DOWNLOAD PDF DESIGN FACTORS FOR SAFETY-CRITICAL SOFTWARE

Chapter 1 : CiteSeerX â€” Design Factors for Safety-Critical Software

Download Citation on ResearchGate | Design Factors for Safety-Critical Software | This report is the fourth of a series of reports prepared for the Nuclear Regulatory Commission Office of Nuclear.

LinkedIn Recent survey data indicates that static code analysis, peer review, and basic coding standards are being neglected in the development of safety-critical connected embedded devices. Safety and security are both critical to the development and deployment of connected systems, but while the two overlap, they are also distinct. Can you define safety and security in an embedded systems context? Within an embedded systems context, safety and security become broad issues, crossing disciplines from mechanical to electronics to software. As contrasted with other technology environments such as cloud computing and mobile apps, an organization developing an embedded device must incorporate disparate engineering disciplines into the process. Thus, safety and security must first be considered at the higher system level, somewhat independent of engineering discipline. It is critical during early requirements analysis and architectural design to incorporate security and safety expertise into the process. There will be overlap in design of systems that are both safe and secure. Perhaps the most obvious overlap is in the concept of reliability. For this reason, we strongly advocate the use of best practices that increase reliability. Some of these are universal across engineering disciplines, such as peer design reviews. Some are specific to an engineering discipline, such as signal integrity analysis or the use of coding standards or static analysis. Beyond this, there are indeed distinctions between safety and security. Improving reliability is not enough to ensure adequate security. A secure design must consider how hackers think, how they might try to enter a device, how expensive it would be for them to do so, etc. Protecting data-at-rest and data-in-motion via encryption can be very important, as well as authenticating over-the-air updates. Providing anti-tamper mechanical features can be critical, too. And, one must consider the motivations and resources of the hacker. If the device you are designing is of interest to just a few, security levels may not need to be as high as if your device has super-secret or financially valuable information that might be of interest to a large group or nation-state. Barr Group recently conducted a survey surrounding safety and security for connected embedded systems, which had some astonishing results. Can you share some of the results around the lack of best practices being used in safety-critical, connected system development? Our recent Embedded Systems Safety and Security Survey did uncover concerning trends around best practices for embedded software development. Of over 1, qualified respondents, we did an analysis of those that self-identified as currently involved in the design of embedded devices that could kill or injure if the device malfunctioned â€” so-called safety-critical devices. Approximately 28 percent are designing these safety-critical devices and it should be a foregone conclusion that well-known fault-reducing best practices in the development of embedded software â€” such as the use of code reviews, coding standards, and static analysis â€” should be universally used in these device designs. Studies have shown that they work. They are generally well-known and supported by a variety of third-party tools. And yet, we are not seeing universal support for them. Perhaps it is time-to-market pressure. Or limited development budgets. Whatever the reason, it needs to change, and I encourage everyone to review the complete survey results which are freely available as a report on our website. In your opinion, what is more of a threat to safety-critical connected systems: Hackers compromising a network and manipulating devices, or errors during the development process that make their way through test and into shipping products? Both are significant concerns. But there is a difference, at least philosophically. Security is an arms race. Hackers always will be coming up with new and unique ways to compromise embedded devices. So, designers always must be vigilant and build an appropriate level of security measures into their devices, considering how they are used, the value of the device and its information, and the resources of the potential hackers. That said, the biggest concern to me is the lack of recognition and use of best practices, as noted above. If design teams and their management are not willing to incorporate even lightweight best practices into their development, how can we as an industry be all that surprised at the safety and security issues that we see in Internet of Things IoT and embedded devices?

Chapter 2 : Safety-critical software development surprisingly short on standards, analysis, and review

This report is the fourth of a series of reports prepared for the Nuclear Regulatory Commission Office of Nuclear Reactor Regulation, and provides the summary and conclusion for this task. It is widely believed in the software engineering community that almost anything can affect the ability of.

