Smart Meter: an mbeddr Case Study

Markus Voelter[1] and Bernd Kolb[2]

[1] independent/itemis
[2] itemis

Abstract. This case study looks at the first real-world mbeddr project that develops a smart meter. A smart meter is an electrical meter that continuously records the consumption of electrical power and sends the data back to the utility provider for monitoring and billing. The software comprises ca. 30.000 lines of mbeddr code. Several of the mbeddr default extensions are used, and additional, project-specific extensions have been developed. Three developers are involved in the project at this time, including people from the core mbeddr team as well as outsiders. While the project is still going on, we can already report some positive experiences and draw some conclusions.

Last Changed: Sept 30, 2012

1 Contacts for more Info

The SmartMeter system is developed by itemis France. Technical contacts are Bernd Kolb (`kolb@itemis.de`) and Stephan Eberle (`eberle@itemis.de`).

2 What is the problem/domain your system addresses?

A smart meter is an electrical meter that continuously records the consumption of electric power in a home and sends the data back to the utility for monitoring and billing. The particular software we develop will run on a 2-chip board (TI MSP-430 for metrology, another TI-processor (tbd.) for the other application logic). Instead of the `gcc` compiler used in mbeddr by default, this project uses an IAR compiler.

The software comprises ca. 30.000 lines of mbeddr code, has several time-sensitive parts that require a low-overhead implementation and will have to be certified by the future operator. The software exploits existing code in the form of header files, libraries and code snippets. While the project is still going on, we can already report some experiences and draw some conclusions.

3 Why was mbeddr/MPS chosen as the basis?

We chose mbeddr to implement the Smart Meter for the following reasons:

- We have to work with an existing code base of questionable quality. We needed to improve the code quality *incrementally*. So starting with the existing C code and then refactoring towards better abstractions seemed like a good idea.
- The existing C extensions provided by mbeddr are a good match for what is needed in Smart Meter (see next section). In particular, Smart Meter is *not* just a state-based or a data flow-based system, it contains multiple different behavioral paradigms. Using a state chart or data flow modeling tool was hence not an option.
- As the goal is to have the meter certified, testing the software is very important. By using the abstraction mechanisms provided by mbeddr, and by exploiting the ability to build custom extensions, testability can be improved significantly. In particular, hardware-specifics can be isolated, which enables testing without the actual target hardware. Also, mbeddr's support for requirements traceability comes in handy for the upcoming certification.

4 How does the system integrate/use mbeddr's existing capabilities?

In Smart Meter, we use the following default extensions:

- *Components:* We use mbeddr's components to improve the structure of the application, and to support different implementations of the same interface. This improves modularity and testability. Mocks components are used excessively for testing.
- *State Machines:* The smart meter communicates with its environment via several different protocols. So far, one of these protocols has been refactored to use a state machine. This has proven to be much more readable than the original C code. We combined components and state machines which allowed us to decouple message assembly and parsing from the application logic in the server component.
- *Requirements Tracing:* Smart Meter also make use of requirements traces. During the upcoming certification process, these will be extremely useful for tracking if and how the customer requirements have been implemented. Because of their orthogonal nature, the traces can be attached to the new language concepts specifically developed for Smart Meter (see next section).
- *Units:* A major part of the Smart Meter application logic performs computations on physical quantities (time [`s`], current [`A`] or voltage [`V`]). So mbeddr's support for physical units comes in handy. The benefits of this extensions are mostly in type checking, but some conversions are also generated into the resulting C code. Using types with units also improves the readability and comprehensibility of the code (important in Smart Meter, which relies on a significant existing code base).

Summing up, the mbeddr default extensions have proven extremely useful in the development of Smart Meter. The fact that the extensions are directly integrated into C (as opposed to the classical approach of using external DSLs or separate modeling tools) reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.

It also support incrementally adding additional extensions as the need arises.

