Using C Language Extensions for Developing Embedded Software - A Case Study

Markus Völter voelter@acm.org
Arie van Deursen Arie.vanDeursen@tudelft.nl
Stephan Eberle stephan.eberle@itemis.com
Bernd Kolb Bernd.kolb@itemis.de
(How well) do domain-specific language extensions work?

And how can we find out?
Domain-Specific Extensions of C for Embedded Software

An Industrial Case Study
An Industrial Case Study

Smart Meter

Measures Voltage and Current
Computes Derived Values
Shows Data on LCD Display
Communicates through Networks

Precision is critical for Certification.
Evolvability is critical for it to be a viable business.

Developed with mbed, a set of domain-specific extensions to C, plus an IDE.
An extensible set of integrated languages for embedded software engineering.

<table>
<thead>
<tr>
<th>User Extensions</th>
<th>Test Support</th>
<th>Decision Tables</th>
<th>Logging &amp; Tracing</th>
<th>to be defined by users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Extensions</td>
<td>Components</td>
<td>Physical Units</td>
<td>State Machines</td>
<td>State Machine Verification</td>
</tr>
<tr>
<td>Core</td>
<td>C99</td>
<td>Model Checking</td>
<td>SMT Solving</td>
<td>Dataflow Analysis</td>
</tr>
<tr>
<td>Platform</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backend Tool</td>
<td>C Compiler, Debugger and Importer</td>
<td>NuSMV</td>
<td>Yices</td>
<td>CBMC</td>
</tr>
</tbody>
</table>

Implementation Concern | Analysis Concern | Process Concern
Setup
RealisQc

Real requirements, real size, real deadlines, representative developers

Reproducible

Not so easy to reproduce, because the source code of Smart Meter is not available. mbeddr itself is open source, though:

http://mbeddr.com/
Research Questions

**Complexity**
Are the abstractions provided by mbeddr beneficial for mastering the complexity encountered in a real-world embedded system? Which additional abstractions would be needed or useful?

**Testing**
Can the mbeddr extensions help with testing the system? In particular, is hardware-independent testing possible to support automated, continuous integration and build? Is incremental integration and commissioning supported?

**Overhead**
Is the low-level C code generated from the mbeddr extensions efficient enough for it to be deployable onto a real-world embedded device?

**Effort**
How much effort is required for developing embedded software with mbeddr?
### Data Collected

<table>
<thead>
<tr>
<th>complexity</th>
<th>Qualitative impact of mbeddr and SM extensions on complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>testing</td>
<td>Measured Coverage</td>
</tr>
<tr>
<td></td>
<td>Test-Specific SMT Code</td>
</tr>
<tr>
<td></td>
<td>Commissioning of the system</td>
</tr>
<tr>
<td>overhead</td>
<td>Compared Size of Binary with Resources of Hardware</td>
</tr>
<tr>
<td></td>
<td>Analyzed/Measured Performance</td>
</tr>
<tr>
<td></td>
<td>Theoretical Discussion of the Overhead of Extensions</td>
</tr>
<tr>
<td>effort</td>
<td>Report and discuss Effort required to build SM separated by implementation, testing, commissioning and extension development</td>
</tr>
</tbody>
</table>
Hardware Architecture

Application Logic

**MSP430 F67791**
- 25 MHz
- 256K Flash ROM
- 32K RAM

Metrology

**MSP430 F6736**
- 25 MHz
- 128K Flash ROM
- 8K RAM

- RS485
- IrDA
- DLMS/COSEM

MQTT ↔ UART
Software Architecture

No RTOS
Interrupt-Driven
One-Threaded Programming

Required Precision leads to 4096 Hz Sampling Rate

Interrupt-Triggered:
Measurement

Foreground Tasks:
App Logic, RTC
Example Smart Meter Code

From the processor vendor. But: no tests, bad structure, buggy, not all features.

