SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

1: Embedded software - Wikipedia

Results of a study of Air Force procedures for formulating and communicating software requirements and their effects on software acquisition for embedded computers. Conceptual models are used to describe software acquisition management activities of the Air Force and software development activities.

It is programmable or non- programmable depending on the application. An Embedded system is defined as a way of working, organizing, performing single or multiple tasks according to a set of rules. In an embedded system, all the units assemble and work together according to the program. Examples of embedded systems include numerous products such as microwave ovens, washing machine, printers, automobiles, cameras, etc. These systems use microprocessors, microcontrollers as well as processors like DSPs. This article gives an overview of what is an embedded system and types of embedded system. The important characteristics of an embedded systems are speed, size, power, reliability, accuracy, adaptability. Therefore, when the embedded system performs the operations at high speed, then it can be used for real -time applications. The Size of the system and power consumption should be very low, then the system can be easily adaptable for different situations. What is an embedded system? An Embedded system is a combination of computer hardware and software. As with any electronic system, this system requires a hardware platform and that is built with a microprocessor or microcontroller. Embedded System Embedded system software is written in a high-level language, and then compiled to achieve a specific function within a non-volatile memory in the hardware. Embedded system software is designed to keep in view of three limits. They are availability of system memory and processor speed. When the system runs endlessly, there is a need to limit the power dissipation for events like run, stop and wake up. Types of Embedded Systems Embedded systems can be classified into different types based on performance, functional requirements and performance of the microcontroller. Types of Embedded systems Embedded systems are classified into four categories based on their performance and functional requirements: Stand alone embedded systems Networked embedded systems Mobile embedded systems Embedded Systems are classified into three types based on the performance of the microcontroller such as Small scale embedded systems Sophisticated embedded systems Stand Alone Embedded Systems Stand alone embedded systems do not require a host system like a computer, it works by itself. It takes the input from the input ports either analog or digital and processes, calculates and converts the data and gives the resulting data through the connected device-Which either controls, drives and displays the connected devices. Examples for the stand alone embedded systems are mp3 players, digital cameras, video game consoles, microwave ovens and temperature measurement systems. These types of embedded systems follow the time deadlines for completion of a task. Real time embedded systems are classified into two types such as soft and hard real time systems. Networked Embedded Systems These types of embedded systems are related to a network to access the resources. The connection can be any wired or wireless. This type of embedded system is the fastest growing area in embedded system applications. The embedded web server is a type of system wherein all embedded devices are connected to a web server and accessed and controlled by a web browser. The basic limitation of these devices is the other resources and limitation of memory. These types of embedded systems have both hardware and software complexities. Applications of Embedded Systems: Embedded systems are used in different applications like automobiles, telecommunications, smart cards, missiles, satellites, computer networking and digital consumer electronics.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

2: Carlson Software - System Requirements

Note: Citations are based on reference standards. However, formatting rules can vary widely between applications and fields of interest or study. The specific requirements or preferences of your reviewing publisher, classroom teacher, institution or organization should be applied.

Unlike the errors we looked at yesterday, you could make the case that the overall system development process here was effective in the sense that the problems were caught before the system was deployed. Lutz is interested in tracking down why so many safety-critical errors are found so late in the process though. This approach allows classification not only of the documented software error called the program fault , but also of the earlier human error the root cause, e. Having thus created an error profile of safety-related software errors, the paper concludes with a set of six guidelines to help prevent them. Functionality is present, but incorrect. Operating faults â€" a required but omitted operation in the software. Conditional faults â€" nearly always an erroneous value on a condition or limit. Safety-related interface faults are associated overwhelmingly with communications errors between a development team and others often between software developers and systems engineers , rather than communications errors within teams. The primary cause of the behavioural, operating, and conditional faults is errors in recognising understanding the requirements. For safety-related interface faults, the most common complexity control flaw is interfaces not adequately identified or understood. Related to process, lack of documentation of hardware behaviour, poor communication between hardware and software teams, and undocumented or communicated interface specifications. In summary, the software developers are working with incomplete information and erroneous assumptions about the environment in which the software will operate. Not all of these things are perfectly knowable up front though: Leveson listed a set of common assumptions that are often false for control systems, resulting in software errors. For functional faults, the most common cause is requirements which have not been identified. Missing requirements are involved in nearly half the safety-related errors that involve recognising requirements. Imprecise or unsystematic specifications were more than twice as likely to be associated with safety-related functional faults. These results suggest that the sources of safety-related software errors lie farther back in the software development process â€" in inadequate requirements â€" whereas the sources of non-safety-related errors more commonly involve inadequacies in the design phase. Six recommendations for reducing safety-related software errors Since safety-related software errors tend to be produced by different mechanisms than non-safety-related errors, we should be able to improve system safety by targeting safety-related error causes. Lutz presents six guidelines: Focus on the interfaces between the software and the system in analyzing the problem domain, since these interfaces are a major source of safety-related software errors. Identify safety-critical hazards early in the requirements analysis. Use formal specification techniques in addition to natural-language software requirements specifications. As requirements evolve, communicate the changes to the development and test teams:

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

3: Best Requirements Management Software | Reviews of the Most Popular Systems

the acquisition and support of software for computers "embedded in" (i.e., an integral part of) major defense systems, such as aircraft or missile weapon systems. The mamagement of software requirements for such computers is a major problem.

