Software Engineering in Embedded Systems

Stephan Heidinger

March 12, 2012

Contents

1 Introduction

This report will introduce an approach to the development of *Embedded Software Systems* and *Embedded Systems* as a whole. The report is based on the Book *"Software Engineering"*[Som11, chapter 20].

1.1 Orientation

Before talking about how to develop software for *Embedded Systems* I would like to establish what Embedded Systems are. Unfortunately there is no strict definition. But when looking into the subject, one finds certain points surfacing in many definitions and descriptions:

- Embedded Systems respond to a physical world.
- Embedded Systems have to respond in real time.
- Embedded Systems often have only small amount of resources.
- Embedded Systems often run on special purpose hardware.
- Embedded Systems use real-time operating systems.

Therefore it is convenient to stick to Sommervilles definition of *Embedded Systems*:

> *"An* embedded software system *is part of a hardware/software system that reacts to events in its environment. The software is 'embedded' in the hardware. Embedded systems are nominaly real-time systems." [Som11, p. 561]*

Embedded Software is the part of an Embedded System, that consists of software.

1.2 Motivation

Why would I want to talk about Embedded Systems? Well, they are everywhere. When we look around ourselves, we'll quickly realize, how many Embedded Systems there actually are and that there are probably more of them than regular computers. Among them we find phones, routers, burglar alarms, coffee machines, any automated system in a car like airbags or distance warners and many more. Upon realizing how many Embedded Systems we use in our daily life, we certainly realize, that embedded systems must be quite important.

Personally I choose this topic, because prior to starting my studies in Constance I did an internship producing a monitoring device with the ability of performing lesser control functions for detectors used in the neutron spallation source SINQ at the Paul Scherrer Institute (PSI) in Villigen, Switzerland.

2 Embedded Systems Design

2.1 Problems

Because of the special circumstances of Embedded Systems we are faced with some problems, that are not important, or at least not as important in regular software.

Deadlines:
Embedded Systems have to react in real time. Therefore they have to meet certain deadlines upon which results have to be ready or certain actions been taken. Deadlines are probably the most important problem we are faced with when developing Embedded Software. Therefore Embedded Systems can be divided into one of two categories depending on the results to not meeting a deadline.

Hard Software Systems are systems where the whole programm will fail, if a deadline is not met. This includes e.g. safety critical applications like airbags, ejection seats, ...

Soft Software Systems are systems, where the result will degrade when deadlines are not met. Eventually the system will fail with an increasing number of unmet deadlines. This includes e.g. signal processing, signal transmission, ...

Environment:
Embedded Systems have to respond to a physical world. This physical world is not a single state world. Rather it is constantly changing, which has to be taken into account when developing Embedded Systems. We may need to react to multiple events at the same time and also need to verify that a result is still valid upon producing this result. This could best be achieved with a concurrent design, but when we encounter really short deadlines, concurrent languages may not be fast enough.

Continuity:
In many cases Embedded Systems run continuously, they never terminate. Therefore Embedded Software has to be reliable, because it is not feasible to just restart them when encountering an error. Additionally we may need to be able to update the software while it is running.

Direct Hardware Interaction:
As Embedded Systems do a wide variety of work, we will encounter a similar variety of specialized and uncommon hardware, e.g. detonators in an airbag, special sensors, special output devices. In some cases this hardware may even be designed directly for our system. It is therefore very probable, that we need to develop drivers for the used hardware along with the Embedded System to be able to use it.
In some cases, when our system cannot possibly meet some deadlines, it is advisable to implement some functions in hardware instead of in software, as this is generally faster.

Safety & Reliability:
Embedded Systems are in many cases responsible for the well-being of living creatures, e.g. airbags, ejection seats, handle dangerous material, e.g. in a nuclear power plant, or are otherwise used in processes with potentially dangerous to catastrophic results upon failure. These failures may then lead to high costs, either economicaly or in (human) life. To reduce these risks special care needs to be taken to ensure correctness of such systems.