Hence:

Phase 1  Reimplement with mbeddr

Phase 2  Two Processors,
         Communication between the two processors,
         Improved comms infrastructure (multiplexing,
         two comm stacks RS485 and IrDa)
         an I2C Bus driver
         an EEPROM controller
         a subset of the required DLMS/COSEM messages
         additional application functionality (historical data rec, reset)
## Size of the System

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Common</th>
<th>Metro</th>
<th>App</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Files</td>
<td>134</td>
<td>101</td>
<td>105</td>
<td>340</td>
</tr>
<tr>
<td>Total LOC</td>
<td>8,209</td>
<td>10,447</td>
<td>10,908</td>
<td>29,564</td>
</tr>
<tr>
<td>Code LOC</td>
<td>4,397</td>
<td>5,900</td>
<td>5,510</td>
<td>15,807</td>
</tr>
<tr>
<td>Comment LOC</td>
<td>950</td>
<td>2,402</td>
<td>2,620</td>
<td>5,972</td>
</tr>
<tr>
<td>Whitespace LOC</td>
<td>2,852</td>
<td>2,145</td>
<td>2,778</td>
<td>7,775</td>
</tr>
</tbody>
</table>

**Common** code runs on both processors, **Metro** runs on the metrology processor and **App** runs on the application / communication processor.

+ roughly the same amount again for tests.
### Use of Extensions

<table>
<thead>
<tr>
<th>Category</th>
<th>Concept</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunks (≈ Files)</td>
<td>Implementation Modules</td>
<td>382</td>
</tr>
<tr>
<td></td>
<td>Other (Req, Units, etc.)</td>
<td>46</td>
</tr>
<tr>
<td>C Constructs</td>
<td>Functions</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>Structs / Members</td>
<td>144 / 270</td>
</tr>
<tr>
<td></td>
<td>Enums / Literals</td>
<td>150 / 1,211</td>
</tr>
<tr>
<td></td>
<td>Global Variables</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>Constants</td>
<td>8,500</td>
</tr>
<tr>
<td>Components</td>
<td>Interfaces / Operations</td>
<td>80 / 197</td>
</tr>
<tr>
<td></td>
<td>Atomic Components</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>Ports / Runnables</td>
<td>630 / 640</td>
</tr>
<tr>
<td></td>
<td>Parameters / Values</td>
<td>84 / 324</td>
</tr>
<tr>
<td></td>
<td>Composite Components</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Component Config Code</td>
<td>1,222</td>
</tr>
<tr>
<td>State Machines</td>
<td>Machines</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>States/Transitions/Actions</td>
<td>14 / 17 / 23</td>
</tr>
<tr>
<td>Physical Units</td>
<td>Unit Declarations</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>Conversion Rules</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Types / Literals with Units</td>
<td>593 / 1,294</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Concept</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Line</td>
<td>Feature Models / Features</td>
<td>4 / 18</td>
</tr>
<tr>
<td>Variability</td>
<td>Configuration Models</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Presence Condition</td>
<td>117</td>
</tr>
<tr>
<td>Custom Extensions</td>
<td>Register Definition</td>
<td>387</td>
</tr>
<tr>
<td></td>
<td>Interrupt Definitions</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Protocol Messages</td>
<td>42</td>
</tr>
<tr>
<td>Statements</td>
<td>Statements total</td>
<td>16,840</td>
</tr>
<tr>
<td></td>
<td>Statements in components</td>
<td>6,812</td>
</tr>
<tr>
<td></td>
<td>Statements in test cases</td>
<td>5,802</td>
</tr>
<tr>
<td></td>
<td>Statements in functions</td>
<td>3,636</td>
</tr>
<tr>
<td>Testing</td>
<td>Test Cases / Suites</td>
<td>107 / 35</td>
</tr>
<tr>
<td></td>
<td>Test-Specific Components</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Stub / Mock Components</td>
<td>9 / 8</td>
</tr>
<tr>
<td></td>
<td>assert Statements</td>
<td>2,408</td>
</tr>
</tbody>
</table>