Of course you want the specification to be correct. No one writes a specification that they know is incorrect. Unambiguous â€" An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities â€" fix them. Complete â€" A simple judge of this is that is should be all that is needed by the software designers to create the software. Consistent â€" The SRS should be consistent within itself and consistent to its reference documents. Ranked for Importance â€" Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS. Traceable â€" Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement? Very often we find that companies do not understand the difference between a System specification and a Software specification. What is the difference between a System Specification and a Software Specification? Very often we find that companies do not understand the difference between a System specification and a Software Specification. Important issues are not defined up front and Mechanical, Electronic and Software designers do not really know what their requirements are. The following is a high level list of requirements that should be addressed in a System Specification: In many systems we work on, some functionality is performed in hardware and some in software. It is the job of the System specification to define the full functionality and like the performance requirements, to set in motion the trade-offs and preliminary design studies to allocate these functions to the different disciplines mechanical, electrical, software. Another function of the System specification is to specify performance. Some portion of that repeatability specification will belong to the mechanical hardware, some to the servo amplifier and electronics and some to the software. It is the job of the System specification to provide that requirement and to set in motion the partitioning between mechanical hardware, electronics, and software. Very often the System specification will leave this partitioning until later when you learn more about the system and certain factors are traded off For example, if we do this in software we would need to run the processor clock at 40 mHz. However, if we did this function in hardware, we could run the processor clock at 12 mHz. I think it is useful to say that explicitly. This is done primarily because most of the complexity is in the software. When the hardware is used to meet a functional requirement, it often is something that the software wants to be well documented. Very often, the software is called upon to meet the system requirement with the hardware you have. Very often, there is not a systems department to drive the project and the software engineers become the systems engineers. For small projects, this is workable even if not ideal. In this case, the specification should make clear which requirements are software, which are hardware, and which are mechanical. In short, the SRS should not include any design requirements. However, this is a difficult discipline. For example, because of the partitioning and the particular RTOS you are using, and the particular hardware you are using, you may require that no task use more than 1 ms of processing prior to releasing control back to the RTOS. Although that may be a true requirement and it involves software and should be tested â€" it is truly a design requirement and should be included in the Software Design Document or in the Source code. Consider the target audience for each specification to identify what goes into what documents. It should define everything Software needs to develop the software. Thus, the SRS should define everything explicitly or preferably by reference that software needs to develop the software. References should include the version number of the target document. Also, consider using master document tools which allow you to include other documents and easily access the full requirements. Take your time with complicated requirements. Vagueness in those areas will come back to bite you later. This is a great question. There is no question that there is balance in this process. We have also seen customers kill good products by spending too much time specifying it. However,

the bigger problem is at the other end of the spectrum. We have found that taking the time up front pays dividends down stream. Here are some of our guidelines: Spend time specifying and documenting well software that you plan to keep. Keep documentation to a minimum when the software will only be used for a short time or has a limited number of users. Have separate individuals write the specifications not the individual who will write the code. The person to write the specification should have good communication skills. Pretty diagrams can help but often tables and charts are easier to maintain and can communicate the same requirements. Conversely, watch out for over-documenting those functions that are well understood by many people but for which you can create some great requirements. Keep the SRS up to date as you make changes. Test the requirements document by using it as the basis for writing the test plan.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

4: Computer Systems VxWorks Embedded Software Engineer Job in Veldhoven, Netherlands

Embedded software is computer software, written to control machines or devices that are not typically thought of as computers, commonly known as embedded www.enganchecubano.com is typically specialized for the particular hardware that it runs on and has time and memory constraints.

Introduction Approximately 3 billion embedded CPUs are sold each year, with smaller 4-, 8-, and bit CPUs dominating by quantity and aggregate dollar amount [1]. This paper seeks to expand the area of discussion to encompass a wide range of embedded systems. The extreme diversity of embedded applications makes generalizations difficult. Nonetheless, there is emerging interest in the entire range of embedded systems e. This paper and the accompanying tutorial seek to identify significant areas in which embedded computer design differs from more traditional desktop computer design. They also present "design challenges" encountered in the course of designing several real systems. These challenges are both opportunities to improve methodology and tool support as well as impediments to deploying such support to embedded system design teams. In some cases research and development has already begun in these areas -- and in other cases it has not. All characterizations are implicitly qualified to indicate a typical, representative, or perhaps simply an anecdotal case rather than a definitive statement about all embedded systems. While it is understood that each embedded system has its own set of unique requirements, it is hoped that the generalizations and examples presented here will provide a broad-brush basis for discussion and evolution of CAD tools and design methodologies. Example Embedded Systems Figure 1 shows one possible organization for an embedded system. An embedded system encompasses the CPU as well as many other resources. In addition to the CPU and memory hierarchy, there are a variety of interfaces that enable the system to measure, manipulate, and otherwise interact with the external environment. Some differences with desktop computing may be: The human interface may be as simple as a flashing light or as complicated as real-time robotic vision. The diagnostic port may be used for diagnosing the system that is being controlled -- not just for diagnosing the computer. Software often has a fixed function, and is specific to the application. In addition to the emphasis on interaction with the external world, embedded systems also provide functionality specific to their applications. Instead of executing spreadsheets, word processing and engineering analysis, embedded systems typically execute control laws, finite state machines, and signal processing algorithms. They must often detect and react to faults in both the computing and surrounding electromechanical systems, and must manipulate application-specific user interface devices. Four example embedded systems with approximate attributes. In order to make the discussion more concrete, we shall discuss four example systems Table 1. Each example portrays a real system in current production, but has been slightly genericized to represent a broader cross-section of applications as well as protect proprietary interests. The four examples are a Signal Processing system, a Mission Critical control system, a Distributed control system, and a Small consumer electronic system. Using these four examples to illustrate points, the following sections describe the different areas of concern for embedded system design: Desktop computing design methodology and tool support is to a large degree concerned with initial design of the digital system itself. To be sure, experienced designers are cognizant of other aspects, but with the recent emphasis on quantitative design e. However, such an approach is insufficient to create embedded systems that can effectively compete in the marketplace. This is because in many cases the issue is not whether design of an immensely complex system is feasible, but rather whether a relatively modest system can be highly optimized for life-cycle cost and effectiveness. While traditional digital design CAD tools can make a computer designer more efficient, they may not deal with the central issue -- embedded design is about the system, not about the computer. In desktop computing, design often focuses on building the fastest CPU, then supporting it as required for maximum computing speed. In embedded systems the combination of the external interfaces sensors, actuators and the control or sequencing algorithms is or primary importance. The CPU simply exists as a way to implement those functions. The following experiment should serve to illustrate this point: Then ask the same people which CPU is used for the engine controller in their car and whether the CPU type influenced the purchasing decision. In high-end

