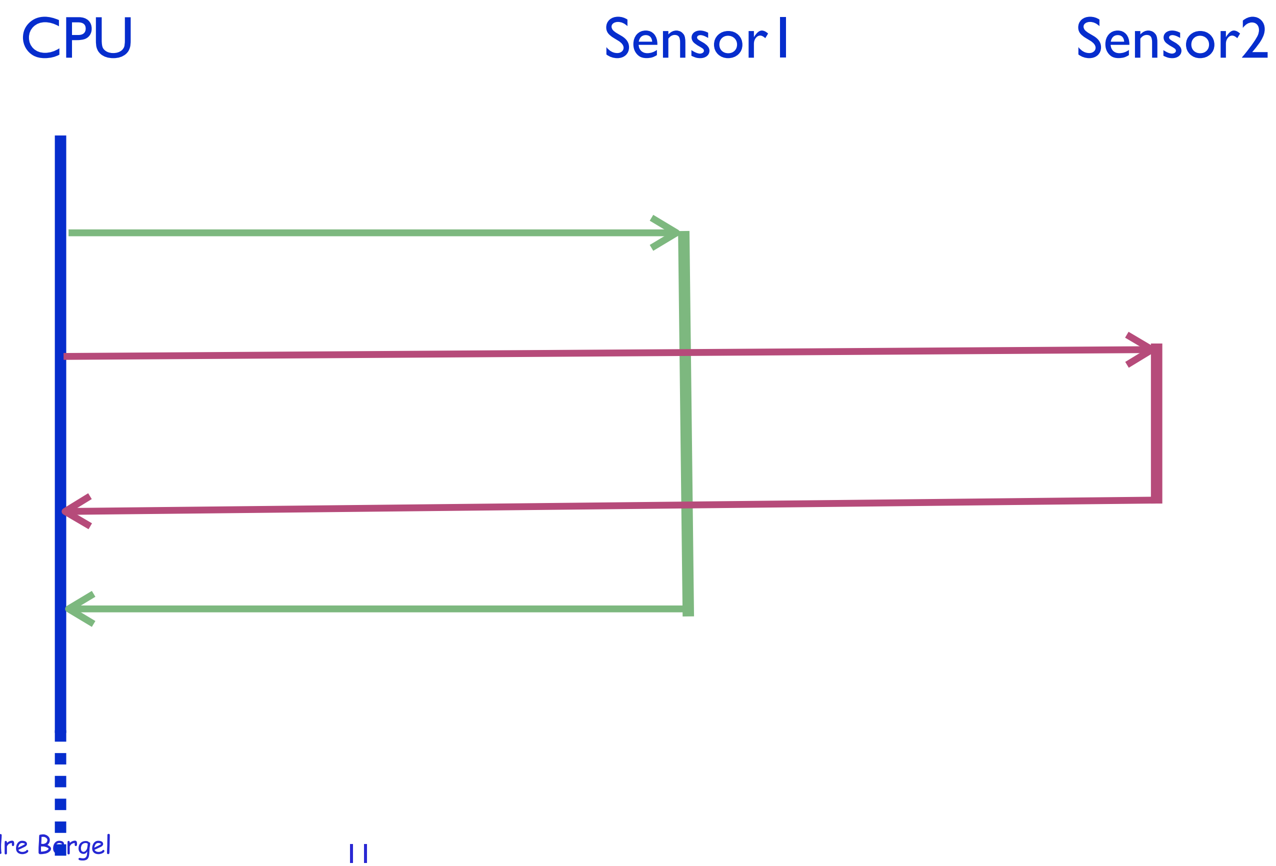
Disruption in the control flow: sequence ordering

What we would like to have...

