Risk Areas In Embedded Software Industry Projects

Philip Koopman
Carnegie Mellon University
ECE Dept., HH A-308
Pittsburgh, PA 15213 USA
+1 (412) 268-5225
Koopman@cmu.edu

ABSTRACT
A powerful way to understand where gaps are in the expertise of embedded system designers is to look at what goes wrong in real industry projects. This paper summarizes the "red flag" issues found in approximately 90 design reviews of embedded system products conducted over a ten year period across a variety of embedded system industries. The problems found can be roughly categorized into the areas of process, requirements, architecture, design, implementation, verification/validation, dependability, project management, and people. A few problem areas, such as watchdog timers and real time scheduling, are standard embedded education topics. But many areas, such as peer reviews, requirements, SQA, and user interface design seem worthy of increased attention in texts and education programs.

Categories and Subject Descriptors
J.7 [**Computer Applications**]: Computers in other systems – *industrial control, process control, real time.*

General Terms
Management, Documentation, Performance, Design, Economics, Reliability, Security, Human Factors, Verification.

Keywords
Embedded system education, software engineering, industry experience, design reviews, real time systems, software process.

1. INTRODUCTION
Most embedded education approaches stem from some attempt to create an overarching set of principles, list key topics, and adopt a particular teaching philosophy. That's a great basis from which to start. But, an interesting question is, what might be missing?

This paper looks at the problems and risks encountered by practicing embedded system designers. If they are making omissions or mistakes that materially affect the quality of their product or introduce undue risks to product success, then those areas seem reasonable to consider as potentially in-scope for embedded system education.

WESE'10, October 24, 2010, Scottsdale, AZ, USA.

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 more than 90 design reviews conducted by the author, spanning approximately the past 10 years. While the data points are self-selected and are vulnerable to reviewer bias, they nonetheless provide insight into what sorts of skill gaps and problem areas are present in embedded software projects.

2. BACKGROUND
The basis of this paper is a retrospective study of design reviews conducted by the author for a variety of embedded system companies. The companies are not identified to protect all parties involved, but most are divisions of large corporations or similar business entities which specialize in embedded systems. Such development groups would be expected to have mature and well organized procedures for designing and supporting moderate to large scale product deployments. A few reviews were of prototypes, but in all cases the developers were skilled, experienced, and tasked with designing real commercial products.

2.1 Product Types Included
The product types that were the subject of reviews generally include the following areas. This list is intended to give an idea of scope of the findings and does not necessarily include every single product:

- Transportation
  - Automotive control
  - Train control
  - Navigation
- Chemical processing
  - Metering and flow control
  - Chemical analysis
  - Process automation
- Buildings
  - Heating, Ventilation, and Cooling (HVAC)
  - Lighting and building security
  - Elevator and related transportation systems
  - Building utility services
- Telecommunication systems and data centers
  - Power regulation, switching, and backup
  - Environmental controls
- Manufacturing
  - Motion control and robotics
  - Inspection
  - Monitoring and equipment maintenance
- Underlying technology
  - Embedded real time control networks
  - Safety critical system design
  - Security

Some embedded application areas are absent from this data, such as consumer electronics and large military combat systems (although the author has had product experience with these areas in the past). There is no reason to believe these results are limited to the listed domains, but it seems plausible that concerns would vary depending upon which market segment a product is in.

All projects reviewed were predominantly software projects, although select hardware aspects were considered when appropriate for evaluating the system as a whole. Code size ranged from a few hundred bytes to about a million lines of code. Development team sizes ranged from one part-time developer to teams of up to 25 developers. (In some cases an overall system had many more developers, but only a specific subsystem was the topic of a review.) Most projects were in assembly language, C, or C++, but other languages were occasionally used. Developers most often followed Waterfall or Vee development approaches, but some products used Spiral, Incremental, or Agile approaches. Most reviews were of US-based teams, with a handful of reviews in Europe and Asia. Perhaps one fifth of US development teams used development partners or remote team members in India or China. In most cases remote developers participated in reviews either in person or via conference call.

Systems were about evenly divided between small microcontrollers and bigger CPUs that ran some sort of RTOS. A very few systems used DSPs or FPGAs, and none used custom or domain-specific silicon. Product volume ranged from prototypes to hundreds of thousands of units per year, although most reviews were for products in the 1,000 to 20,000 units per year range.