embedded systems, the tools used for desktop computer design are invaluable. However, many embedded systems both large and small must meet additional requirements that are beyond the scope of what is typically handled by design automation. These additional needs fall into the categories of special computer design requirements, system-level requirements, life-cycle support issues, business model compatibility, and design culture issues. Computer Design Requirements Embedded computers typically have tight constraints on both functionality and implementation. In particular, they must guarantee real time operation reactive to external events, conform to size and weight limits, budget power and cooling consumption, satisfy safety and reliability requirements, and meet tight cost targets. In many cases the system design must take into account worst case performance. Predicting the worst case may be difficult on complicated architectures, leading to overly pessimistic estimates erring on the side of caution. Reactive computation means that the software executes in response to external events. These events may be periodic, in which case scheduling of events to guarantee performance may be possible. On the other hand, many events may be aperiodic, in which case the maximum event arrival rate must be estimated in order to accommodate worst case situations. Most embedded systems have a significant reactive component. Worst case design analyses without undue pessimism in the face of hardware with statistical performance characteristics e. Small size, low weight Many embedded computers are physically located within some larger artifact. Therefore, their form factor may be dictated by aesthetics, form factors existing in pre-electronic versions, or having to fit into interstices among mechanical components. In transportation and portable systems, weight may be critical for fuel economy or human endurance. Among the examples, the Mission Critical system has much more stringent size and weight requirements than the others because of its use in a flight vehicle, although all examples have restrictions of this type. Packaging and integration of digital, analog, and power circuits to reduce size. Safe and reliable Some systems have obvious risks associated with failure. In mission-critical applications such as aircraft flight control, severe personal injury or equipment damage could result from a failure of the embedded computer. Traditionally, such systems have employed multiply-redundant computers or distributed consensus protocols in order to ensure continued operation after an equipment failure e. This vulnerability is often resolved at the system level as discussed later. Low-cost reliability with minimal redundancy. Harsh environment Many embedded systems do not operate in a controlled environment. Excessive heat is often a problem, especially in applications involving combustion e. Additional problems can be caused for embedded computing by a need for protection from vibration, shock, lightning, power supply fluctuations, water, corrosion, fire, and general physical abuse. For example, in the Mission Critical example application the computer must function for a guaranteed, but brief, period of time even under non-survivable fire conditions. De-rating components differently for each design, depending on operating environment. Cost sensitivity Even though embedded computers have stringent requirements, cost is almost always an issue even increasingly for military systems. Although designers of systems large and small may talk about the importance of cost with equal urgency, their sensitivity to cost changes can vary dramatically. A reason for this may be that the effect of computer costs on profitability is more a function of the proportion of cost changes compared to the total system cost, rather than compared to the digital electronics cost alone. However, with in the Small system decisions increasing costs by even a few cents attract management attention due to the huge multiplier of production quantity combined with the higher percentage of total system cost it represents. Variable "design margin" to permit tradeoff between product robustness and aggressive cost optimization. System-level requirements In order to be competitive in the marketplace, embedded systems require that the designers take into account the entire system when making design decisions. End-product utility The utility of the end product is the goal when designing an embedded system, not the capability of the embedded computer itself. Then, software is used to coordinate the mechanisms and define their functionality, often at the level of control system equations or finite state machines. Finally, computer hardware is made available as infrastructure to execute the software and interface it to the external world. While this may not be an exciting way for a hardware engineer to look at things, it does emphasize that the total functionality delivered by the system is what is paramount. But, it is the safety and reliability of the total embedded system that really matters. The Distributed system example is mission critical, but does not employ computer redundancy. Instead, mechanical safety backups are activated