5 How was mbeddr extended?

So far we made use of this possibility in the Smart Meter project in the following ways:

– *Registers:* The Smart Meter software makes excessive use of registers (metrology: access the sensor values, UART: receive/send data). This cannot be abstracted away easily due to environmental constraints and how the software is written. In addition some registers are special purpose registers: when a value is written to such a register, a hardware-implemented computation is automatically triggered based on the value supplied by the programmer. The result of this computation is then stored in the register. If we want to run code that works with these registers on the PC for testing, we face two problems: first, the header files that define the addresses of the registers are not valid for the PC's processor. Second, there are no special-purpose registers on the PC, so no automatic computations would be triggered. We solved this problem with a language extension that allows us to define registers first class and access them from C code (see code below). The extension also supports specifying an expression that performs the computation. When the code is translated for the real device, the real registers are accessed using the processor header files. In testing we use generated `struct`s to hold the register data and insert the expression into the code that updates the `struct`, simulating the hardware-based computation.

```
exported register8 ADC10CTL0 compute as val * 1000

void calculateAndStore( int8 value ) {
  int8 result = // some calculation with value
  ADC10CTL0 = result; // actually stores result * 1000
}
```

– *Interrupts:* Many aspects of the Smart Meter system are driven by interrupts. To integrate the component-based architecture used in Smart Meter with interrupts, it is necessary to trigger component runnables (methods) via an interrupt. To this end, we have implemented a language extension that allows us to declare interrupts. In addition, the extension provides runnable triggers that express that a runnable is triggered by an interrupt.

```
module Processor {
  exported interrupt USCI_A1
  exported interrupt RTC
}
exported component RTCImpl {
  void interruptHandler() <- interrupt {
    hw->pRTCPS1CTL &= ~RT1PSIFG;
  }
}
```

The extension also provides a concept to assign an interrupt to a runnable during component instantiation. We also check that each interrupt-triggered runnable has at least one interrupt assigned. In addition, for testing purposes on the PC, we have language constructs that simulate the occurrence of an interrupt: the test driver simulates the triggering of interrupts based on a test-specified schedule and checks whether the system reacts correctly.

– *Emulation:* Many parts of the Smart Meter application logic (beyond the component runnables discussed above) are triggered by interrupts, reacting to hardware events or timeouts. When testing the system on a PC, these external triggering events are not available, or course. So we started to write an emulation layer which will trigger interrupts based on various user-defined expressions. For example, an interrupt can be triggered every few seconds or once the value of a given register is changed. This will enable us to write integration tests. The emulator will not be a hardware emulator (emulating instructions) but rather a simple way of testing the interactions between different interrupt-triggered components.

6 What is your conclusion?

mbeddr works for Smart Meter. The mbeddr default extensions have proven extremely useful in the development of Smart Meter. The fact that the extensions are directly integrated into C (as opposed to the classical approach of using external DSLs or separate modeling tools) reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.

Language Modularity Works. Building a language extension should not require changes to the base language. The extensions for Smart Meter demonstrate this point. The registers extension discussed above requires new top level module contents (the register definition themselves), new expressions (for reading and writing into the registers), and embedding expressions into new contexts (the code that simulates the hardware computation when registers are written). All of those have been built without changing C. Similarly, the interrupt-based runnable triggers have been hooked into the generic trigger facility that is part of the components language. Even the units extension, which provides new types, new literals, overloaded typing rules for operators and some adapted code generators has been developed in a modular way.

Once a language is designed in a reasonable way, the language (or parts of it) should be reusable in contexts that had not been specifically anticipated in advance. The Smart Meter system contains examples: expressions have been embedded in the register definition for simulating the hardware behavior, and types with units have been used in decision tables. Again, no change to the existing languages has been necessary.

Language Development Efforts are ok. As an example, the implementation of the language extensions for registers (and the simulation for testing) was done in half a day. The addition of interrupts, interrupt-triggered runnables and the way to ”wire them” up was ca. one day. In the context of a development project which, like Smart Meter, is planned to run a few person years, these efforts can easily be absorbed. The benefits are well worth the effort in terms of the improved safety and testability.