The results of some design reviews beyond those in the data set were excluded due to contractual obligations or because they did not result in formal review reports. But, if included, that data would not have materially changed the outcome of this study.

2.2 Design Review Process

A typical design review involves the steps of setting up the review engagement, learning some domain background, obtaining as many project documents as possible, selectively reviewing documents, setting a meeting agenda, traveling to hold an on-site review visit, and generating a written report. A minority of reviews to examine very specific areas or answer narrow questions were done via e-mail and phone, with no visit.

On-site visits typically lasted one day or, in some cases, two days. The amount of information available before the visit ranged from essentially nothing to thousands of pages of design information (often including complete source code listings). The degree to which developers self-identified problems before a visit varied, but most problems were identified by the reviewer without hints from the review team. More importantly, in almost all cases the review team accepted the problems identified as valid feedback. (This is not to say that every recommendation was necessarily carried out. But, for the most part, teams agreed that the areas identified as critical risks were in fact major issues that significantly affected the likelihood of project or product success.)

Perhaps a third of all the design reviews, mostly in the first years, were carried out by two independent reviewers in parallel, with a shared visit and jointly issued report. More recent reviews were single-reviewer events in large part due to economic constraints.

Most reviews were performed at about the time the product was ready to start acceptance testing or ready to be released. In cases where a problem was identified, an attempt was made to trace back to a reasonable root cause. For example, a bug-prone module might be identified as having implementation problems, design problems, architecture problems, or requirement problems depending on which stage in the design process was the most effective place to have avoided the bug (and other similar bugs).

Each review engagement produces a set of recommendations, including “red flag” issues that present significant and immediate risks to the success of the project or product. Other, less pressing, “yellow flag” risks and reviewer observations are also listed in review reports.

Over time, reviews became more formal and repeatable as a list of typical problem areas and review questions was developed using the input of a number of experts over the first few years of conducting reviews. A triage process based on this list was formally used for perhaps a third of the reviews, and the general knowledge of what was in this list informed most of the other reviews. The list presented in this paper does not strictly conform to the items in that proprietary checklist, but is similar in nature. The checklist has approximately three times as many topics as the red flag items discussed in this paper. In other words, two thirds of the entries on the checklist are worth asking about, but have failed to generate any red flag in a decade of performing reviews.

Very few reviews covered all checklist areas due to lack of time. Rather, emphasis was placed on areas that seemed to the reviewer and the developers to be the most likely place to be sources of major risk. The study presented here is solely concerned with red flag issues identified in one or more reviews.

There is no way to know how many significant risks were missed because reviewers didn’t think to ask the right questions. However, the chance of this happening was reduced by initially by the use of multiple reviewers, and in later years by the use of a comprehensive checklist-based triage process as just described.

2.3 Background of designers

The design teams reviewed varied in technical background significantly. Many team members had degrees and experience in mechanical or electrical (non-electronic) engineering. A number had electronic and computer engineering degrees. A few had computer science or, rarely, software engineering degrees. For the most part, senior developers started as domain specialists and picked up embedded computing on the job. Junior developers were more likely to have had software training of some sort, but usually had a non-software engineering background.

Over the years, there has been a trend for many design teams to advance to a higher level of software process sophistication (for example, many teams attained SEI Capability Maturity Model [6] Level 2 or above). This is in large part due to a concerted effort to improve software quality. It is also in part due to hiring of developers with formal software training into embedded system product teams. But, high process maturity is not universal across embedded projects. In particular, each company seems to find its own way up the software learning curve as it introduces the first non-trivial computing capability into its products and attempts to write software using domain experts who have little or no formal software training.

3. RISK AREAS IDENTIFIED

The following risk areas were identified as red flag (significant) risks in one or more reviews in this study. They are grouped and organized to provide some structure in terms of typical development process stages and activities. The ordering does not connote any severity or frequency. A typical item in this list was a red flag in one to five reviews and a yellow flag in several more.

The examples and likely consequences of risk areas given are generally representative of the risks actually seen without revealing company-specific information. (To the degree that examples or statements are true about any particular product, they are also true of many different products that were reviewed.)

3.1 Development Process

#1. Informal development process

The process used to create embedded software is ad hoc, and not written down. The steps vary from project to project and developer to developer. This can result in uneven quality.

#2. Not enough paper

Too few steps of development result in a paper trail. For example, test results may not be written down. Among other things, this can require re-doing tasks such as testing to make sure they were fully and correctly performed.