when the computer system loses control in order to safely shut down system operation. A bigger and more difficult issue at the system level is software safety and reliability. This is a difficult problem that will take many years to address, and may not be properly appreciated by non-computer engineers and managers involved in system design decisions [12] discusses the role of computers in system safety. Cheap, available systems using unreliable components. Controlling physical systems The usual reason for embedding a computer is to interact with the environment, often by monitoring and controlling external machinery. In order to do this, analog inputs and outputs must be transformed to and from digital signal levels. Additionally, significant current loads may need to be switched in order to operate motors, light fixtures, and other actuators. All these requirements can lead to a large computer circuit board dominated by non-digital components. In some systems "smart" sensors and actuators that contain their own analog interfaces, power switches, and small CPUS may be used to off-load interface hardware from the central embedded computer. This brings the additional advantage of reducing the amount of system wiring and number of connector contacts by employing an embedded network rather than a bundle of analog wires. However, this change brings with it an additional computer design problem of partitioning the computations among distributed computers in the face of an inexpensive network with modest bandwidth capabilities. Distributed system tradeoffs among analog, power, mechanical, network, and digital hardware plus software. Power management A less pervasive system-level issue, but one that is still common, is a need for power management to either minimize heat production or conserve battery power. While the push to laptop computing has produced "low-power" variants of popular CPUs, significantly lower power is needed in order to run from inexpensive batteries for 30 days in some applications, and up to 5 years in others. Ultra-low power design for long-term battery operation. Life-cycle support Figure 2 shows one view of a product life-cycle a simplified version of the view taken by [13]. An embedded system lifecycle. First a need or opportunity to deploy new technology is identified. Then a product concept is developed. This is followed by concurrent product and manufacturing process design, production, and deployment. But in many embedded systems, the designer must see past deployment and take into account support, maintenance, upgrades, and system retirement issues in order to actually create a profitable design. Some of the issues affecting this life-cycle profitability are discussed below. Component acquisition Because an embedded system may be more application-driven than a typical technology-driven desktop computer design, there may be more leeway in component selection. Thus, component acquisition costs can be taken into account when optimizing system life-cycle cost. For example, the cost of a component generally decreases with quantity, so design decisions for multiple designs should be coordinated to share common components to the benefit of all.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

5: Embedded system - Wikipedia

Loose Ends Introduction Requirements and specifications are very important components in the development of any embedded system. While it is a common tendency for designers to be anxious about starting the design and implementation, discussing requirements with the customer is vital in the construction of safety-critical systems. For activities in this first stage has significant impact on the downstream results in the system life cycle. For example, errors developed during the requirements and specifications stage may lead to errors in the design stage. When this error is discovered, the engineers must revisit the requirements and specifications to fix the problem. This leads not only to more time wasted but also the possibility of other requirements and specifications errors. Many accidents are traced to requirements flaws, incomplete implementation of specifications, or wrong assumptions about the requirements. While these problems may be acceptable in non-safety-critical systems, safety-critical systems cannot tolerate errors due to requirements and specifications. Therefore, it is necessary that the requirements are specified correctly to generate clear and accurate specifications. There is a distinct difference between requirements and specifications. A requirement is a condition needed by a user to solve a problem or achieve an objective. A specification is a document that specifies, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system, and often, the procedures for determining whether these provisions have been satisfied. For example, a requirement for a car could be that the maximum speed to be at least mph. The specification for this requirement would include technical information about specific design aspects. Another term that is commonly seen in books and papers is requirements specification which is a document that specifies the requirements for a system or component. It includes functional requirements, performance requirements, interface requirements, design requirements, and developement standards. So the requirements specification is simply the requirements written down on paper. Key Concepts Establishing Correct Requirements The first step toward developing accurate and complete specifications is to establish correct requirements. As easy as this sounds, establishing correct requirements is extremely difficult and is more of an art than a science. There are different steps one can take toward establishing correct requirements. Although some of the suggestions sound fairly obvious, actually puttting them into practice may not be as easy as it sounds. The first step is to negotiate a common understanding. There is no point in trying to establish exact specifications if the designers and customers cannot even agree on what the requirements are. Problem stems from ambiguities in stating requirements. Possible interpretations of this requirement includes building a bus, train, or airplane, among other possibilities. Although each of these transportation devices satisfy the requirement, they are certainly very different. Ambiguous requirements can be caused by missing requirements, ambiguous words, or introduced elements. The above requirement does not state how fast the people should be transported from Boston to Washington D. Taking an airplane would certainly be faster than riding a bus or train. These are also missing requirements. What exactly does "group" imply? A group can consist of 5 people, people, people, etc. The requirement states to "create a means" and not "design a transportation device". This is an example of introduced elements where an incorrect meaning slipped into the discussion. It is important to eliminate or at least reduce ambiguities as early as possible because the cost of them increases as we progress in the development life cycle. Often the problem one has in establishing correct requirements is how to get started. One of the most important things in getting started is to ask questions. Context-free questions are high-level questions that are posed early in a project to obtain information about global properties of the design problem and potential solutions. Examples of context-free questions include who is the client? These questions force both sides, designer and customer, to look at the higher issues. Also, since these questions are appropriate for any project, they can be prepared in advance. Another important point is to get the right people involved. There is no point in discussing requirements if the appropriate people are not involved in the discussion. Related to getting the right people involved is making meetings work. Having effective meetings is not as easy

