Facilitating application development for wireless sensor networks

Atis Elsts

University of Latvia
and Institute of Electronics and Computer Science

Riga, Latvia
About me

- B.Sc. and M.Sc. from University of Latvia (2006 and 2008, respectively)
- PhD. thesis defended 13th December 2013
- Work experience:
  - Institute of Electronics and Computer Science (EDI)
    - Half-time work in science since 2010
    - Full-time work in science since 2011
  - Mikrotik (2005 - 2010)
    - Mikrotik is a Latvian company that makes routers
    - The routers are deployed worldwide (official distributors in 102 countries)
    - I worked on routing protocols (OSPF, BGP etc.) and other networking stuff
    - Adapted & implemented a mesh routing protocol (honorable mention in Latvian IT best student thesis contest in 2008)
The programming of WSN

- WSN software & hardware development is complex because of:
  - **Embedded** device complexity
  - **Networked** device complexity
  - **Energy** constraints
- It may be impossible to shield the developer from implications of low-level choices
- Still, significant classes of WSN **applications** are conceptually simple
  - Sense-and-send
  - Event-detection
  - Data query / reply

*Figure. Typical data flow in a sensor network node*
The users of WSN

- WSN was envisioned as a tool for domain experts, not computer scientists
- At the moment, WSN app programming for them is hard to impossible
- Financial incentives to empower them:
  - Salary for qualified engineers is the largest cost factor in real-world WSN deployments [Corke 2012]
- Scientific incentives:
  - Make the users contributors, not just observers!

*Figure. Our target audience: the grey area*
Our approach

SEAL – Sensor Application Development Language

- A domain-specific language (in contrast to a library or an embedded DSL)
- Minimalistic
- Declarative (in contrast to imperative)
- Extensible
- Bottom-up design

Figure. Conceptual architecture
Example application

Light monitoring application

```
read Light, period 10s;  # Reads light sensor
output Radio;            # Sends light sensor measurements to other nodes

read RemoteLight;        # Reads light sensors from direct neighborhood nodes

define MinLight min(Light, RemoteLight);  # lesser of...

when MinLight < 100lux:  # Whenever any sensor is below 100 lux threshold...
    use Led, on;          # Turn on light
end
```
Example application

Light monitoring application

```
read Light, period 10s;
output Radio;

read RemoteLight;
define MinLight
    min(Light, RemoteLight);
when MinLight < 100lux:
    use Led, on;
end
```
Example application

Light monitoring application

```plaintext
read Light, period 10s;  # Sensor use case
output Radio;            # Output use case

read RemoteLight;        # Sensor use case
define MinLight
    min(Light, RemoteLight);
when MinLight < 100lux:
    use Led, on;           # Actuator use case
end
```
Example application

Light monitoring application

```plaintext
read Light, period 10s;
output Radio;

read RemoteLight;
declare MinLight
    min(Light, RemoteLight);
when MinLight < 100lux:
    use Led, on;
end
```

Parameters
Example application

Light monitoring application

```plaintext
read Light, period 10 s;
output Radio;

read RemoteLight;
define MinLight
   min(Light, RemoteLight);
when MinLight < 100 lux:
   use Led, on;
end
```

Custom suffixes, expressed in familiar physical units
Example application

Light monitoring application

```plaintext
read Light, period 10s;  ← Executed periodically
output Radio;  ← Executed whenever all active local sensors have been read
read RemoteLight;  ← Executed on reception of net packets
define MinLight
  min(Light, RemoteLight);
when MinLight < 100lux:
  use Led, on;
end
```

Conditional expression is reevaluated whenever the parts may change their values

Executed one time whenever this code subbrach is activated
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Implementation in C code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Component use case</strong></td>
<td>Tells that the named component must be used. The usage is type-dependent and means that the component-specific use function from component library must be called.</td>
<td>Calls of component usage functions, scheduled with use of timers (possibly periodic).</td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>use LED, period 1s;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Light;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>output SDcard;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Elements of SEAL

<table>
<thead>
<tr>
<th>Element</th>
<th>Apraksts</th>
<th>Implementation in C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>“when” statement</td>
<td>Depending on a value of binary condition decides which subbranched of code should be activated (selected for periodic execution).</td>
<td>Event-trigged recalculation of binary expressions, depending on which some timers may be activated and others deactivated.</td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| when Light < 100lux:  
use LED, on;  
else:  
use LED, off;  
end |                                                                          |                                                                                            |
## Elements of SEAL