2.2 Design Steps

Certain decisions about hardware, e.g. its performance, costs, power consumption (especially in mobile devices), strongly affect the overall performance of the system. As these parts are not easily exchangeable in Embedded Systems they have to be given early consideration and Sommverille does not encourage a top-down approach[Som11, p. 540]. He also emphasizes strongly, that there is no standard system design process for Embedded Systems. Instead of that, he proposes some design steps, that are sensible to use. Still, as there is no standard approach, some of these steps may not be feasible for a certain system and one has to think about which steps to use and which to drop. Also there is no fixed order in which to do the steps.

Plattform Selection
The plattform selection consists mainly of the selection of the used hardware, the real-time operating system and the used programming language(s). The main factors influencing this decision are timing contraints, the kind of power available and its limitations (e.g. mobile devices), the price of the finished system and the experience and preferences of the development team.

Stimulus/Response Analysis
An Embedded System can be described by a list of stimuli and their corresponding responses. A stimulus is an event the system has to respond to. The Stimulus/Response Analysis consists of thinking about what is going to happen to the system and planning appropriate responses. Then these stimuli and responses are composed into a list.

When thinking about stimuli we have to differentiate between periodic and aperiodic stimuli:

Periodic stimuli describe stimuli that occur once in a fixed period. They often describe the normal state of the system, where nothing special happens (e.g. polling of a sensor).

Aperiodic stimuli describe stimuli, that happen unexpectedly and irregularly. Often they represent special situations, where action of the system is required (e.g. alarms, failures, ...).

Timing Analysis
The goal of Timing Analysis is to find timing contraints for each stimulus/response pair. These timing contraints can then be transformed into deadlines. The system needs to be designed in order to be able to meet these deadlines. This can be achieved through *static analysis* or *simulation* and will be discussed later.

Process design
The Process Design step aggregates the stimli and responses into a set of (concurrent) processes. This is further discussed in *Architectural Patterns*.

Algorithm Design
In the Algorithm Design step we transform each stimulus/response pair into an algorithm. This step is especially important for computationally intensive tasks like signal processing. If we cannot design an algorithm fast enough to accomplish deadlines, we may also decide to implement some algorithms in hardware.

Data Design
The Data Design step covers the design of data structures to store all data. As we may have concurrent processes, we have to ensure, that the data stays consistent. Common practices are the use of *semaphores*, *critical regions* and *monitors*.

When processes work at different speeds, especially if we have a producer/consumer situation, the use of *(circular) buffers* is advisable.

Process Scheduling
A scheduling algorithm has to be devised or found, that ensures the meeting of the set deadlines.

Special Purpose Hardware
In this step one must think about what to built in hardware and what in software including thought about uncommon vs. common hardware. If we encounter bottleneck algorithms in either the Timing Analysis or Process Scheduling steps we may think about implementing those potential bottlenecks in hardware (e.g. FPGA).

2.3 Real-time System Modelling

Events or stimuli in an Embedded System often change the system to change its state. Therefore it is often convenient to describe and model Embedded and Real-time Systems as *state models*. The UML statecharts are a good way to visualize the state model thus improving the overall understanding of the system.

2.4 Real-time Programming

The programming of Embedded Systems does also have to take care about the deadlines. Therefore one might want to use assembler languages or system-level languages as C. These languages provide the advantage of high efficiency, but have a downside as they do not have built-in libraries or faculties to manage shared resources or concurrency. Object Oriented languages as C++ or (embedded) Java provide these features, but come with a huge overhead due to hiding data representations in objects and thus reducing speed.

The development team now has to decide, whether they want to use less error-prone object oriented languages, or need the speed offered by the faster languages. Although faster systems (e.g. cell phones) start to appear more often, therefore reducing the need for fast languages, systems with very limited resources and the need for fast languages still exist.