as it sounds. However, since they play a central role in establishing requirements it is essential to know how to make meetings work. There are important points to keep in mind when creating effective meetings, which include creating a culture of safety for all participants, keeping the meeting to an appropriate size, and other points. Ideas are essential in establishing correct requirements, so it is important that people can get together and generate ideas. Every project will also encounter conflicts. Conflicts can occur from personality clashes, people that cannot get along, intergroup prejudice such as those between technical people and marketing people, and level differences. It is important that a facilitator is present to help resolve conflicts. In establishing requirements, it is important to specifically establish the functions, attributes, constraints, preferences, and expectations of the product. Usually in the process of gaining information, functions are the first ones to be defined. Functions describe what the product is going to accomplish. It is also important to determine the attributes of a product. Attributes are characteristics desired by the client, and while 2 products can have similar functions, they can have competely different attributes. After all the attributes have been clarified and attached to functions, we must determine the constraints on each of the atrributes. Preferences, which is a desirable but optional condition placed on an attribute, can also be defined in addition to its constraints. This will largely determine the success of the product. Testing is the final step on the road to establishing correct requirements. There are several testing methods used, as listed below. This involves asking questions such as how fast? Technical review - A testing tool for indicating the progress of the requirements work. It can be formal or informal and generally only deals with technical issues. Technical reviews are necessary because it is not possible to produce error-free requirements and usually it is difficult for the producers to see their own mistakes. User satisfaction test - A test used on a regular basis to determine if a customer will be satisifed with a product. Black box test cases - Constructed primarily to test the completeness, accuracy, clarity, and conciseness of the requirements. Existing products - Useful in determining the desirable and undesirable characteristics of a new product. At some point it is necessary to end the requirements process as the fear of ending can lead to an endless cycle. This does not mean that it is impossible to revisit the requirements at a later point in the development life cycle if necessary. However, it is important to end the process when all the necessary requirements have been determined, otherwise you will never proceed to the design cycle. Establishing good requirements requires people with both technical and communication skills. Technical skills are required as the embedded system will be highly complex and may require knowledge from different engineering disciplines such as electrical engineering and mechanical engineering. Communication skills are necessary as there is a lot of exchange of information between the customer and the designer. Without either of these two skills, the requirements will be unclear or inaccurate. It is essential that requirements in safety critical embedded systems are clear, accurate, and complete. The problem with requirements is that they are often weak about what a system should not do. In a dependable system, it is just as important to specify what a system is not suppose to do as to specfiy what a system is suppose to do. These systems have an even greater urgency that the requirements are complete because they will only be dependable if we know exactly what a system will do in a certain state and the actions that it should not perform. Requirements with no ambiguities will also make the system more dependable. Extra requirements will usually be required in developing a dependable embedded system. The universe can be considered a system, and so can an atom. A system is very loosely defined and can be considered as any of the following definitions. Systems engineering is not a technical specialty but is a process used in the evolution of systems from the point when a need is identified through production and construction to deployment of the system for consumer use. The development of embedded systems also requires the knowledge of different engineering disciplines and can follow the techniques used for systems engineering. Therefore, it is appropriate that the steps used in establishing system requirements also be applicable to requirements for embedded systems. The conceptual system design is the first stage in the systems design life cycle and an example of the systems definition requirements process is shown in Figure 1. Each individual box will be explain below. Example of system requirements definition process [Blanchard90] In establishing system requirements, the first step is to define a need. This need is based on a want or desire. Usually, an individual or organization identifies a need for an item or function and then a new or modified system is developed to fulfill

the requirement. After a need is defined, feasibility studies should be conducted to evaluate various technical approaches that can be taken. The system operational requirements should also be defined. This includes the definition of system operating characteristics, maintenance support concept for the system, and identification of specific design criteria. In particular, the system operational requirements should include the following elements. Performance and physical parameters - Definition of the operating characteristics or functions of the system. Use requirements - Anticipation of the use of the system. Operational deployment or distribution - Identification of transportation and mobility requirements. Includes quantity of equipment, personnel, etc.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

6: Classification of Embedded Systems with Applications

Requirements: B Eng Computer or Electronic Engineering degree 2 Years experience in embedded and / or PC software development 1 year experience in C and/or C++ A desire to perform computer software development Systematic and analytical approach to problem solving Adhere to and contribute to good development standards and principles Good.