#3. No written requirements

Software requirements are not written down or are too informal. They may only address changes for a new product version without any written document stating old version requirements. This can lead to misunderstandings about intended product functions and difficulty in designing adequate tests.

#4. Requirements with poor measurability

Software requirements can't be tested due to missing or subjective measurement criteria. As a result, it is difficult to know whether a requirement such as "product shall be user friendly" has been met.

#5. Requirements omit extra-functional aspects

Product requirements may state hardware processing speed and hardware reliability, but omit software response times, software reliability, and other non-functional requirements. Implementing and testing these undefined aspects is left at the discretion of developers and might not meet market needs.

#6. High requirements churn

Functionality required of the product changes so fast the software developers can't keep up. This is likely to lead to missed deadlines and can result in developer burnout.

#7. No SQA function

Nobody is formally assigned to perform an SQA function, so there is a risk that processes (however light or heavy they might be) aren't being followed effectively regardless of the good intentions of the development team. Software Quality Assurance (SQA) is, in essence, ensuring that the developers are following the development process they are supposed to be following. If SQA is ineffective, it is possible (and in my experience likely) that some time spent on testing, design reviews, and other techniques to improve quality is also ineffective.

#8. No mechanism to capture technical and non-technical project lessons learned

There is no methodical effort to identify technical, process, and management problems encountered during the course of the project so that the causes of these problems can be corrected. As a result, mistakes are repeated in future projects.

3.2 Architecture

#9. No defined software architecture

There is no picture showing the system's software architecture. (Many such pictures might be useful depending upon the context – but often there is no picture at all.) Ill defined architectures often lead to poor designs and poor quality code.

#10. No message dictionary for embedded network

There is no list of the messages, payloads, timing, and other aspects of messages being sent on an embedded real time network such as CAN. As a result, there is no basis for analysis of real time network performance and optimization of message traffic.

#11. Poor modularity of code

The design has poorly chosen interfaces and poorly decomposed functionality, resulting in high coupling, poor cohesion, and overly long modules. In particular, interrupt service routines are often too big and mask interrupts for too long. The result is often increased risk of software defects due to increased complexity.

3.3 Design

#12. Design is skipped or is created after code is written

Developers create the design (usually in their heads) as they are writing the code instead of designing each module before that module is implemented. The design might be written down after code is written, but usually there is no written design. As a result, the structure of the implementation is messier than it ought to be.

#13. Flowcharts are used when statecharts would be more appropriate

Flowcharts are used to represent designs for functions that are inherently state-based or modal and would be better represented using a state machine design abstraction. Associated code usually has deeply nested, repetitive "if" condition clauses to determine what state the system is in, rather than having an explicit state variable used to control a case statement structure in the implementation. The result is code that is significantly more bug prone code and difficult to understand than code based on a state-machine based design.

#14. No real time schedule analysis

There is no methodical approach to real time scheduling. Typically an ad hoc approach to real time scheduling is used, frequently featuring conditional execution of some tasks depending upon system load. Testing rather than an analytic approach is used to ensure real time deadlines will be met. Often there is no sure way to know if worst case timing has been experienced during such testing, and there is risk that deadlines will be missed during system operation.

#15. No methodical approach to user interface design

The user interface does not follow established principles (e.g., [5]), making use of the product difficult or error-prone. The interface might not take into account the needs of users in different demographic groups (e.g., users who are colorblind, hearing impaired, or who have trouble with fine motor control).

3.4 Implementation

#16. Inconsistent coding style

Coding style varies dramatically across the code base, and often there is no written coding style guideline. Code comments vary significantly in frequency, level of detail, and type of content. This makes it more difficult to understand and maintain the code.

#17. Resources too full

Memory or CPU resources are overly full, leading to risk of missing real time deadlines and significantly increased development costs. An extreme example is zero bytes of program and data memory left over on a small processor. Significant developer time and energy can be spent squeezing software and data to fit, leaving less time to develop or refine functionality.

#18. Too much assembly language

Assembly language is used extensively when an adequate high level language compiler is available. Sometimes this is due to lack of big enough hardware resources to execute compiled code. But more often it is due to developer preference, reuse of previous project code, or a need to economize on purchasing development tools. Assembly language software is usually more expensive to develop and more bug-prone than high level language code.

#19. Too many global variables

Global variables are used instead of parameters for passing information among software modules. The result is often code that has poor modularity and is brittle to changes.