Open-standards-based, real-time operating systems enable reuse of legacy code, and facilitate the development of more complex systems without sacrificing certifiability or real-time performance. Over the past 30 years, several major trends have impacted commercial avionics. Digital systems have displaced analog devices and have evolved to automate increasingly complex functions. These functions range from non-critical cabin entertainment features to safety-critical automated flight controls. These avionics functions have become increasingly software-intensive, while at the same time becoming increasingly safety-critical. Due to many factors e. For example, non-critical cabin entertainment software may be hosted on the same CPU as the highly-critical flight control software. Due to the cost of developing avionics software, avionics manufacturers havebeen driven to leverage reusable software. Ideally, such software is developed once and then reused many times. For example, a math library could be reused in hundreds of applications on dozens of different aircraft. As a result of these trends, five key factors now play significant roles in the development of avionics software: In the remainder of this article, we discuss these factors in more detail and how they enable avionics developers to create safety-critical software efficiently. In the early s, civil aviation regulatory authorities e. FAA, EASA recognized that avionics equipment was becoming increasingly complex, software-intensive and safety-critical. For example, relatively simple flight control functions, once performed by analog devices, were being replaced by more sophisticated digital systems. In response, regulatory authorities and commercial avionics manufacturers co-authored guidelines for developing avionics software i. These guidelines were intended to ensure that avionics developers employed a degree of process rigor during software development and verification commensurate with the criticality of the function being performed. For example, modern flight control software typically requires Level A design assurance, which imposes the most stringent process rigor, since a fault therein could be catastrophic i. Conversely, a cabin entertainment system would typically require Level E design assurance and the least stringent process rigor, since a fault therein would have no effect on the safe operation of the aircraft, just a plane full of unhappy passengers. As noted, not all software in modern avionics has the same level of safety-criticality. Further, as the level of process rigor increases, the cost to develop and verify software increases. For example, software developed to Level A design assurance can easily cost 5 to 10 times more than software developed with a relatively low degree of process rigor, say Levels D or E. Consequently, manufacturers try to minimize the amount of software categorized at higher levels Designing safety-critical avionics software using open standards of safety criticality. And, due to factors that drive increasing levels of integration, they often host numerous software functions â€” with varying levels of criticality â€” on a single CPU. However, this integration creates a special challenge. Specifically, how does one prevent these different software functions from interfering with one another? Solving this challenge is especially important when mixing high- and low-criticality software on the same CPU. For example if the cabin entertainment system corrupts the flight control function aircraft attitude data i. Similarly, the cabin entertainment system could get stuck in an infinite loop, thereby hijacking the CPU and denying the flight controls time to execute. In both cases, a catastrophic event could occur. In recent years, a new class of partitioned real time operating systems p-RTOS , such as LynuxWorks LynxOS, has been developed that provide the ironclad guarantees of non-interference and deterministic behavior required by safety-critical systems. A p-RTOS allows the software developer to create brick wall partitions for each function. This partitioning prevents one function from interfering with another, in terms of time i. To enforce time partitioning, each software function is allocated a strict budget of CPU time. If a function attempts to overrun its CPU budget, the timer interrupt fires, the p-RTOS takes control of the CPU, preempts the offending function and allows the next function in the schedule to run. To enforce space partitioning, each software function is allocated a strict quota of its own memory e. RAM and stack space. Resource partitioning is