Must be fail-proof Embedded Design Examples To demonstrate the variation in design requirements from one embedded system to the next, as well as the possible effects of these requirements on the hardware, we will now take some time to describe three embedded systems in some detail. Digital Watch At the current peak of the evolutionary path that began with sundials, water clocks, and hourglasses is the digital watch. Among its many features are the presentation of the date and time usually to the nearest second , the measurement of the length of an event to the nearest hundredth of a second, and the generation of an annoying little sound at the beginning of each hour. As it turns out, these are very simple tasks that do not require very much processing power or memory. In fact, the only reason to employ a processor at all is to support a range of models and features from a single hardware design. The typical digital watch contains a simple, inexpensive 4-bit processor. Because processors with such small registers cannot address very much memory, this type of processor usually contains its own on-chip ROM. And, if there are sufficient registers available, this application may not require any RAM at all. In fact, all of the electronicsâ€" processor, memory, counters, and real-time clocksâ€"are likely to be stored in a single chip. The only other hardware elements of the watch are the inputs buttons and outputs display and speaker. If, after production, some watches are found to keep more reliable time than most, they can be sold under a brand name with a higher markup. For the rest, a profit can still be made by selling the watch through a discount sales channel. For lower-cost versions, the stopwatch buttons or speaker could be eliminated. This would limit the functionality of the watch but might require few or even no software changes. And, of course, the cost of all this development effort may be fairly high, because it will be amortized over hundreds of thousands or even millions of watch sales. In the case of the digital watch, we see that software, especially when carefully designed, allows enormous flexibility in response to a rapidly changing and highly competitive market. Video Game Player When you pull the Sony PlayStation 2 out from your entertainment center, you are preparing to use an embedded system. In some cases, these machines are more powerful than personal computers of the same generation. Yet video game players for the home market are relatively inexpensive compared with personal computers. It is the competing requirements of high processing power and low production cost that keep video game designers awake at night. They might even encourage their engineers to design custom processors at a development cost of millions of dollars each. So, although there might be a bit processor inside your video game player, it is probably not the same processor that would be found in a general-purpose computer. In all likelihood, the processor is highly specialized for the demands of the video games it is intended to play. Because production cost is so crucial in the home video game market, the designers also use tricks to shift the costs around. For example, one tactic is to move as much of the memory and other peripheral electronics as possible off of the main circuit board and onto the game cartridges. So, while the system might have a powerful bit processor, it might have only a few megabytes of memory on the main circuit board. This is just enough memory to bootstrap the machine to a state from which it can access additional memory on the game cartridge. We can see from the case of the video game player that in high-volume products, a lot of development effort can be sunk into fine-tuning every aspect of a product. Mars Rover In , two unmanned spacecrafts arrived on the planet Mars. As part of their mission, they were to collect samples of the Martian surface, analyze the chemical makeup of each, and transmit the results to scientists back on Earth. Those Viking missions were amazing. Surrounded by personal computers that must be rebooted occasionally, we might find it remarkable that more than 30 years ago, a team of scientists and engineers successfully built two computers that survived a journey of 34 million miles and functioned correctly for half a decade. Clearly, reliability was one of the most important requirements for these systems. What if a memory chip had failed? Or the software had

contained bugs that had caused it to crash? Or an electrical connection had broken during impact? There is no way to prevent such problems from occurring, and on other space missions, these problems have proved ruinous. So, all of these potential failure points and many others had to be eliminated by adding redundant circuitry or extra functionality: Its primary goal was to demonstrate the feasibility of getting to Mars on a budget. They might have reduced the amount of redundancy somewhat, but they still gave Pathfinder more processing power and memory than Viking. The Mars Pathfinder was actually two embedded systems: These choices reflect the different functional requirements of the two systems. An embedded software developer is the one who gets her hands dirty by getting down close to the hardware. Embedded software development, in most cases, requires close interaction with the physical worldâ€"the hardware platform. These application developers typically do not have any interaction with the hardware. Hardware knowledge The embedded software developer must become intimately familiar with the integrated circuits, the boards and buses, and the attached devices used in order to write solid embedded software also called firmware. Efficient code Because embedded systems are typically designed with the least powerful and most cost-effective processor that meets the performance requirements of the system, embedded software developers must make every line of code count. The ability to write efficient code is a great quality to possess as a firmware developer. Peripheral interfaces At the lowest level, firmware is very specialized, because each component or circuit has its own activity to perform and, furthermore, its own way of performing that activity. Embedded developers need to know how to communicate with the different devices or peripherals in order to have full control of the devices in the system. Reacting to stimuli from external peripherals is a large part of embedded software development. For example, in one microwave oven, the firmware might get the data from a temperature sensor by reading an 8-bit register in an external analog-to-digital converter; in another system, the data might be extracted by controlling a serial bus that interfaces to the external sensor circuit via a single wire. Robust code There are expectations that embedded systems will run for years in most cases. This is not a typical requirement for software applications written for a PC or Mac. Now, there are exceptions. However, if you had to keep unplugging your microwave in order to get it to heat up your lunch for the proper amount of time, it would probably be the last time you purchased a product from that company. Minimal resources Along the same lines of creating a more robust system, another large differentiator between embedded software and other types of software is resource constraints. The rules for writing firmware are different from the rules for writing software for a PC. Take memory allocation, for instance. An application for a modern PC can take for granted that it will have access to practically limitless resources. But in an embedded system, you will run out of memory if you do not plan ahead and design the software properly. For example, using standard dynamic memory allocation functions can cause fragmentation, and eventually the system may cease to operate. This requires a reboot since you have no place to store incoming data. Quite often, in embedded software, a developer will allocate all memory needed by the system at initialization time. This is safer than using dynamic memory allocation, though it cannot always be done. Reusable software As we mentioned before , code portability or code reuseâ€"writing software so that it can be moved from hardware platform to hardware platformâ€"is very useful to aid transition to new projects. This cannot always be done; we have seen how individual each embedded system is. Throughout this book, we will look at basic methods to ensure that your embedded code can be moved more easily from project to project. Development tools The tools you will use throughout your career as an embedded developer will vary from company to company and often from project to project. This means you will need to learn new tools as you continue in your career. Typically, these tools are not as powerful or as easy to use as those used in PC software development. This requires you, as the firmware developer, and the one responsible for debugging your code, to be very resourceful and have a bag of techniques you can call upon when the debug environment is lacking. These are just a few qualities that separate embedded software developers from the rest of the pack. We will investigate these and other techniques that are specific to embedded software development as we continue. The Lowest Common Denominator One of the few constants across most embedded systems is the use of the C programming language. More than any other, C has become the language of embedded programmers. This has not always been the case, and it will not continue to be so forever. However, at this time, C is the closest thing there is to

