Avoiding the Top 43 Embedded Software Risks

Philip Koopman
Carnegie Mellon University
ECE Dept., HH A-308
Pittsburgh, PA 15213 USA
http://BetterEmbSW.BlogSpot.com/

ABSTRACT

This paper briefly distills the lessons learned from almost 100 reviews of industry embedded software projects. What I found was that even developers without formal training tend to get the basics right if they have spent time working their way through intro to embedded books and a few hands-on learning projects. There are a number of more advanced technical problems that tend to surface in embedded projects (for example, concurrency management). But, more of the risks stem from having a poor engineering process, cutting corners on quality assurance, or believing all you need is source code to succeed. As design engineers we might like to dwell on nitty-gritty technical aspects. But there is more to success than that. Embedded developers need to deal with the higher level risk areas I identify, and can benefit from the concrete best practices described.

1. INTRODUCTION

This paper identifies 43 risk areas for real products in a variety of embedded system application areas. The items were identified over the course of almost 100 design reviews conducted by the author over more than a decade.

Application areas covered by the reviews include: transportation, chemical processing, building infra-structure, telecommunication systems, manufacturing, and a few cross-cutting underlying technologies such as embedded system safety and security. Systems were about evenly divided between small micro-controllers and bigger CPUs that ran some sort of RTOS. Team sizes ranged from one part time developer to teams of up to 25 developers, with most programs being small or medium in size and written in assembly language, C, or C++. Most reviews included a day of on-site discussions with the developers. Many developers were domain experts with mechanical or electrical engineering background, although some had formal computer software education. A more detailed description of the study, including more detailed description of each risk area, can be found in [1].

2. RISK AREAS

Each review engagement produced a set of recommendations, including "red flag" issues that present significant and immediate risks to the success of the project or product. In other words, a red flag area is one that should be fixed before sending the product to market (or, if the product is already on the market, needs to be fixed as soon as possible to limit the extent of the risk). The table on the next page lists all the areas in which one or more reviews identified a red flag risk.

It's important to point out that not every project needs to be perfect. Rather, it's important to mitigate or avoid risks that are significant in the context of a particular product and application area. This list of red flag risks takes that context into account. In particular, each identified risk area was a significant problem for a particular project being reviewed, not simply just a best practice that had been skipped.

Several high level patterns can be seen in the list of risks, both in terms of what is there and that is not there.

2.1 THE BASICS

Overall teams tended to get the basics right. No team had serious problems getting an A/D converter working, getting a serial port working, or with other intro-level embedded skills. (This is not to say that problems of this sort never happen. Rather, this sort of problem tends to be caught and fixed by developers before external reviewers are brought in.) In part this is because information about those topics is readily available through a variety of sources. But, just as importantly, it is relatively obvious when something

like that isn't working in a product. You don't need an outside expert to tell you that you're A/D converter isn't collecting data!

43 Embedded Software Risk Areas

#1. Informal development process
#2. Not enough paper
#3. No written requirements
#4. Requirements omit extra-functional aspects
#5. Requirements with poor measurability
#6. No defined software architecture
#7. Poor code modularity
#8. Too many global variables
#9. No message dictionary for embedded network
#10. Design skipped or is created after code is written
#11. Flowcharts are used in place of statecharts
#12. Inconsistent coding style
#13. Ignoring compiler warnings
#14. No peer reviews
#15. No real time schedule analysis
#16. Use of home-made RTOS
#17. Inadequate concurrency management
#18. No methodical approach to user interface design
#19. No test plan
#20. No stress testing
#21. No defect tracking
#22. No run-time fault instrumentation nor error logs
#23. Defect resolution for 3rd party software
#24. Disaster recovery not tested
#25. Insufficient consideration of reliability/availability
#26. Insufficient consideration of safety
#27. Insufficient consideration of security
#28. No IP protection plan
#29. No or incorrect use of watchdog timers
#30. Inadequate system reset approach
#31. High requirements churn
#32. No version control
#33. No backward compatibility plan
#34. No software update plan
#35. Lessons learned not being recorded
#36. Acting as if software is free
#37. Use of cheap tools instead of good ones
#38. High turnover and developer overload
#39. No training for managing outsource relationships
#40. Resources too full
#41. Too much assembly language
#42. Project schedule not taken seriously
#43. No Software Quality Assurance (SQA) function

2.2 ADVANCED TECHNICAL RISKS

A number of the risks are advanced technical areas specific to embedded systems, and are the types of things that most experienced developers would expect to be risk areas. Inadequate concurrency management, incorrect use of watchdog timers, poor modularity, and other similar topics are the sorts of things that developers often get wrong. They are also the types of things I thought would be the most prevalent risks.