achieved in a similar manner, wherein each function is granted explicit access i. With proper use of these capabilities, developers have a high degree of confidence that the software they develop will be free from defects wherein a fault in one function could interfere with the intended, correct and safe operation of another. Even with a partitioned RTOS, software developers must adhere to sound design principles in order to avoid introducing subtle design flaws that could compromise partitioning and lead to unwanted interactions between partitions. Regarding design flaws, partitioning provides the software developer with a means of erecting barriers that prevent software in one partition from interfering with software in another partition. However, in many cases, software applications in different partitions need a means of communicating or interacting. But, if done incorrectly, the developer can create holes in the partitions that may lead to problems. A control coupling exists when software application X can cause application Y to perform an action on the behalf of X. Similarly, a data coupling exists when X can send data to Y, presumably so that Y will read and use the X data. As long as X works properly, Y likely will too. However, if X experiences a fault, which causes it to make an erroneous request of Y or to send Y erroneous data, then the fault in X propagates to Y. These scenarios are especially problematic if X has a low-level design assurance say Level E and Y has high-level design assurance say Level A. In limited cases, unit-level tests may be needed to drive hard to reach paths in a given piece of code or white box. For example, Level C design assurance requires statement coverage, such that every statement has been executed at least once. An outcome of this requirement is that every path through the code must be exercised by at least one test. Essentially, these coverage requirements take a more sophisticated view of the paths that exist in the conditions and decisions of the code. A decision is a Boolean expression composed of one or more conditions and zero or more Boolean operators. For Level B coverage, testing must drive this decision to take all possible outcomes or paths at least once for this decision, there are two outcomes: For Level A coverage, testing would have to drive this decision to take all possible outcomes as for Level B , but in addition, Level A coverage requires that each condition take all possible outcomes or paths at least once for both these conditions, there are two outcomes: Further, each condition must be shown to independently impact the decision outcome, while holding the other condition s fixed in a sense, yet more paths. Consequently, one must take care when creating paths in the software. If the code is too complex, it becomes difficult to understand and errors may not be caught by either review or test. Further, achieving the appropriate level of coverage for complex code can become very difficult, time-consuming and costly. Note that Level D imposes no structural coverage requirement. Level E imposes no testing requirement. For example, LynxOS is conformant with both. Such standards bring two distinct advantages First, they embody the collected wisdom of many RTOS experts e. This experience base ensures that the design of a standards-based RTOS rests on very solid ground. When designing safety-critical software, the more solid the ground underneath you i. Second, an open-standards-based RTOS facilitates reuse of proven legacy code, often from external sources, and enables the evelopment of more complex software without sacrificing certifiability or real-time performance. This approach is especially attractive in markets such as commercial avionics, which require costly certification activities. For example, if the manufacturer can incur the cost of developing a software component once, along with associated certification evidence, and it can then reuse that software and evidence at a drastically reduced cost, it gains an advantage over its competitors who lack that capability. ABOUT US Every day, millions of people worldwide are touched by products that rely on Lynx Software Technologies softwareâ€”from Internet and phone communications, to airline flight-control systems, office automation and medical devices. Lynx Software Technologies software provides the hidden intelligence that empowers, protects and secures our modern world.

Chapter 3 : Designing Safety-Critical Avionics Software Using Open Standards - Lynx Software Technolog

CiteSeerX - Document Details (Isaac Councill, Lee Giles, Pradeep Teregowda): This report is the fourth of a series of reports prepared for the Nuclear Regulatory Commission Office of Nuclear Reactor Regulation, and provides the summary and conclusion for this task.

Subscribe now This is a guest blog post written by Dr. Work smarter, better, and faster with weekly tips and how-tos. With a heavy emphasis on design review and testing, these methods are naturally quite slow. But, with the ever-increasing pressure to move faster, even these traditionally highly-regulated domains have been undergoing a quiet transformation to agile, cloud-based methodologies. Regardless of your development model or toolset, there are some key issues that set the development of safety critical software apart from the majority of enterprise- or consumer-facing applications. In other words, these devices are required to pass a regulatory audit prior to being placed into the market. The audit entails presenting evidence of rigorous planning and documentation, i. How to meet this challenge: All of the above means that companies developing safety-critical software have to commit a lot of resources and time to adhere to the regulations in order to pass the audits and place their products on the market. This can be challenging, especially for smaller companies and those new to the requirements for developing software for regulated applications. Implementation of the required processes is considerably easier when appropriate software tools are used that help automate various regulatory aspects and that integrate with the software development environment. In order to pass regulatory audits associated with product approval, detailed specifications have to be recorded, including any changes to these specifications as well as their approval. Regulatory audits require a documented history trail, for example, a history of all the changes that were requested, approved and implemented throughout the system or device development lifecycle. This is doubly the case when there are changes in the specifications or if teams are using different development approaches. For example, a medical device manufacturer specifies the overall requirements for the device. It then communicates the requirements of the sub-systems to the allocated software, electrical, and mechanical development teams. These teams have to work together to communicate any requirement dependencies, risks, and changes that affect the other sub-systems to enable the effective integration of the sub-systems, as well as and mitigate any foreseeable risks of the sub-systems to ensure the effectiveness and safety of the entire device. A risk assessment includes the determination of key hazards, risks, failure modes, and mitigations, for software where the device risks have to be linked to software items. International standards like ISO provide guidance about medical device risk management and how risks have to be assessed and mitigated within the software lifecycle. Each mitigation action must be tested to verify that the mitigation works as intended and that the new mitigation action which can often be another software item does not introduce a new risk. If a new risk is introduced, it needs to be assessed and mitigated following the same risk management process. All risk management related activities should be documented and follow a documented plan that was set up prior to the risk assessment being performed. Regulatory audits often require full traceability of risk management activities where the auditor checks that everything that has been built was analysed for risks and that each risk has been mitigated. This entails establishing documented verification plans that outline the overall strategy and approach to the testing of the software systems. Test results are typically also formally recorded, reviewed and signed off. Regulatory audits require full traceability of software development activities whereby checks are carried out that confirm that everything that has been specified was built and that everything that was built has also been appropriately verified. Links between risks, mitigation actions, and their verification Solution overview Most of these requirements pose serious challenges for safety-critical software systems developers. A lot of regulatory affairs tasks are still managed manually today â€” this may be highly error-prone and cause downstream issues in the construction of technical files for product registration and in regulatory audits. This provides the benefits of assisting in team collaboration, product definition, and in providing the risk management and traceability that is crucial for safety-critical domain software companies.