#20. Ignoring compiler warnings

Programs compile with ignored warnings and/or the compilers used do not have robust warning capability. A static analysis tool is not used to make up for poor compiler warning capabilities. The result can be that software defects which could have been caught by the compiler must be found via testing, or miss detection entirely. If assembly language is used extensively, it may contain the types of bugs that a good static analysis tool would have caught in a high level language.

#21. Inadequate concurrency management

Mutexes or other appropriate concurrent data access approaches aren't being used. This leads to potential race conditions and can result in tricky timing bugs.

#22. Use of home-made RTOS

An in-house developed RTOS is being used instead of an off-the-shelf operating system. While the result is sometimes technically excellent, this approach commits the company to maintaining RTOS development skills as a core competency, which may not be the best strategic use of limited resources.

3.5 Verification & Validation

#23. No peer reviews

Code, requirements, design and other documents are not subject to a methodical peer review, or undergo ineffective peer reviews. As a result, most bugs are found late in the development cycle when it is more expensive to fix them.

#24. No test plan

Testing is ad hoc, and not according to a defined plan. Typically there is no defined criterion for how much testing is enough. This can result in poor test coverage or an inconsistent depth of testing.

#25. No defect tracking

Defects and other issues are not being put into a bug tracking system. This can result in losing track of outstanding bugs and poor prioritization of bug-fixing activities.

#26. No stress testing

There is no specific stress testing to ensure that real time scheduling and other aspects of the design can handle worst case expected operating conditions. As a result, products may fail when used for demanding applications.

3.6 Dependability

#27. Insufficient consideration of reliability/availability

There is no defined dependability goal or approach for the system, especially with respect to software. In most cases there is no requirement that specifies what dependability means in the context of the application (e.g., is a crash and fast reboot OK, or is it a catastrophic event for typical customer?). As a result, the degree of dependability is not being actively managed.

#28. Insufficient consideration of security

There is no statement of requirements and intentional design approach for ensuring adequate security, especially for network-connected devices. The resulting system may be compromised, with unforeseen consequences.

#29. Insufficient consideration of safety

In some systems that have modest safety considerations, no safety analysis has been done. In systems that are more overtly safety critical (but for which there is no mandated safety certification), the safety approach falls short of recommended practices. The result is exposure to unforeseen legal liability and reputation loss.

#30. No or incorrect use of watchdog timers

Watch dog timers are turned off or are serviced in a way that defeats their intended role in the system. For example, a watchdog might be kicked by an interrupt service routine that is triggered by a timer regardless of the status of the rest of the software system. Systems with ineffective watchdog timers may not reset themselves after a software timing fault.

#31. Insufficient consideration of system reset approach

System resets might not ensure a safe state during reboots that occur when the system is already in operation, resulting in unsafe transient actuator commands.

#32. Neither run-time fault instrumentation nor error logs

There is no run-time instrumentation to record anomalous operating conditions, nor are there error logs to record events such as software crashes. This makes it difficult to diagnose problems in devices returned from customers.

#33. No software update plan

There is no plan for distributing patches or software updates, especially for systems which do not have continuous Internet access. This can be an especially significant problem if the security strategy ends up requiring regular patch deployment. Updating software may require technician visits, equipment replacement, or other expensive and inconvenient measures.

#34. No IP protection plan

There is no plan to protect the intellectual property of the product from code extraction, reverse engineering, or hardware/software cloning. (Protection strategies can be legal as well as technical.) As a result, competitors may find it excessively easy to successfully extract and sell products with exact software images or extracted proprietary software technology.

3.7 Project Management

#35. No version control

Sometimes source code is not under version control. More commonly, the source code is under version control but associated tools, libraries, and other support software components are not. As a result, it may be difficult or impossible to recreate and modify old software versions to fix bugs.

#36. No backward compatibility and version management plan

There is no plan for dealing with backward compatibility with old products, product migration, or installations with a mix of old and new product versions. The result may be incompatibilities with fielded equipment or a combinatorial explosion of multi-component compatibility testing scenarios necessary for system validation.

#37. Use of cheap tools (software components, etc.) instead of good ones

Developers have inadequate or substandard tools (for example, free demo compilers instead of paid-for full-featured compilers) because tool costs can't be reckoned against savings in developer time in the cost accounting system being used. As a result, developers spend significant time creating or modifying tools to avoid spending money on tool procurement.