An important reason why these problem areas are common is that it is difficult to tell there is a problem via ordinary system testing. If a system is working fine in normal use, you can't really tell whether the watchdog timer has been set up properly. Nor are you likely to find bugs caused by subtle and infrequent race conditions. But that doesn't mean such problems aren't there, especially if you didn't think to look for them.

2.3 BEYOND TECHNICAL RISKS

To my continuing surprise (I am, after all, a techie), by far most of the red flag risk areas weren't really technical; they were in the areas of process, quality, management and other softer areas. In other words, just looking at the code itself doesn't reveal where most of the risks are. You have to take a broader look at the bigger development picture.

Also, most of the risks weren't caused by developers slacking on things they were trying to do. By and large, the developers I encountered were smart, hard-working, well-intentioned designers. They were trying to do the right thing, and usually did well at whatever they set out to do.

The most prevalent problem was developers not realizing there were additional things they should be doing, and not realizing the huge risk that omitting those extra activities were placing on their success.

Some examples of risks caused by missing pieces of development process include:

- Design was skipped, leading to spaghetti code that seemed to (mostly) work, but was almost impossible to understand or maintain
- Peer reviews were skipped, leaving far too many bugs to be found (and missed) by testing
- No formal test plan was used, leaving gaps in testing that let bugs slip through to product

3. AVOIDING THE RISKS

In an ideal world, every product development team would get a senior developer from outside the project to do a review to identify risks, using the risk areas in this paper as a starting point. (And, you should do that if you can!)

But the world is not ideal. Even if you can get such a review, not all risks are created equal for every project. So, some common sense and prioritization has to be done to make sure the risks that matter most are the ones that are mitigated.

Nonetheless, there are some general guidelines that are likely to help keep development teams on the path to avoiding big risks. These guidelines summarize the major points of [2], which was created as a way to help developers deal with these same 43 risk areas. Think of these as guidelines that will get things moving in the right direction so that risk-hunting is likely to come up with few, if any, killer risks for a project.

1. Define your development process. If the process goes out the window in a time crunch, you'll likely pay for it later. Use the right amount of formality for your situation. (Zero paperwork isn't the right answer.)
2. Define requirements in a measurable, traceable way. If you can't measure a requirement you don't know if you met it.
3. Document your software architecture. Make your code modular, and avoid global variables.
4. Create a concrete software design before you write code. If you don't use any statecharts, something is wrong.
5. Follow a defined, consistent coding style. Use static checking tools to keep your code clean.
6. Do methodical peer reviews of everything (requirements, code, test plans – everything!).
7. Do real time scheduling analysis, and use a 3rd party RTOS if you are doing preemptive task switching.
8. Pay attention to concurrency to avoid tricky timing bugs. Home-brew methods are risky.
9. Follow good user interface design principles. Do user testing (engineers don't count as "users").
10. Have a formalized, traceable test plan. Test until you reach a defined level of coverage.
11. Ensure that problems found both in test and run-time are identified and tracked.
12. Explicitly specify and plan for performance, dependability, security, and safety up-front. Most embedded systems involve all these, and it is painful to try to add them at the end of the project.
13. Ensure that the watchdog timer will trip if any task in the system hangs, and that system resets are safe.
14. Manage change (requirements, versions, patches, people, process).
15. All software is expensive. Don't act like it's free.
16. Leave plenty of slack resources. Use as little assembly language as possible (zero assembly code is a good amount).
17. Find a way to make sure all of the above practices are actually being done. Paying lip service just gives the illusion of risk mitigation

4. CONCLUSIONS

Red flag risk areas in industry embedded software projects included advanced technical topics, but also go well beyond just code. Mitigating these risks requires following a well-defined development process that ensures appropriate formality and effort is applied to the spectrum of development activities from requirements through test.

No project has infinite time and resources for development. The art is in making sure that all the areas that matter for your project are covered to an appropriate degree. It is almost never that case that entirely skipping one of the practices identified is a good idea. And while there are plenty of potential risks and development mistakes, the 43 identified in this paper are worth checking for based on what I've seen in industry design reviews.

5. REFERENCES

[1] Koopman, P., "Risk Areas In Embedded Software Industry Projects", Workshop on Embedded System Education, October, 2010.

[2] Koopman, P., *Better Embedded Software*, Drumnadrochit Education, 2010.