<table>
<thead>
<tr>
<th>Elements</th>
<th>Apraksts</th>
<th>Implementation in C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>“do” statement</td>
<td>Determines the order of SEAL code subbranches. May determine other</td>
<td>Timer “chaining”. Each subbranch corresponds to a single</td>
</tr>
<tr>
<td></td>
<td>execution parameters as well (such as the number of times, period,</td>
<td>timer.</td>
</tr>
<tr>
<td></td>
<td>delay, and duration).</td>
<td></td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do, once:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Light;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then, times 2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Humidity;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then, initialTimeout 10s:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read Temperature;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Elements of SEAL

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Implementation in C code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;define&quot; statement</td>
<td>Defines a “virtual” sensor</td>
<td>Function composition between sensor reading functions and predefined data processing functions.</td>
</tr>
<tr>
<td>Example:</td>
<td>define MaxTemp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max(Temperature);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read MaxTemp;</td>
<td></td>
</tr>
<tr>
<td>&quot;set&quot; statement</td>
<td>Initializes or changes a system state variable.</td>
<td>Scalar variable declaration and assignment, respectively.</td>
</tr>
<tr>
<td>Examples:</td>
<td>set isAlarm True;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add(count,1);</td>
<td></td>
</tr>
<tr>
<td>&quot;load&quot; statement</td>
<td>Imports a Python file (component library extension) or a C file (run-time library extension).</td>
<td>Extension of set of Python modules imported by SEAL compiler. Extension of set of C files compiled.</td>
</tr>
<tr>
<td>Examples:</td>
<td>load “extension.py”;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>load “ext_impl.c”;</td>
<td></td>
</tr>
</tbody>
</table>
SEAL-Blockly

- **Blockly** – visual programming language & tool from Google
- SEAL-Blockly – **Blockly** adaptation for SEAL
- **Pros:**
  - Familiar environment (web browser)
  - All possible blocks are predefined and visible in menus
  - Syntax errors are not possible
- **Cons:**
  - Visual programs can be harder to read (Green, 1992)
  - In some contexts textual programming is more appropriate (Pane & Myers, 1996)
SEAL-Blockly compared to SEAL, and compared to programming in C

**Code comparison**

<table>
<thead>
<tr>
<th>SEAL-Blockly:</th>
<th>SEAL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>read Light;</td>
<td>Read Light;</td>
</tr>
<tr>
<td>when Light &lt; 50:</td>
<td>When Light &lt; 50:</td>
</tr>
<tr>
<td>Use RedLed, On;</td>
<td>Use RedLed, On;</td>
</tr>
<tr>
<td>Else:</td>
<td>Else:</td>
</tr>
<tr>
<td>Use RedLed, Off;</td>
<td>Use RedLed, Off;</td>
</tr>
<tr>
<td>End</td>
<td>End</td>
</tr>
<tr>
<td>Output Radio;</td>
<td>Output Radio;</td>
</tr>
</tbody>
</table>

**C code:**

```c
void appMain(void)
{
    alarmInit(&redledBranchAlarm, redledBranch1Callback, NULL);
    alarmInit(&redledBranch2Alarm, redledBranch2Callback, NULL);
    sealNetRegisterInterest5(5, condition1Callback);
    bool newBranchStatus;
    branch0Start();
    newBranchStatus = branch1Evaluate();
    if (newBranchStatus) {
        ...
        and so on
```
Getting practical

How does SEAL handles common WSN programming idioms?
• Processing (e.g. averaging) data before sending them out
• Waiting on an event (polling or interrupt-based)
• Handling interactive queries
• Implementing state machine logic
• Tweaking system-level stuff for application needs
• Performing actions in strict order & with precise timing
Getting practical

How does SEAL handle common WSN programming idioms?

- Averaging data before sending them out
  
  \begin{verbatim}
  read avg(take(Light, 10));
  \end{verbatim}

- Waiting on an event (interrupt-based)
  
  \begin{verbatim}
  read InputPin, interruptPort 1, interruptPin 2;
  when InputPin = 1: ...
  \end{verbatim}

- Handling interactive queries
  
  \begin{verbatim}
  const COMMAND 1234;
  read RemoteCommand; read RemoteAddress;
  when RemoteCommand = COMMAND
      and RemoteAddress = BASE_STATION: ...
  \end{verbatim}
How does SEAL handle common WSN programming idioms?