Chapter 4 : Design Factors for Safety-Critical Software - CORE

One of the most important aspects of developing safety-critical software is determining which requirements and standards are going to be followed. Depending on the newness of either your product or the intended market, you may need to get outside help to determine what standards need to be met.

Release on by , this book has page count that enfold important information with easy reading experience. Release on by , this book has page count that contain valuable information with lovely reading experience. Release on by , this book has page count that attach constructive information with easy reading experience. Release on by , this book has page count that include essential information with lovely reading experience. Release on by , this book has page count that attach important information with easy reading experience. Release on by , this book has page count that enclose important information with easy reading structure. Release on by , this book has 72 page count that enfold useful information with easy reading experience. Release on by , this book has page count that include useful information with easy reading experience. Release on by , this book has page count that include constructive information with lovely reading experience. Release on by , this book has page count that consist of important information with lovely reading experience. Release on by , this book has page count that contain essential information with easy reading experience. Release on by , this book has page count that attach essential information with lovely reading experience. Release on by , this book has page count that enfold constructive information with lovely reading experience. Release on by , this book has page count that attach important information with easy reading structure. Release on by , this book has page count that consist of constructive information with lovely reading experience. Release on by , this book has page count that contain constructive information with easy reading structure. Release on by , this book has page count that include helpful information with easy reading structure. Release on by , this book has page count that consist of essential information with easy reading structure. Release on by , this book has page count that enfold helpful information with easy reading structure. Release on by , this book has page count that contain important information with lovely reading experience.

Chapter 5 : Software Safety: Examples, Definitions, Standards, Techniques

How to design and test safety critical software systems. factors t hat distinguish legacy software testing techniques. from safet y engineering aimed at safety-critical software.