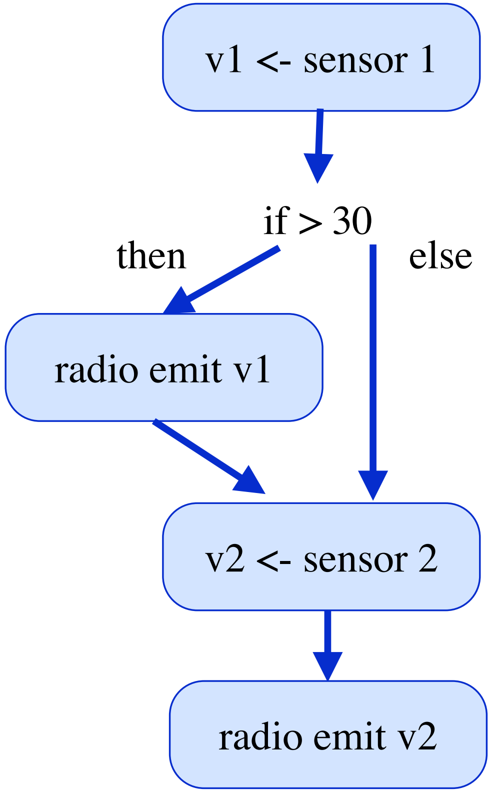


Disruption in the control flow: sequence ordering

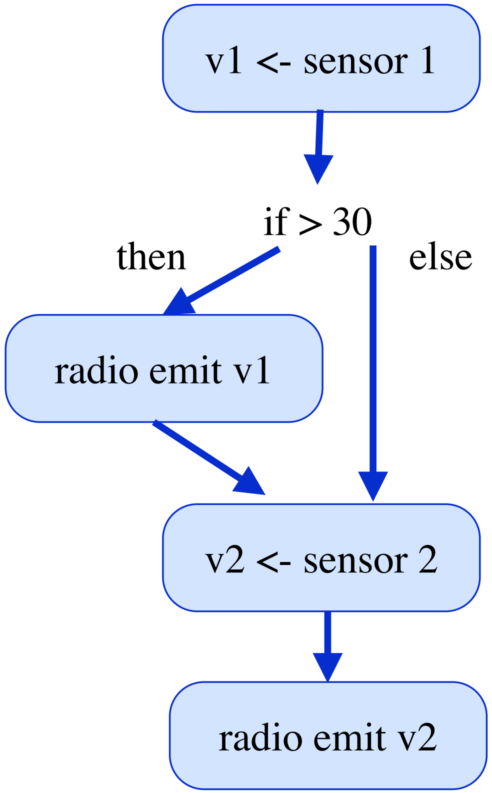
... but it is difficult to prevent this:



Example in nesC-like language



Example in nesC-like language




```
int controlFlowState = 0;
start {
  call Sensor1.getData();}

Sensor1.dataReady(v1) {
  if (v1 > 30)
    call Radio.emit(v1);
  call Sensor2.getData();}

Radio.emitDone() {
  if(controlFlowState == 0){
    controlFlowState = 1;
    call Sensor2.getData();}}

Sensor2.dataReady(v2) {
  controlFlowState = 1;
  call Radio.emit(v2);}
```

Ideas of Realtalk: controlled disruption

- Use of a light-weight continuation mechanism (kind of co-routine)
- *At compile time:* function body is cut into fragments
- *At runtime:* Fragments are inserted into a queue at runtime to be sequentially processed

Realtalk: modelling electronic elements as objects

```
RTObject subclass: #SensingApplication
  variableNames: 'sensor1 sensor2 radio timer'

SensingApplication main {
   timer invoke: #sample every: 500
}

SensingApplication sample {
  | v1 v2 |
  v1 := sensor1 read.
  v2 := sensor2 read.
  radio broadcast: (v1 + v2)
}
```

Realtalk: modelling electronic elements as objects

```
SensingApplication subclass: #BlinkingApplication
  variableNames: ‘leds’
  aliases: { #sample -> #readAndSend}.


BlinkingApplication sample {
  self readAndSend.
  leds greenToggle.
}
```

Classes are instantiated at compile-time only

```
"Deploying BlinkingApplication"

BlinkingApplication composeWith: {
   #leds    -> Leds .
   #timer   -> Timer .
   #sensor1 -> LightSensor .
   #sensor2 -> SoundSensor
}
```

Long latency operations are statically defined

```
LightSensor read {
  <AsynchronePrimitive>
  “mapping from LightSensor read to
Photo.getData(…)“
  ...
}
```


Hasso Plattner Institut

Controlled disruption: an overview

- At compile-time:
  - Use of long latency operations is localised
  - Methods are cut into an ordered set of small fragments

- At run-time:
  - When a long-latency operation has completed, the next fragment is processed
  - Method contexts hold dynamic information (i.e., local variables, …)