• Implementing state-machine logic

```plaintext
set AlarmState False;
read Temperature;
when Temperature > 40C: set AlarmState True;
end
```

• Tweaking system-level stuff for application needs

```plaintext
output Network, protocol "TDMA", routing "CTP", radioChannel 11, radioTxPower 10dBm;
```

• Performing actions in strict order & with precise timing

```plaintext
do: use LED, on;
then, once: read Humidity;
then, initialTimeout 1s: use LED, off;
end
```
Language complexity of SEAL

- Smaller grammar & fewer keywords: less time required to learn the language completely
- Fewer special-purpose symbols and operators: code is more intuitive to read

**Table. Language complexity comparison**

<table>
<thead>
<tr>
<th></th>
<th>SEAL</th>
<th>TinyScript</th>
<th>TinyDB</th>
<th>snLog</th>
<th>C</th>
<th>nesC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonterminal symbols</td>
<td>36</td>
<td>31</td>
<td>26</td>
<td>21</td>
<td>70</td>
<td>24 new + 13 changed</td>
</tr>
<tr>
<td>Keywords¹</td>
<td>19</td>
<td>15</td>
<td>20</td>
<td>10</td>
<td>31</td>
<td>15 new</td>
</tr>
<tr>
<td>Operators</td>
<td>8</td>
<td>23</td>
<td>12</td>
<td>16</td>
<td>46</td>
<td>1 new</td>
</tr>
<tr>
<td>Non-alphanumeric characters</td>
<td>14</td>
<td>19</td>
<td>14</td>
<td>13</td>
<td>30</td>
<td>0 new</td>
</tr>
<tr>
<td>with special meaning</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹ – not including aggregation etc. functions
Application complexity of SEAL

- Metrics: line count, effort according to Halstead's metrics, cyclomatic complexity
- Cyclomatic complexity – a metric for the number of branches in the execution flow
- Higher cyclomatic complexity suggests that code is harder to understand

**Table. Cyclomatic complexity comparison for 8 applications in 5 implementations**

<table>
<thead>
<tr>
<th></th>
<th>SEAL</th>
<th>BASIC</th>
<th>TinyScript</th>
<th>C</th>
<th>nesC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sense &amp; print</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Sense &amp; send</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>SAD</td>
<td>2</td>
<td>n/a</td>
<td>n/a</td>
<td>4</td>
<td>n/a</td>
</tr>
<tr>
<td>Exercise 1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Exercise 2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Exercise 3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Exercise 4</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
SEAL resource usage overhead

**Figure.** SEAL, C, and nesC application binary code usage
SEAL applications

SEAL applications (not in system level, except the first):

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Feature demonstrated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measuring microclimate [Elsts 2012a]</td>
<td>Periodic sampling</td>
</tr>
<tr>
<td>Controlling a watering system [Elsts 2012a]</td>
<td>Distributed processing</td>
</tr>
<tr>
<td>Collecting light measurements [Selavo 2007]</td>
<td>Interactive query support</td>
</tr>
<tr>
<td>Detecting seismic activity [Werner-Allen 2006]</td>
<td>Event-driven processing</td>
</tr>
<tr>
<td>Light monitoring in tunnels [Ceriotti 2011]</td>
<td>Intra-node proc. (using C)</td>
</tr>
<tr>
<td>Building a backup controller [Zviedris 2010]</td>
<td>Imperative elements</td>
</tr>
<tr>
<td>Analog signal processing [Strazdins 2011]</td>
<td>Intra-node processing</td>
</tr>
<tr>
<td>Generating a signal [Mednis 2012c]</td>
<td>Sequential execution</td>
</tr>
<tr>
<td>Detecting potholes #1 [Mednis 2011]</td>
<td>Intra-node processing</td>
</tr>
<tr>
<td>Detecting potholes #2 [Mednis 2011]</td>
<td>Intra-node processing</td>
</tr>
<tr>
<td>Detecting vehicle movement [Mednis 2012a]</td>
<td>Optimized intra-node processing</td>
</tr>
<tr>
<td>Finding parking space [Kothari 2007]</td>
<td>Distributed processing</td>
</tr>
</tbody>
</table>
Application: environmental monitoring

Environmental monitoring in precision agriculture
Distributed microclimate measurements in an orchard:
  • temperature, humidity, light

The potential users are agroscientists in Latvia
State Institute of Fruit-Growing (http://lvai.lv)

17 node network in 2011, 19 node sensor network in 2012
>50 000 sensor measurements collected in 2011
>200 000 sensor measurements collected in 2012

In 2012: application logic of the whole network:
described with 14 lines of SEAL code!
Conclusion

• A new WSN programming language SEAL proposed
  • Aimed directly at domain experts
  • Motivated by real-world applications
  • Extensible
  • Implementation is feasible on low-power MCU
  • A front-end for visual programming implemented (SEAL-Blockly)

• Future work:
  • More of macroprogramming
  • Formal methods (e.g. for test set generation)
  • Support of other platforms (e.g. TinyOS, Arduino)
Thank you!