#38. Schedule not taken seriously

The software development schedule is externally imposed on an arbitrary basis or otherwise not grounded in reality. As a result, developers may burn out or simply feel they have no stake in following development schedules.

#39. Presumption in project management that software is free

Project managers and/or customers (and sometimes developers) make decisions that presume software costs virtually nothing to develop or change. This is one contributing cause of requirements churn.

#40. Risk of problems with external tools and components

External tools, software components, and vendors are a critical part of the system development plan, and no strategy is in place to deal with unexpected bugs, personnel turnover, or business failure of partners and vendors.

#41. Disaster recovery not tested

Backups and disaster recovery plans may be in place but untested. Data loss can occur if backups are not being done properly.

3.8 People

#42. High turnover and developer overload

Developers have a high turnover rate. As a result, code quality and style varies. Lack of a robust paper trail makes it difficult to continue development. Often more important is that replacement developers may lack the domain experience necessary for understanding the details of system requirements.

#43. No training for managing outsource relationships

Engineers who are responsible for interacting with outsource partners do not have adequate time and skills to do so, especially for multi-cultural partnering. This can lead to significant ineffectiveness or even failure of such relationships.

4. ANALYSIS

4.1 Projects Don't Need To Be Perfect

It is important to point out that not every project needs to get everything on the preceding list perfect. Red flag areas were based on risk specific to a particular domain and product. A development team could totally ignore many or most items on the above list, so long as this didn't create a significant risk of product or project failure. For example, having the watchdog timer turned off is likely to be a red flag on unattended equipment with 24x7 operational requirements, but might not be a big deal for a non-critical hand-operated device that is power cycled before each use in normal operation.

In other words, items were red flags because they were significant risks in the context of that particular product, *not* because they were on a list of best practices that ignored application tradeoffs.

That having been said, identification of red flag issues was at the discretion of the reviewer with feedback from the developers being reviewed, and therefore somewhat subjective.

4.2 Back to Basics – But Less Than Expected

Perhaps surprisingly, there are only a very few risk areas that are almost universally accepted as embedded system core educational topics. Real time scheduling, watchdog timers, and concurrency management are likely to be on a typical embedded system educator's list of desirable technical topics for either a first or second course in the area. But most of the problem areas aren't like that. Many of the items are things omitted by typical embedded system texts and courses.

That doesn't mean core technical areas don't matter. I believe it is important to give embedded system designers a principled understanding of core engineering principles and underlying technology. But, these results suggest that informally trained embedded designers (who have, however, been formally schooled in a rigorous way of thinking about technical problems in general)

tend to find ways to fill in basic technical areas on their own, even if they haven't have technology-specific formal training. So, apparently, self-teaching with a book in one hand and a development board in the other works out pretty well for the basics. But it doesn't seem to work well enough beyond that.

Most risk areas seemed uncorrelated with developer backgrounds, but there were a few areas in which team members' formal educational background affected likely risk areas in ways that most people would probably expect. For example, developers with formal software training were more likely to use a version control system. Differences were not as dramatic as might be expected. In part, this is because non-software engineers are trained to follow a methodical development approach (such as creating written requirements and formal test plans) for non-software aspects of the system, and that approach carried over to software. However, developers without formal embedded training were more likely to have gaps in embedded-specific technology areas such as concurrency control and real time scheduling since they had not seen that material in an introductory programming text (and may well have been self-taught from an introductory embedded system text that lacked that information).

4.3 Knowing You Have A Problem

Most of the problem areas might be characterized as having the property that they are the result of a gap in the developer's understanding or a gap in the software process being used. In other words, developers didn't realize (or didn't have time) to look for some types of problems. Basic functionality for a desired system was usually there at the time of the design review. For example, everyone had figured out by the design review how to use an A/D converter well enough to get acceptable sample quality. And, developers were quite capable of finding and fixing obvious problems with functionality. The risks tended to come more from having a high probability of undetected subtle bugs, missing chances to have avoided big problems that surfaced late in the project, and missing chances to avoid project schedule or cost problems.

On the whole, it seems that smart, motivated developers can figure out most of the technology and fix most problems if they have a way to know what's broken. The biggest risks come when they don't realize something in their technology or development process is broken, or when they attempt ad hoc solutions to difficult problems because they don't know of the existence of more robust solution approaches. In other words, the biggest risks come from lack of a comprehensive education and correspondingly comprehensive process.