All mbeddr C extensions used a lot.
Some extensions built specifically for SM.
The Code
Components (mbeddr)

```cpp
// ADC is the analog-digital converter
interface IADC {
  int16 read(uint8 addr)
}

component ADCDriver {
  provides IADC adc
  int16 adc_read(uint8 addr) <= op adc.read {
    int16 val = // low level code to read from addr
    return val;
  } }

component CurrentMeasurer {
  requires IADC currentADC
  internal void measureCurrent() {
    int16 current = currentADC.read(CURR_SENSOR_ADDR);
    // do something with the measured current value
  } }
```
State Machines (mbeddr)

```plaintext
statemachine FrameParser initial = idle {
  var uint8 idx = 0
  in event dataReceived(uint8 data)
    state idle {
      entry { idx = 0; }
      on dataReceived [data == LEADING_BYTE] -> wakeup
    }
    state wakeup {
      on dataReceived [data == START_FLAG]
        -> receivingFrame { idx++; }
    }
    state receivingFrame { .. }
}

// create and initialize state machine
FrameParser parser;
parser.init;
// trigger dataReceived event for each byte
for (int i=0; i<data_size; i++) {
  parser.trigger(dataReceived|data[i]);
}
```
```c

testcase testFrameParser1 {  
    FrameParser p;
    assert(0) p.isInState(idle);
    // invalid byte; stay in idle  
    parser.trigger(dataReceived|42);
    assert(0) p.isInState(idle);
    // LEADING_BYTE, go to awakening  
    parser.trigger(dataReceived|LEADING_BYTE);
    assert(0) p.isInState(awakening);
}

testcase testFrameParser2 { ... }  
testcase testFrameParser3 { ... }  

int32 main(int32 argc, char* argv) {  
    return test[testFrameParser1,  
    testFrameParser2,  
    testFrameParser3];
```
Mocks & Units (mbeddr)

```c
mock component USCIReceiveHandlerMock {
    provides ISerialReceiveHandler handler
    Handle* hnd;

    sequence {
        step 0: handler.open { } do { hnd = handle; }
        step 0: handler.dataReceived {
            assert 0: parameter data: data == 1 }
        step 1: handler.dataReceived {
            assert 1: parameter data: data == 2 }
        step 2: handler.dataReceived { .. }
        step 3: handler.dataReceived { .. }
        step 4: handler.finsihed { } do { close(hnd); }
    }
}
```

```c
unit V := for voltage
unit A := for Amps
unit Ω := V · A⁻¹ for resistance

uint16/Ω/ resistance(uint16/V/ u, uint16/A/[] i, uint8 ilen) {
    ilen
    uint16/A/ avg_i = \sum_{p = 0}^{ilen} i[p];

    return \frac{avg_i}{u}; Error: type uint16 /V⁻¹· A/ is not a subtype of uint16 /Ω/
}
```

```c
} resistance (function)
```
Product Lines (mbeddr)

```plaintext
feature model SMTFeatures
  root opt
  Data_LEDS opt
    DataReadLED
    DataWriteLED [DigitalIOPortPin pin]
  DISPLAY xor
    DISPLAY_V10
    DISPLAY_V22
    WRITABLE_FLASH_MEMORIES

exported composite component MetrologyPlatformLayer {
  provides IWatchdogTimer watchdogTimer
    ? {DataReadLED && WRITABLE_FLASH_MEMORIES}
  provides IDigitalOutputPin pin1
    ? {DataWriteLED}
  provides IDigitalOutputPin pin2
```
Registors (smart meter)

```c
exported register8 ADC10CTL0 compute as val * 1000

void calculateAndStore( int8 value ) {
    int8 result = // some calculation with value
    ADC10CTL0 = result; // stores result * 1000 in reg.
}
```
Interrupts (smart meter)

```plaintext
module USCIProcessor {
    exported interrupt USCI_A1
    exported interrupt RTC

    exported component RTCImpl {
        void interruptHandler() <- interrupt {
            hw->pRTCPCTL &~ RT1PSIFG;
        }
    }

    instances usciSubsystem {
        instance RTCImpl rtc;
        bind RTC -> rtc.interruptHandler
        connect ... // ports
    }
}
```
Messages (smart meter)

```c
// a field representing a timestamp for 10:20:00
uint8[6] f_time = \{0x00A, // field type identifier
                  UNIT_TIME24, // unit used: time
                  3, // 3 payload bytes follow
                  10, 20, 00 // the time itself
            \};

// a field representing a measured value
uint8[4] f_value = ...;

message CurrentMeasuredValue:42 {
    int32 timestamp; // time of measurement
    uint16/A/ value;  // measured value in Amps
    uint16 accuracy; // accuracy in 1/100 %
}
message ... { ... }
...

// a message that uses the two fields
uint8[5] message = ...

atomic component CoreMeasurer {
    field uint16/A/ lastValue = 0;
    message data 42 { :currentTime, &lastValue, 100};
    void measure() {