Example of controlled disruption

```
SensingApplication readingTwoSensors {
   | value1 value2 |

   leds display: 1.
   value1 := lightSensor read.

   leds display: 2.
   value2 := soundSensor read.

   leds display: 3.
   radio broadcast: (value1 + value2).
}
```

1 - Long latency operations are localised

```
SensingApplication readingTwoSensors {
   | value1 value2 |

   leds display: 1.
   value1 := lightSensor read.

   leds display: 2.
   value2 := soundSensor read.

   leds display: 3.
   radio broadcast: (value1 + value2).
}
```


Hasso Plattner Institut

2 - Methods are cut into fragments

```
SensingApplication readingTwoSensors {
   | value1 value2 |

   leds display: 1.
   value1 := lightSensor read.

   leds display: 2.
   value2 := soundSensor read.

   leds display: 3.
   radio broadcast: (value1 + value2).
}
```


Hasso Plattner Institut

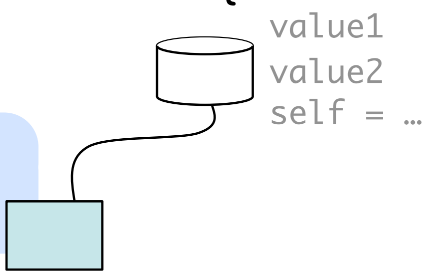
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



Hasso Plattner Institut

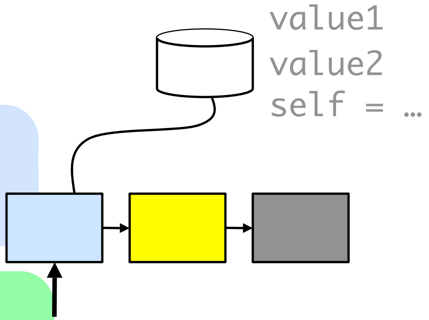
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



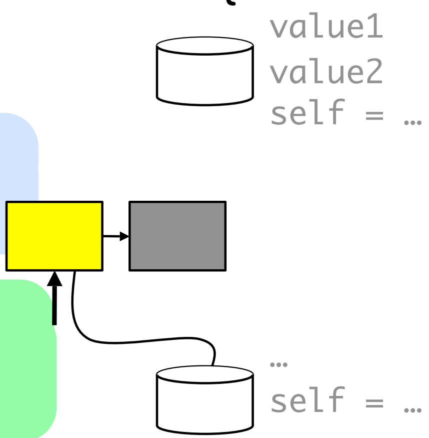
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



Hasso Plattner Institut

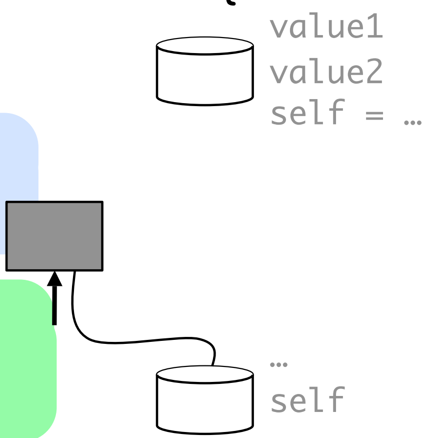
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

   leds display: 1.
   value1 := lightSensor read.

   leds display: 2.
   value2 := soundSensor read.

   leds display: 3.
   radio broadcast: (value1 + value2).
}
```



Hasso Plattner Institut

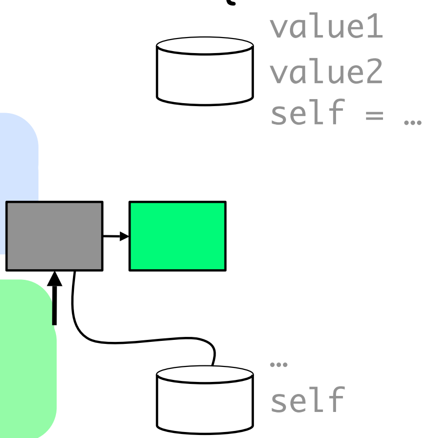
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



Hasso Plattner Institut

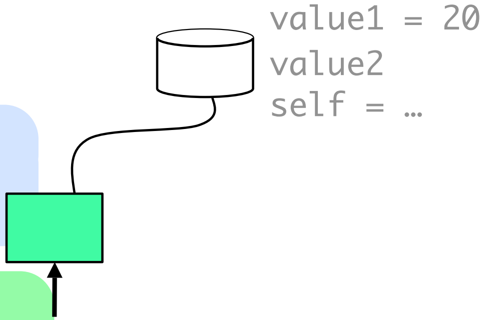
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



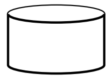
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



value1 = 20
value2
self = …

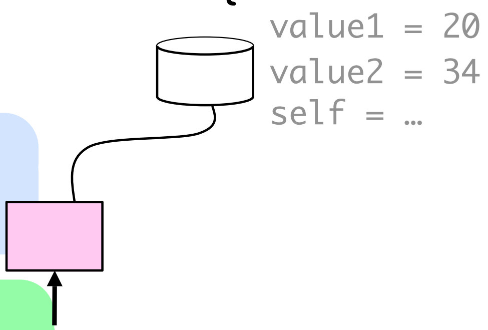
3 - Fragments are inserted into the queue