4.4 Weak Process Hurts

A surprise (to me at least) was that a significant fraction of the problem areas ended up being software process problems instead of technology problems. While most embedded system educators appear to be technologists at heart, the fact of the matter is that poor software process is a huge problem impeding the success of embedded system development efforts. (It's hard to have a good product with bad technology. But it's also hard to succeed with an ineffective development process.)

The lack of process content in most developer degree programs and educational support materials is deeply ingrained, and has various sources. But it is really hurting embedded developers, and is a critical skill set they must currently pick up once in industry.

4.5 Embedded Software Problems Are Only A Little Special

Many of the red flag areas would not be out of place in a list of enterprise computing project risks. Some software practices are good ideas regardless of the domain. However, the ways to mitigate risks are often different for embedded applications than for desktop applications.

4.6 Five Forebodes Failure

One of the informal observations made across the course of these reviews was that developer teams with exactly 5 primary contributors have the most spectacular project failures. Invariably these teams had previously completed a project with 3 or 4 members successfully, and increased the team size to tackle a more complex project without making any changes in their software process. But they failed with the new, 5-person team.

While this is an anecdotal result, projects that grow past 4 developers in size should seriously consider switching to a heavier weight software process (more paper, more formality, more methodical rigor). Smaller teams still seem to benefit from good process, but basically can get away with informality with less dramatic risks than larger teams (5 or more developers) working on more complex projects.

5. EDUCATIONAL IMPLICATIONS

5.1 Risk Areas And Formal Education

Embedded system software development is often performed by engineers with no formal training in that area. Rather, developers most often start as domain experts and pick up embedded software skills informally. As mentioned previously, the surprising part is not that such developers have gaps, but rather that they seem to do a pretty good job of filling in the gaps in basic technology areas all on their own. In other words, there isn't much difference in the risks areas identified in projects being performed by computer-trained embedded system engineers vs. non-computer trained domain experts.

The gaps that were identified are largely either in a few system integration technical areas or in the broader area of software development process. Most computer engineers (and even many computer scientists) receive little software process training. Thus, even embedded system engineers with formal computer-related degrees typically haven't see much formal educational material that would fill these gaps.

I believe that plugging the gaps in embedded system projects isn't likely to be solved by having more engineers take the usual sort of existing embedded system college courses. The problem is really that these topics typically aren't being taught to (nor packaged for learning by) embedded designers. This data suggests that it might be useful to rethink the core skills that should be taught in embedded system courses and included in texts.

5.2 Course Organization

Informal awareness of the types of topics that cause problems in industry embedded projects has been guiding graduate and undergraduate course content choice at my institution for a number of years. But, until I performed this study, I was operating on gut feel instead of data. As a result of this analysis, a

two-course embedded systems sequence has been updated to address most of the risk areas.

18-348 Embedded System Engineering is a course which is mostly taught to third- and fourth-year Electrical and Computer Engineering (ECE) undergraduate students. The syllabus superficially appears to be a rather typical introduction to microcontrollers course using 16-bit CPUs. But, portions of lectures, homework assignments, and lab assignments have been crafted to instill an understanding of the basics of methodical software process. For example, every assignment has carefully written, numbered requirements. Most assignments require documented peer reviews, designs, test plans, defined acceptance tests, and so on as an addition to the main, technology-centric objectives. These are lightweight approaches to instill awareness rather than rigorous treatment of process topics, largely because undergraduates lack the world-view and experience to appreciate and learn about process topics. But they are a start in the right direction. Technology topics from the list given earlier in this paper directly taught at this level are concrete, technical, or linked directly to implementation: #11 modularity, #12 design before implement, #13 statecharts, #14 real time scheduling, #16 coding style, #19 globals, #20 compiler warnings, #21 concurrency, and #30 watchdog timers.

18-649 Distributed Embedded Systems is taught to ECE Masters Degree students, usually in their first year of graduate school, and to fourth-year undergraduates as a follow-on to 18-348. The remaining risk areas not covered by 18-348 are covered in this course, with the coverage increasing over recent years as lectures have been modified to correspond to the risk area list. Course lectures are divided into three parts: one third cover software process and advanced embedded system technology, one third cover embedded networking and distributed systems, and one third cover dependable and critical system design. That is to say that most of the lecture content is technical (which is what most students are interested in learning when they sign up for the course), and only a few lectures are overtly process-centric. A semester-long course project is used to demonstrate the execution of process methods and (for students who are at a point that they are ready to learn the lesson) instill the value of having a lightweight but complete process for software development. A companion text [4] based on the experiences described in this paper also provides risk area information to the students.