        lastValue = // perform actual measurement
    }
}
Answers to RQs
The developers *naturally think* in terms of extensions, and suggested additional ones during the project.

*mbeddr components help structure* the overall architecture and enable reuse and configurability.

*mbeddr extensions facilitate strong* static checking, improve *readability* and help avoid low-level mistakes.
RQ Testing

mbeddr components are instrumental in improving testability through clear interfaces and small units, leading to 80% test coverage for core components.

The custom extensions and the components facilitate hardware-independent testing, continuous integration and automated dry runs of the certification process.

The modularization facilitated by components helps track down problems during commissioning.
The memory requirements of SMT are low enough for it to run on the intended hardware, with room for growth.

Componentization enables deployment of only the functionality necessary for a variant, conserving resources.

The performance overhead is low enough to achieve the required 4,096 Hz sample rate on the given hardware.
## RQ Effort

<table>
<thead>
<tr>
<th>Development Tasks</th>
<th>Effort</th>
<th>% Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>200 PD</td>
<td>66%</td>
</tr>
<tr>
<td>Reimplementation</td>
<td>145 PD</td>
<td>48%</td>
</tr>
<tr>
<td>Additional Functionality</td>
<td>55 PD</td>
<td>18%</td>
</tr>
<tr>
<td>Tests, Simulators</td>
<td>48 PD</td>
<td>16%</td>
</tr>
<tr>
<td>Integration &amp; Commissioning</td>
<td>38 PD</td>
<td>13%</td>
</tr>
<tr>
<td>Custom Language Extensions</td>
<td>14 PD</td>
<td>5%</td>
</tr>
</tbody>
</table>
The effort for the additional functionality, integration and commissioning is lower than what is common in embedded software.

The effort for building the extensions is low enough for it to be absorbed in a real project.

Overall, using mbeddr does not lead to significant effort overrun, while resulting in better-structured software.
Discussion
Validity

Internal  Bias, Team Expertise
          Example Smart Meter Code

Conclusion Design of mbeddr
            Cognitive Dimensions of N.
            Concepts vs. Language
            Language vs. Tool

External  Beyond SM
          Beyond the Team
          Beyond the mbeddr Extensions
          Beyond mbeddr’s MPS Implementation
Discussion

**Debugging** on the DSL Level
an on the generated level

**Code Quality**  
Readable to build Trust
Readable for Debugging
MISRA Compliant: 25% automatic

**Maintainability**  
No long term experience
But good indications:
additional functionality
Drawbacks and Challenges

Limited Generator optimizations
same execution paradigm, not a problem yet.

2.5 X Longer Build Times

Tool Lock in: no way without MPS

Diff/Merge in MPS only

Learning Curve

Language Engineering Skills to build new L
Other Approaches
How is it different from...
Model-Driven-∗

Fully open and extensible
Multiple paradigms, not one-size-fits-all
Mix of „Model and Code“
How is it different from...

Macros

More syntactic flexibility
Higher Expressivity (do more than with Macros)
Type Checking
Generally better IDE support
How is it different from...

C++

Requires no C++ Compiler
Components more suitable for Embedded
Different Features: units, state machines

TMP:  Better IDE support
      Better Error Messages
      LE better done in LWB
Conclusions
The extensions help master complexity and lead to software that is more testable, easier to integrate and commission and is more evolvable.

Specific: mbeddr & Smart Meter
Despite the abstractions introduced by mbeddr, the additional overhead is very low and acceptable in practice.
The development effort is reduced, particularly regarding evolution and commissioning.

Specific: mbeddr & Smart Meter
Based on mbeddr and Smart Meter, we consider language extension a very fruitful approach.

We have also used it in other domains, including robot control, engine management and insurance product definition.
Using real industry projects as case studies yields practically meaningful results, despite the drawbacks.
Using C Language Extensions for Developing Embedded Software - A Case Study

Markus Völter  voelter@acm.org
Arie van Deursen  Arie.vanDeursen@tudelft.nl
Stephan Eberle  stephan.eberle@itemis.com
Bernd Kolb  Bernd.kolb@itemis.de