3 Architectural Patterns

According to Sommerville, Architectural Patterns are not supposed to be a *"generic design to be instantiated"[Som11, p. 547]*. He wants them to be understood as a means to gain and infer knowledge about the system, describing them as *"encapsulat[ing] knowledge about the organisation of system architectures, when these architectures should be used and their advantages and disadvantages"[Som11, p.547]*. Furhtermore he describes only three rough patterns that would lead to a very inefficient system if not optimized before implementation. Still these patterns are usefull for a first design approach until more knowledge about the workings of the system can be generated.

3.1 Observe and React

The *Observe and React* pattern is mainly used for monitoring systems. It consists of a set of sensors and most commonly some kind of display. During normal operation the system checks each sensor according to timing contraints and most commonly displays the state of the environment. When the system encounters exceptional situations the system may flash an alert via changing the display, activating an alarm or contacting someone directly. In some cases it may be convenient to also take some precautionary action like shutting down a system to prevent damage.

In a first approach it is acceptable to use one display for every sensor, but iterative improvements may merge displays until eventually the system will result in a small number of display devices.

3.2 Environmental Control

The *Environmental Control* pattern is used in systems, where the system controls the state of the environment. It consists of a set of sensors and a set of actors. Through the sensors the system learns about the current state of the controlled environment. If this state differs from the desired state, the system will take direct action to correct this state back towards the desired one. In most cases there will be monitoring of the actors as well, althought the control and monitoring of actors may be merged for increased performance or feedback loops may be integrated. A display or set of displays to present the current state may be sensible for users to get an overview.

This pattern is usefull, when human supervision is not practical, for example when a simple task like the keeping of a fluid level is required, the system has to deal with many quickly changing variables like in an airbag or ABS system or when there are just to many variables to take into consideration.

3.3 Process Pipelining

The *Process Pipelining* pattern is used in systems that have to work with a huge amount of data at the same time (e.g. data collection or data transformation). As the data may need to be processed quickly to prevent the loss of incoming data, the pattern breaks the processing up into small steps, which are then executed by several concurrent processes. This approach is especially efficient

in systems using multiple or multicore processors. As results are produced and consumed, use of buffers is advisable.

Another variant of the *Process Pipelining* pattern are data aquisition systems, where huge amounts of data are collected and have to be stored for later processing. It is common to use buffers to temporarily store the data before it can be written to storage.

4 Timing Analysis

As expressed before, the meeting of deadlines is essential in Embedded Systems. *Timing Analysis* is used to ensure that the system is able to meet its deadlines, by calculating how often each process needs to be started so that results are produces before its deadline. In hard software systems timing analysis must be done, while it is strongly recommended in soft systems.

As Embedded Systems normaly have to cope with a mixture of periodic and aperiodic stimuli, timing analysis is hard, as we have to predict how often aperiodic stimuli occur and our prediction may just be wrong. When the software runs on faster hardware one might decide to simulate aperiodic stimuli as periodic stimuli. In this case one has to make sure, that the cause of the aperiodic stimulus is checked often enough, that approriate action can be taken in order to meet the deadline.

Sommerville describes three key factors that have to be considered in timing analysis:

Deadlines are the points in time, where the stimulus must be processed and a result be available.

Frequency is the number of times per second a process must be executed in order to be able to meet its deadlines.

Execution Time is the time needed to process the stimulus and produce the result. You have to distinguish between *average* and *worst execution time*. In hard software systems you will always need to consider the worst case time.

When you gather this data, it is preferable to list each kind of sensor/stimulus seperately, even if they have the same timing requirements, as this allows more easily for future changes.

5 Real-time Operating Systems

An integral part of every Embedded Software System is the operating system the software is running on. In most cases, regular operating systems are not needed for the task of the Embedded system. Those systems tend to be slow and usually don't allow for a very fine grained control required in Embedded Software. This is due to nice graphical user interfaces, huge amount of unneeded libraries. The operating systems in Embedded systems are called *Real-time Operating Systems* or *RTOS*. These are mostly systems, that are reduced to a core functionality, thus gaining speed and giving more direct control to the user. Some common systems are *emdebian*, *Windows/CE*, *VXWorks* and *RTLinux*.

The interesting question presenting itself is, of course, what those "core funtionalites" are. According to Sommerville there are five things every RTOS must feature:

A real-time clock that delivers information for the scheduler.