```
SensingApplication readingTwoSensors {
   | value1 value2 |

    leds display: 1.
    value1 := lightSensor read.

    leds display: 2.
    value2 := soundSensor read.

    leds display: 3.
    radio broadcast: (value1 + value2).
}
```



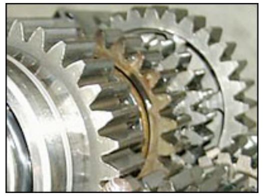
Hasso Plattner Institut

Benefits of controlled disruption

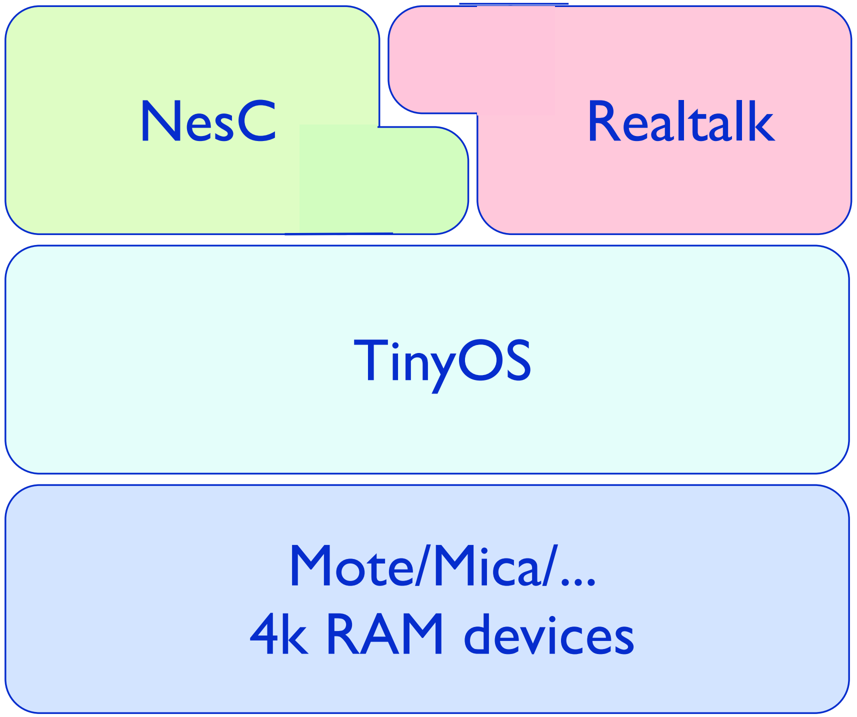
- Long-latency operations are blocking
- Data passing is achieved through the lexical scope of variables
- Sequence ordering reflects the application‘s control flow

Low memory overhead

Applications	Realtalk	nesC	ratio
CounterTo Leds	1532+46	1570+46	0.97
RadioToLeds	2382+81	2348+67	1.01
CounterToLeds AndRadio	9674+354	9398+384	1.02
SensorToRadio	10088+356	9618+386	1.04



Implementation based on TinyOS/NesC



- Compiler available: www.bergel.eu/realtalk.html

Hasso Plattner Institut

Conclusion

- Realtalk is a programming language dedicated to small sensor devices
- Control flow of program reflects the control flow of the application
- Generated programs do not have any overhead
- On going work focuses on the effect of polymorphism and method context generation regarding battery consumption.