a standard in the embedded world. Because successful software development so frequently depends on selecting the best language for a given project, it is surprising to find that one language has proven itself appropriate for both 8-bit and bit processors; in systems with bytes, kilobytes, and megabytes of memory; and for development teams that range from one to a dozen or more people. Yet this is precisely the range of projects in which C has thrived. The C programming language has plenty of advantages. It is small and fairly simple to learn, compilers are available for almost every processor in use today, and there is a very large body of experienced C programmers. In addition, C has the benefit of processor-independence, which allows programmers to concentrate on algorithms and applications rather than on the details of a particular processor architecture. However, many of these advantages apply equally to other high-level languages. So why has C succeeded where so many other languages have largely failed? As we shall see throughout the book, C gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. In fact, Brian W. Kernighan and Dennis M. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do. These may be combined and moved about with the arithmetic and logical operators implemented by real machines. Few popular high-level languages can compete with C in the production of compact, efficient code for almost all processors. And, of these, only C allows programmers to interact with the underlying hardware so easily. Other Embedded Languages Of course, C is not the only language used by embedded programmers. In the early days, embedded software was written exclusively in the assembly language of the target processor. This gave programmers complete control of the processor and other hardware, but at a price. Assembly languages have many disadvantages, not the least of which are higher software development costs and a lack of code portability. In addition, finding skilled assembly programmers has become much more difficult in recent years. Assembly is now used primarily as an adjunct to the high-level language, usually only for startup system code or those small pieces of code that must be extremely efficient or ultra-compact, or cannot be written in any other way. Forth is efficient but extremely low-level and unusual; learning to get work done with it takes more time than with C.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

7: Introduction - Programming Embedded Systems, 2nd Edition [Book]

Introduction Requirements and specifications are very important components in the development of any embedded system. Requirements analysis is the first step in the system design process, where a user's requirements should be clarified and documented to generate the corresponding specifications.

However, they may also use some more specific tools: In circuit debuggers or emulators see next section. Utilities to add a checksum or CRC to a program, so the embedded system can check if the program is valid. For systems using digital signal processing , developers may use a math workbench to simulate the mathematics. System level modeling and simulation tools help designers to construct simulation models of a system with hardware components such as processors , memories , DMA , interfaces , buses and software behavior flow as a state diagram or flow diagram using configurable library blocks. Simulation is conducted to select right components by performing power vs. Typical reports that helps designer to make architecture decisions includes application latency, device throughput, device utilization, power consumption of the full system as well as device-level power consumption. A model-based development tool creates and simulate graphical data flow and UML state chart diagrams of components like digital filters, motor controllers, communication protocol decoding and multi-rate tasks. Custom compilers and linkers may be used to optimize specialized hardware. An embedded system may have its own special language or design tool, or add enhancements to an existing language such as Forth or Basic. Another alternative is to add a real-time operating system or embedded operating system Modeling and code generating tools often based on state machines Software tools can come from several sources: Software companies that specialize in the embedded market Ported from the GNU software development tools Sometimes, development tools for a personal computer can be used if the embedded processor is a close relative to a common PC processor As the complexity of embedded systems grows, higher level tools and operating systems are migrating into machinery where it makes sense. For example, cellphones , personal digital assistants and other consumer computers often need significant software that is purchased or provided by a person other than the manufacturer of the electronics. Embedded systems are commonly found in consumer, cooking, industrial, automotive, medical applications. Household appliances, such as microwave ovens, washing machines and dishwashers, include embedded systems to provide flexibility and efficiency. Debugging[edit] Embedded debugging may be performed at different levels, depending on the facilities available. The different metrics that characterize the different forms of embedded debugging are: From simplest to most sophisticated they can be roughly grouped into the following areas: Interactive resident debugging, using the simple shell provided by the embedded operating system e. Forth and Basic External debugging using logging or serial port output to trace operation using either a monitor in flash or using a debug server like the Remedy Debugger that even works for heterogeneous multicore systems. An in-circuit emulator ICE replaces the microprocessor with a simulated equivalent, providing full control over all aspects of the microprocessor. A complete emulator provides a simulation of all aspects of the hardware, allowing all of it to be controlled and modified, and allowing debugging on a normal PC. The downsides are expense and slow operation, in some cases up to times slower than the final system. This is used to debug hardware, firmware and software interactions across multiple FPGA with capabilities similar to a logic analyzer. Software-only debuggers have the benefit that they do not need any hardware modification but have to carefully control what they record in order to conserve time and storage space. The view of the code may be as HLL source-code , assembly code or mixture of both. Because an embedded system is often composed of a wide variety of elements, the debugging strategy may vary. For instance, debugging a software- and microprocessor- centric embedded system is different from debugging an embedded system where most of the processing is performed by peripherals DSP, FPGA, and co-processor. An increasing number of embedded systems today use more than one single processor core. A common problem with multi-core development is the proper synchronization of software execution. A graphical view is presented by a host PC tool, based on a recording of the system behavior. The trace recording can be performed in software, by the RTOS, or by special tracing hardware. RTOS tracing allows