An interrupt handler which allows important processes to interrupt other, less important processes.

RTOS' have to be able to manage at least two different priorities, *interrupt level* and *clock level*. The former are processes with a high priority like processes with a fast reaction time or a close deadline. The real-time clock is one of these processes. They have to be allowed to interrupt other processes for the system as a whole to work and be successfull. The latter are primarly periodic processes.

There may also be more levels to allow for finer tuned distinction and for background processes that are not very important. Also there may be distinctions inside those levels. The correct allocation of priorities requires extensive analysis and often simulation.

A scheduler for deciding which processes to execute at which time.

Schedulers can be seperated into two classes: *pre-emptive* and *non-pre-emptive*, where the former is allowed to stop running processes to allow other processes to run while the latter lets a process run until termination upon starting it.
Common strategies are *round-robin*, where each process is granted a time-slot until the next process is executed, *rate monolithic* or *shortes job first* scheduling, where the job with the shortes execution time will be started first, and *shortes deadline first* or *highest priority first*, where the job with the highest priority get preference over other processes.

A resource manager for allocation of system resources (e.g. memory and processor time) to the processes.

A dispatcher to start execution of processes.

6 Examples

6.1 Radiation Warning System

Imagine we want to build an experiment environment around a nuclear reactor. The whole complex is sufficiently shielded, so we don't need to take care about any radiation on the outside. Requirements are, that we have multiple rooms adjacent to the reactor, where we want to conduct our experiments. As we have people working in these rooms, we need to make sure radiation levels inside our complex are below certain thresholds. We install sets of sensors in every room to check the radiation levels. This will lead us to something like the layout shown in figure 1.

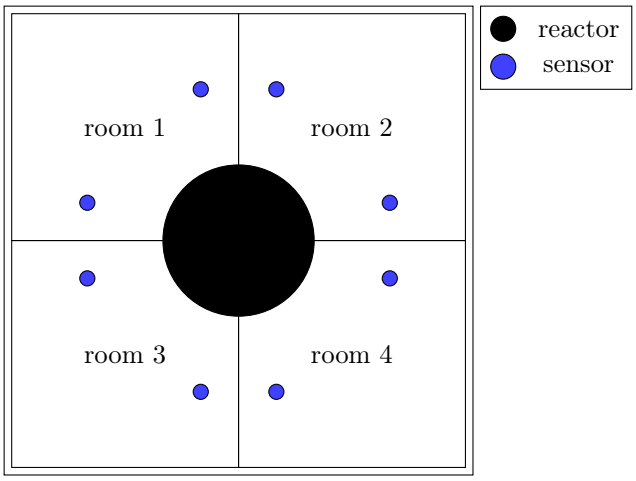

Figure 1: Sketch of our experiment environment setup.

Stimuli/Response Analysis

Next we need to look at the possible stimuli we may encounter in this environment and the responses we may want to take. When a single sensor detects a violation of the radiation threshold ('goes positive'), we are not overly concerned, as this has to be confirmed by at least one more sensor. Nevertheless a yellow warning light around the sensor should be flashed, so the people working here will be informed. As soon as a second sensor in the same room also detects a threshold violation, we are certain enough, that something is wrong and we flash a red warning light in the whole room to inform the people working here. Additionally we sound an acoustic alarm.

When the system detects a low voltage drop of $10 - 20\%$ this might still be a normal fluctuation, still we want to run a power supply test. When the voltage drops further ($> 20\%$) we switch to a backup power supply and notify a technician to reestablish power to the system.

This will eventually lead to a listing as proposed in table 1.

Stimulus	Response
single sensor in one room positive	flash yellow light around sensor
two sensors in one room positive	flash red light in the room, sound acoustic alarm in the room
Voltage drop of $10 - 20\%$	run power supply test
Voltage drop of $> 20\%$	switch to backup power, run power supply test, call technician

Table 1: List of stimuli and responses.