5.3 Software Process Educational Philosophy

Most university students lack experience with the complexities of real systems, and have not yet encountered situations in which lack of good development process has caused a project failure. (Or, at least, they have not recognized that process issues may have contributed to the failure of a project they have been involved in.) Because of this lack of experience, it is often difficult to motivate computer engineering students to study these topics, which are more traditionally thought of as software engineering. Rather, students often want to focus solely on what they consider technical (non-process) matters in coursework.

The course sequence just discussed attempts to address this issue by creating exposure to key ideas at the undergraduate level and concentrating on a more direct treatment of process issues at the graduate student level via a hands-on course project experience.

Unlike most computer engineering course projects, the goal of the 18-649 course project is to learn good process. (To be sure, the project has to work! But fancy technical aspects are not the end goal.) The approach is to have an experience in using a reasonable process that incorporates risk areas identified in this paper. For this to work, the project has to be complex enough that most students are likely to fail or have a very difficult time if they ignore the process, but well structured enough that students are likely to succeed if they follow a good process.

To this end, the project in 18-649 is a simulation of a fine-grain distributed building elevator system. In this system every component (button, light, motor, etc.) has its own CPU that communicates with other CPUs via an embedded network. This project was chosen for many reasons, including giving students exposure to discrete event simulation and a taste of how distributed embedded control systems can be designed. But, most importantly, there are a number of quite subtle system-level behaviors that emerge from the interaction of component behaviors that are both representative of real elevator behavior and difficult to get right just by writing code. Moreover, there are sufficient technical subtleties in real elevator behavior to maintain student interest throughout the semester in their quest to create an elevator that actually delivers all the simulated passengers without triggering the simulated safety shutdown mechanism.

To accomplish the goal of emphasizing process, all students work from the same set of high level specifications and have the same acceptance tests. The elevators have to actually deliver passengers safely. But, most grading points are awarded for following a reasonable software process and delivering items such as requirements, architectural documents, design diagrams, unit tests, integration tests, and traceability tables. High level system requirements are modified mid-semester to require more sophisticated (and realistic) elevator behavior, and students must then update all aspects of their design package to correspond to the modified system while maintaining end-to-end traceability.

In keeping with the process spirit of the project, grading is an exercise in Software Quality Assurance (SQA). Graders award points based on whether the process was followed (e.g., did you submit design diagrams for every module?) rather than the technical excellence of the result (optimization is not judged). Based on experience, it seems helpful to give some technical feedback to students rather than having grading be purely SQA, but such feedback is given as technical advice without attaching grading points to non-process evaluations. A small number of bonus points is awarded to the *one* team who has the most efficient performance on acceptance tests. The same number of bonus points is available to *all* teams at the end of the semester who have excellent end-to-end design packages. The message most students get from this is that getting the process right is what they should emphasize.

By the end of 18-649, most students have experienced some bug or other difficulty that they themselves attribute to a process failure, and that experience is a main objective of running the project. (For example, they may encounter a bug that took a long time to track down because a design diagram had not been updated to correspond to modified code.) Informally, it appears that most students are skeptical about the need for rigorous software process at the start of the course. By the end of the project two-thirds to three-quarters of the students seem to have

gained an appreciation for the benefits of a process. (This estimate is taken from comments made during oral presentations where students are encouraged to be honest, and for the most part seem inclined to say what is on their mind.) The remaining students will often say they think the process content was a waste of time. But, at least they have experienced such an approach.

There are no doubt many other ways to approach teaching good software development process to engineering students who are not overtly interested in that. But, the point is that focusing a technically challenging project experience on methodical design methods rather than technical excellence of the outcome seems to help students understand and absorb process lessons in appropriate risk areas.

5.4 Related Work

It is no surprise that cross-disciplinary hardware/software education is required to educate embedded system designers ([12] and [13] discuss this, and this idea is also present in most curriculum designs cited below to some degree). There has been little formalized work on analyzing the needs of industry with regard to embedded systems. [2] is based in part on an analysis that takes into account industry surveys, and suggests that embedded system education should be more cross-disciplinary and more representative of embedded industry experiences. These are important observations and worthy goals. My results extend these observations by reporting problems that even experienced industry designers aren't able to resolve on a consistent basis.