The generated C code is efficient enough. Generating code from higher-level abstractions may introduce performance and resource consumption overhead. While we have not yet performed a systematic analysis of the overhead incurred by the mbeddr extensions, it is low enough to run the Smart Meter system on the hardware intended for it. Some extensions (registers, interrupts or physical units) have no runtime overhead at all since they have no representation in the generated C code. Others, such as the components, incur a very small overhead as a consequence of indirections from function pointers (to support polymorphism).

Interop with textual code is a bit painful, but works. Additional effort is required to integrate with existing legacy code. As a consequence of the projectional editor, we have to parse the C text (with an existing parser) and construct the MPS AST. mbeddr provides an importer for header files as a means of connecting to existing libraries. However, mostly as a consequence of C's preprocessor which allows all kinds of mischief to be done to otherwise well-structured C code, this importer is not trivial. For example, we currently cannot import all alternatives expressed by `#ifdef`s. Users have to specify a specific configuration to be imported (in the future, we will support importing of all options by mapping the `#ifdef`s to mbeddr's product line variability mechanism). Also, header files often contain platform-specific keywords or macros. Since they are not supported by the mbeddr C implementation, these have to be removed before they can be imported. The header importer provides a regular expression-based facility to remove these platform specifics before the import. The Smart Meter project, which is heavily based on an existing code base, also drove the need for a complete source code importer (including `.c` files, and not just header files), which we are currently in the process of developing.

The integration of legacy code describe in this paragraph is clearly a disadvantage of projectional editing. However, because of the advantages of projectional editors discussed in this paper, we feel that it is a good trade-off.

mbeddr scales reasonably. We have performed scalability tests and found that mbeddr scales to at least the equivalent of 100,000 lines of C code in the developed system. These tests were based on automatically generated sample code and measured editor responsiveness and transformation times. While there are certainly systems that are substantially larger, a significant share of embedded software is below this limit and can be addressed with mbeddr. The Smart Meter system is 30.000 lines of mbeddr code. Since there is a factor of ca. 1.5 between the mbeddr code and generated C, the Smart Meter system corresponds to ca. 45.000 lines of C.

More generally, Smart Meter did not run into any significant performance problems with mbeddr or MPS. There is some overhead involved because of the additional generation step (which is not present in regulae C code). On the other hand, mbeddr reports all kinds of errors directly in the IDE without relying on the compiler. As compiling the current code base takes between 5 and 45 seconds (depending on the scope of the rebuild) the turn-around time has actually improved. In addition, productivity of the developers has improved due to the very precise code completion, search capabilities, as well as the possibility to easily rename and move various program elements.

The MPS IDE however requires enough memory. The users notebooks were equipped with dual core processors as well as 8GB of RAM running Windows 7 or Mac OS.

The learning curve for C programmers is acceptable. Based on initial experiences from the Smart Meter project, an end-user will need a few days to get used to the projectional editor and the changes mbeddr C makes relative to regular C. After this learning phase, some developers report that they actually *prefer* the projectional editor over a textual one.

One criticism that has been used against language extension is that the language will grow large and that it is hard for users to learn all its constructs. In our experience, this is not a problem in mbeddr for the following three reasons: first, the extensions provide linguistic abstractions for concepts that are well known to the users: state-based behavior, interfaces and components or test cases. Second, the additional language features are easily discoverable because of the IDE support. Third, and most important, these extensions are modularized, and any particular end user will only use those extensions that are relevant to whatever his current program addresses. This avoids overwhelming the user with too much ”stuff” at a time.

7 Status? Is it in use yet?

The Smart Meter is a real-world project. A substantial part of the to-be-built system has been implemented. Test cases run, and the code runs on the actual target device with acceptable performance. The system continues to be developed, and the project is on track.

So far, the Smart Meter project has proven to be a good case for development with mbeddr. The teams (mbeddr team and Smart Meter team) are still convinced that it has been the right decision to use mbeddr for the project.