Examples, Definitions, Standards, Techniques tdh06u Definitions: Not all of these can be rectified by the programmers. The three types are: Software logic errors, Software support errors, Hardware failures Software logic errors are often a result of the programmer making errors in the coding, whether it is simply a mistake on their part, or an incorrect set of requirements they are following. In addition, it may have been a mistake made in the design phase which follows through into the implementation of the system. Software support errors are linked to the software being used to create the program. Perhaps these errors are from the compiler, an external library being used, or even the programming language. Due to the fact that software needs hardware to run on, software safety also has to take hardware safety into account. Therefore, hardware failure is an important aspect to consider in safety critical systems. Even if the software is completely bug free and perfect, if the hardware malfunctions in some way, perhaps memory corruption or interrupt failure, this can also lead to software malfunction. Within a system, part of software safety is identifying the safety critical areas and making sure that these areas are covered adequately for the use of the system. Software safety can be applicable to a variety of systems, but most often it is associated with system-critical systems. Any errors in these types of systems can have a profound negative impact. For example, in nuclear reactor systems, airplane computers or life control systems the need for safety is clear. Obviously, for rockets, safety is critical to ensure there a minimised risk of an accident. This automated steering system is able to control the car without the driver turning the wheel at all. Obviously, the need for safety is vital in this situation. There are many standards on software safety, as shown here. Unfortunately these are not available to read for free. One of the major standards for writing safe software is IEC unfortunately the standard itself is not available for free. Following this standard ensures a high amount of safety consideration and checking. When following this standard, risk analysis is performed and potential risks are given SIL Software Integrity Level ratings from 1 to 4, on a logarithmic scale, where 1 requires least risk reduction a factor of and 4 requires the greatest risk reduction or more times. These SIL ratings are used to determine which safety methods are performed on each component. Obviously not every component of the software is likely to be safety critical so this is taken into consideration by the IEC standard, resulting in non-critical modules not having any safety requirements. The NASA software safety standard has a clear definition of the purpose of software safety. In another report about software safety , in terms of an electronic mining system, software safety is considered in detail in every part of the development process. At first, a risk analysis and assessment is conducted, then each risk is categorised and any methods than can be used to reduce these risks are considered and possibly implemented. Safety requirements are often set in the specification of software for which the safety of the software is important. These specify the methods to be used to either prevent faults or errors, or how to detect and appropriately handle them. This is done as early as possible in the systems development life cycle. Its purpose is to identify as many potential hazards or risks in the system as possible. These can then be classified and analysed. Once this stage is complete, the findings are used to create safety requirements for the project to ensure the system is covering all the potential hazards that have been thought of. Obviously this is not going to catch every single hazard in the software, but it gives a good basis and saves a lot of time in the later stages by avoiding having to find and correct hazards in the software. Each hazard found can be given a way of either avoiding the hazard or minimising the potential risk. This is similar to Preliminary Hazard Analysis, except each identified hazard is now identified as having a number of possible states from which the hazard state is linked to. These can then be carefully analysed in order to ensure hazard risk from these states is minimised. Safety requirements are developed during hazard testing. Fault response times are found from these tests. This is the amount of time during which a fault can be detected and avoided before it becomes a more serious hazard. Many of these tests are performed in a simulated environment, but in order to ensure the simulation mirrors reality, there should be a test with the actual system. Software Safety Requirements Review: This is the

process of going over the requirements which have been set and formulated throughout the earlier stages in analysis to ensure completeness and accuracy. This is done in order to check all the necessary requirements are stated and that they are accurate and complete, before implementation actually begins. This is done to reduce the likelihood of changing them during the implementation stage as this is much more costly than identifying and changing them in this early stage. During the Software Design phase: A criticality analysis can be performed to categorise different aspects of the system with regards to safety. There are four types of safety level, labelled C0 to C3. C0 is assigned to those modules or components with no safety-critical aspects. C1 modules are those modules with only a relationship to safety via interfacing with other modules which are safety linked. C2 modules are safety related, in that if they fail alone, there will be no dangerous faults, but in combination with another module, it could cause a danger. C3 are the most safety critical. If just one C3 module fails, it could cause a dangerous fault. During the analysis phase, a fault tree may have been drawn. This shows the relationship between hazardous states and states leading up to these. During this stage of the life cycle, the fault tree is modified by adding more detail. The existing tree would have been at a high level, but the new tree has more detail of the software architecture which has been designed during this phase. It is checked to make sure no changes to the overall tree have been made, but simply the high level nodes have been expanded deeper, showing a more in-depth overview of the system. This tree can then be used, again, to identify software hazards and how this time more precisely they may occur in the software. It is also used to assign criticality to different modules, as it can be seen exactly where hazards may occur and which modules have different effects on the potential occurrence of the hazard. Its main purpose is to identify any structural weaknesses in the design and to be able to rectify these before implementation begins. FMEA cannot be performed before this stage, as a lot of information is needed about the system before this process can be undertaken. For example, a high level design of the system, the programming language to be used and the compiler that will be used. In FMEA, there are a range of failure modes that are possible to occur within software. These are tested against the design and if it is possible to reach any of these states then it means there is an error in either the requirements or the transfer of requirements into the design. Therefore, the requirements may need to be altered, and the design changed to reflect this. During the late design and coding phase: Detailed Fault Tree Analysis: Once the coding has been started, the team now have knowledge of exactly how the system will be implemented. Due to this, the fault tree can be further extended to cover more low level components. This will, as before, increase the number of potential hazard states detected, and allow for further precautions to minimise the occurrence of hazards in the system. This technique involves looking at each variable in the system and examining the effects of any potential failures. It is very time consuming, and therefore only used in components where safety is key. This technique is sometimes entirely skipped, as, for many systems, the time spent performing detailed FMEA is not feasible. This might be the case, where, in the case of system failure, there would not be a huge loss. For detailed software FMEA , each variable has a certain number of failure modes. There are three types of variable in terms of the tests in FMEA, these are: Each of these types has a different set of failure modes. Booleans only have two, true when it should be false and false when it should be true. Obviously enumerated types have a different number of failure modes depending on the number of options. A high number of options results in the number of failure modes being very large. Since analog variables are difficult to measure, there are simply two types of failure: There are also additional sub-techniques within detailed software FMEA which include calculating failure modes for program logic and variable mapping. More information on these can be found on page 10 of this document. In addition to the theoretical techniques mentioned, a commonly used technique in systems involving safety is defensive programming. This involves working with the mentality that anything that has even a very slight possibility of being misused or malfunctioning must be protected against. This generally results in a system that is of higher security and safety of one that was written without defensive programming. Software verification and testing: Testing against safety requirements: The requirements from the start of the project are tested against in many ways. Generally this is done by testing using a simulator, testing using bench equipment, and finally testing using the real system. The tests aim to show that, in all the identified ways, a software failure is protected against. However, even if the system passes all the tests in the test plan, it does