A number of embedded system educators already emphasize some of the areas on the risk list, most typically the areas identified for inclusion in 18-348 as well as distributed system and dependability topics. Examples include [1], [7], [10], [11] and previous courses at Carnegie Mellon [3]. Other curriculum proposals include an explicit software engineering courses (e.g., [8]). No doubt there are some other degree programs that address most or all of these areas in one way or another (for example, Carnegie Mellon has an interdisciplinary Master of Science in Information Technology – Embedded Software Engineering degree [9] that requires both graduate embedded system technical courses and graduate software engineering courses). But such programs are not the norm. My belief is that software process concepts should be integrated throughout the embedded curriculum, and not just an optional or isolated course module.

Embedded system courses almost universally use hands-on project content as a way for students to get a feel for system integration issues. This certainly gives students experience in how difficult complex projects can be and gives them a chance to test their fundamental technical skills. However, I have found that even engineers who have been through a large number of industry design projects have gaps. Thus, I believe that merely experiencing a design project without guidance and reflection upon solid principles and these specific risk areas is not enough to fill these gaps. It is difficult to teach yourself ways to fix problems when you don't ever realize you got things wrong.

6. CONCLUSIONS

This paper identifies 43 areas that were identified as red flag risk areas across reviews of 90 industry embedded system projects in the past decade. The most striking aspect of the list is that, by and large, even self-trained developers are not at huge risk of missing the basics of embedded systems. Rather, most risks are either complex system integration skills (e.g., concurrency management) or software development process issues (e.g., requirements problems or inadequate test plans).

While many of the areas identified might not seem specific to embedded systems, they are the risk areas that are actually affecting real embedded projects. Embedded educators should take notice and take steps to ensure that future courses and degree programs address most of these areas, preferably in a way that teaches the skills most useful in an embedded system context.

7. ACKNOWLEDGMENTS

The author wishes to thank all the design teams that have been through the design review process with him, and the companies who have sponsored those reviews. It's never fun having an outsider come in to tell you all the mistakes you've made, and I appreciate the openness and willingness to discuss things shown by so many developers over the years.

8. REFERENCES

[1] Capsi, P., et al., 2005, "Guidelines for a graduate curriculum on embedded software and systems," *ACM TECS*, vol. 4, no. 3, pp. 587-611, August 2005.

[2] Grimheden, G.; Torngren, M., 2005, "What is embedded systems and how should it be taught?---results from a didactic analysis," *ACM TECS*, vol. 4, no. 3, pp. 633-651, August 2005.

[3] Koopman, P., et al., 2005, "Undergraduate embedded system education at Carnegie Mellon," *ACM TECS*, vol. 4, no. 3, pp. 500-528, August 2005.

[4] Koopman, P., 2010. *Better Embedded System Software*, Drumnadrochit Press, Wilmington.

[5] Nielsen, J., 1993. *Usability Engineering*, AP Professional, Boston.

[6] Paulk, Mark C., et al., 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison Wesley, Boston.

[7] Sangiovanni-Vincentelli, A.; Pinto, A., 2005, "An overview of embedded system design education at Berkeley," *ACM TECS*, vol. 4, no. 3, pp. 472-499, August 2005.

[8] Seviora, R., 2005, "A curriculum for embedded system engineering," *ACM TECS*, vol. 4, no. 3, pp. 569-586, Aug. 2005.

[9] MSIT-ESE program web page, Carnegie Mellon University, accessed July 30, 2010. http://mse.isri.cmu.edu/software-engineering/web1-Programs/MSIT-ESE/index.html

[10] Ricks, K., Jackson, D., & Stapleton, W., "Incorporating embedded programming skills into an ECE curriculum," *ACM SIGBED Review*, vol. 4 no. 1, pp. 17-26, Jan 2007.

[11] Tsao, S., Huang, T., & King, C., "The development and deployment of embedded software curricula in Taiwan," *ACM SIGBED Review*, vol. 4, no. 1, pp. 64-72, Jan 2007.

[12] Henzinger, T. & Sifakis, J., "The discipline of Embedded Systems Design," *Computer*, pp. 32-40, Oct. 2007.

[13] Lavi, J. et al., "Engineering of Computer-Based Systems – a proposed curriculum for a degree program at the bachelor level," *ECBS98 Conference*, pp. 1-8, March 1998.