System Modelling

For the understanding and presentation of the system, we want to represent the system as a state chart using the UML language as shown in figure 2. In the

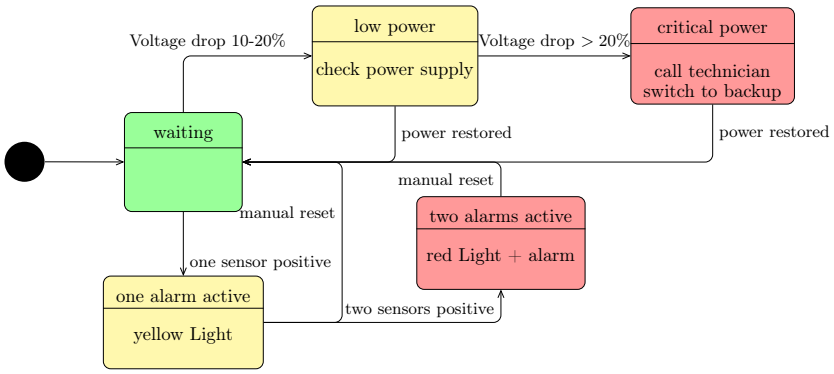

Figure 2: UML state chart of our system.

'waiting' state we do nothing. When the first sensor in a room goes positive, we go into an 'one alarm active' state which can be left by manual interaction or a second sensor going positive. While the former brings us back into the waiting state, the latter would lead us into the 'two alarms active' state which can only be left by manual interaction. Upon a voltage drop of $10 - 20\%$ we go into the 'low power' state and upon a further drop we go into the 'critical power' state, both of which will be left for the waiting state upon power being restored.

Timing Analysis
As timing requirements we define the requirements in table 2. We need to make sure, that we poll each sensor once inside $500ms$. As we have four rooms with two sensors per room, and an additional sensor for the power supply, we get a total of 9 sensors. Considering that we have to wait for two sensors two become positive and including a safety margin of 20%, we need to run our polling process at least once every $t = \frac{0.8 \cdot 500ms}{9 \cdot 2} = 22ms$. The other timing requirements a mostly dependend on how the hardware is built, as it takes only a short impulse to activate a light or initate the alertion of the technician.

Stimulus/Response	Timing Requirement
voltage drop detection	switch to backup: 50ms
sensor reaction	poll twice a second
light activation	500 ms
alert technician	5000ms

Table 2: List of Timing Requirements

7 Assessment

During my internship at the PSI I produced an Embedded System without any specific knowledge of Software Engineering or the special requirements for Embedded Systems themselves. Many of the steps described by Sommerville were done by the people working at PSI, e.g. the choice of hardware, RTOS and programming language.

In the early stages of the project we ran into some misunderstandings concerning the design of the system. If I had done proper system design, this misunderstandings could probably have been prevented as the misunderstandings would have been discovered in the state charts.

A huge problem I encountered was scheduling the polling of sensor data. At a first look, this problem could have been solved through proper timing analysis. I however think timing analysis may have helped a little, but the problem would still have persisted, as the system design required a possibility for the user to change the priority of the sensors during runtime and the system should additionally reduce the frequency of polling if the system state stays consistent for a certain amount of time (which again could be set by the user).

Although not everything Sommerville describes may be usefull in every Embedded System design process, I think this introduction into the field of Embedded Software Engineering field is really usefull. Embedded System design is as broad a field, as Embedded Systems are. They do themselves have a huge variety in how they are built, how they work and what they do, so that a single chapter will only be able to provide a lookout into this field. When planning to work into this field, more specific study will be required.

8 Conclusion

I think the most important thing in Embedded Software Development is to allways keep in mind, that your system has to react to the real world in real time. The system reacts to stimuli from this real world and has to deliver this result before it is needed. Tools like state models and architectural patterns may help in the first steps to better understand the system und thus arrive at a more efficient system. Timing analysis is very important as soon as deadlines are present.

References

[Som11] Ian Sommerville. *Software Engineering*, volume 9. ed, international ed. Pearson, Boston, Munich, 2011.