not prove it will not malfunction. The aim of this testing is simply to maximise safety in the system as far as is feasibly possible.

Chapter 6 : 4 challenges in developing safety-critical software (and what to do about them) - Atlassian Blog

Safety-critical software systems are developed within a risk-based framework: the regulatory framework requires the assessment and mitigation of all reasonably foreseeable risks prior to placing the products on the market. A risk assessment includes the determination of key hazards, risks, failure modes, and mitigations, for software where the.

For safety critical systems, the key questions are how do I know my design will do what I say and how do I know it does not have some unforeseen latent error? FPGA based state machines also sometimes referred to Finite State Machines FSM offer a unique advantage over a traditional CPU design in that safety critical tasks can be truly isolated and run on hardware tailored to the tasks. A CPU is a state machine. Every CPU architecture is a general purpose state machine that accepts instructions aka assembly, machine code and performs IO or manipulates data based on those instructions. The advances in speed, power, and price in CPUs is breathtaking. If you have any kind of hardware IO, you share CPU cycles with an interrupt handler that cannot be deterministically modeled over time. Interrupts happen when they happen and your safety critical code shares the same processor and memory. The abstraction from the machine code provided by a high level language is just that, an abstraction from what is really happening at the low level. Yes you can examine the machine code produced by the compiler but that somewhat negates the value of a high level language. Decreasing design complexity increases confidence in correctness The CPU and memory are a shared resource among all software routines executing on a processor. If one swimmer in the pool exhibits poor judgement, all swimmers are exposed to the same water. Yes you can have various memory protection schemes that are well tested and robust, but even those have their own cost in time, complexity, and money. The algorithm can be truly isolated from any non-essential IO. Your algorithm gets its own swimming pool. You can tailor your algorithm to your problem. This leads to greater clarity of design. It is much easier to prove a positive in testing and documentation than to disprove a negative. Once timing is closed on a particular block, it can be moved unchanged to other FPGA fabrics Although funding concerns have prevented this project reaching market, we had a recent medical device project that served as the inspiration and proving ground for this approach. The customer was time constrained to build a prototype of a complex medical device that integrated several OEM vendor modules from different sources into a single device. They wanted to base the prototype on an existing mature medical device that had already been thoroughly tested. The challenge was the existing product only had one available serial port for this expansion but needed to communicate with a half dozen additional sensors that all wanted to communicate via serial. The biggest advantage to this approach is fault isolation. Each sensor had its own dedicated data path that shared no resources with other sensors. The parsed data of interest was then re-packetized into a single serial data stream of a common format. In this way we interleaved serial data in many formats into a common format single stream. FSM From A Sparx Medical Project The reverse path to command all the different sensors from a single serial stream was similar but in the opposite direction. The command stream was de-serialized and the data of interest was then routed to unique command handlers based upon a pattern matching algorithm. Each sensor had its own command handler that took the data of interest and re-packetized it into the native sensor format. With this design, we could isolate all our algorithms from each other. Plus we had the ability to rapidly prototype our soft design for each sensor on a totally different FPGA platform. But that is another blog post.

Chapter 7 : Match Book For Safety Critical Software Design - calendrierdelascience.com

Successful creation of safety critical software is dependent on many factors. Figure 4 illustrates there is a hierarchy of constraint necessary within the development organization.