developers to understand timing and performance issues of the software system and gives a good understanding of the high-level system behaviors. Reliability[edit] Embedded systems often reside in machines that are expected to run continuously for years without errors, and in some cases recover by themselves if an error occurs. Therefore, the software is usually developed and tested more carefully than that for personal computers, and unreliable mechanical moving parts such as disk drives, switches or buttons are avoided. Specific reliability issues may include: The system cannot safely be shut down for repair, or it is too inaccessible to repair. Examples include space systems, undersea cables, navigational beacons, bore-hole systems, and automobiles. The system must be kept running for safety reasons. Often backups are selected by an operator. Examples include aircraft navigation, reactor control systems, safety-critical chemical factory controls, train signals. The system will lose large amounts of money when shut down: Telephone switches, factory controls, bridge and elevator controls, funds transfer and market making, automated sales and service. A variety of techniques are used, sometimes in combination, to recover from errorsâ€”both software bugs such as memory leaks , and also soft errors in the hardware: This encapsulation keeps faults from propagating from one subsystem to another, improving reliability. This may also allow a subsystem to be automatically shut down and restarted on fault detection. Immunity Aware Programming High vs. For low-volume or prototype embedded systems, general purpose computers may be adapted by limiting the programs or by replacing the operating system with a real-time operating system. Embedded software architectures[edit] There are several different types of software architecture in common use. Simple control loop[edit] In this design, the software simply has a loop. The loop calls subroutines , each of which manages a part of the hardware or software. Hence it is called a simple control loop or control loop. Interrupt-controlled system[edit] Some embedded systems are predominantly controlled by interrupts. This means that tasks performed by the system are triggered by different kinds of events; an interrupt could be generated, for example, by a timer in a predefined frequency, or by a serial port controller receiving a byte. These kinds of systems are used if event handlers need low latency, and the event handlers are short and simple. Usually, these kinds of systems run a simple task in a main loop also, but this task is not very sensitive to unexpected delays. Sometimes the interrupt handler will add longer tasks to a queue structure. Later, after the interrupt handler has finished, these tasks are executed by the main loop. This method brings the system close to a multitasking kernel with discrete processes. Cooperative multitasking[edit] A nonpreemptive multitasking system is very similar to the simple control loop scheme, except that the loop is hidden in an API. The advantages and disadvantages are similar to that of the control loop, except that adding new software is easier, by simply writing a new task, or adding to the queue. Preemptive multitasking or multi-threading[edit] In this type of system, a low-level piece of code switches between tasks or threads based on a timer connected to an interrupt. This is the level at which the system is generally considered to have an "operating system" kernel. Depending on how much functionality is required, it introduces more or less of the complexities of managing multiple tasks running conceptually in parallel. As any code can potentially damage the data of another task except in larger systems using an MMU programs must be carefully designed and tested, and access to shared data must be controlled by some synchronization strategy, such as message queues , semaphores or a non-blocking synchronization scheme. Because of these complexities, it is common for organizations to use a real-time operating system RTOS , allowing the application programmers to concentrate on device functionality rather than operating system services, at least for large systems; smaller systems often cannot afford the overhead associated with a generic real-time system, due to limitations regarding memory size, performance, or battery life. The choice that an RTOS is required brings in its own issues, however, as the selection must be done prior to starting to the application development process. This timing forces developers to choose the embedded operating system for their device based upon current requirements and so restricts future options to a large extent. These trends are leading to the uptake of embedded middleware in addition to a real-time operating system. Microkernels and exokernels[edit] A microkernel is a logical step up from a real-time OS. The usual arrangement is that the operating system kernel allocates memory and switches the CPU to different threads of execution. User mode processes implement major functions such as file systems, network interfaces, etc. In general, microkernels succeed when the task switching and intertask communication is fast and fail when they are slow. Exokernels

communicate efficiently by normal subroutine calls. The hardware and all the software in the system are available to and extensible by application programmers. Monolithic kernels[edit] In this case, a relatively large kernel with sophisticated capabilities is adapted to suit an embedded environment. This gives programmers an environment similar to a desktop operating system like Linux or Microsoft Windows , and is therefore very productive for development; on the downside, it requires considerably more hardware resources, is often more expensive, and, because of the complexity of these kernels, can be less predictable and reliable. Common examples of embedded monolithic kernels are embedded Linux and Windows CE. Despite the increased cost in hardware, this type of embedded system is increasing in popularity, especially on the more powerful embedded devices such as wireless routers and GPS navigation systems. Here are some of the reasons: Ports to common embedded chip sets are available. They permit re-use of publicly available code for device drivers , web servers , firewalls , and other code. Development systems can start out with broad feature-sets, and then the distribution can be configured to exclude unneeded functionality, and save the expense of the memory that it would consume. Many engineers believe that running application code in user mode is more reliable and easier to debug, thus making the development process easier and the code more portable. Additional software components[edit] In addition to the core operating system, many embedded systems have additional upper-layer software components. If the embedded device has audio and video capabilities, then the appropriate drivers and codecs will be present in the system. In the case of the monolithic kernels, many of these software layers are included. In the RTOS category, the availability of the additional software components depends upon the commercial offering.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

8: System Requirements for Procare Software & Hardware

Many embedded systems have requirements that differ significantly both in details and in scope from desktop computers. In particular, the demands of the specific application and the interface with external equipment may dominate the system design.

The best place to find jobs and internships all across Europe in the broad fields of Engineering, Software, Science and Technology. Register and face the future with your own personal adventure! Upload your resume or CV As an applicant or job seeker looking for new career opportunities, you can simply upload your resume and apply to positions at most leading companies all across Europe. We offer you the chance to browse through over technical specialties. Hey, is your specialty not there, just send us an e-mail here. You design, construct and maintain bridges, roads or buildings and many more to make our life easier, faster and safer. Are you looking for a challenging position in Civil engineering? If you have a technical degree and are looking for an opportunity in Aerospace Science, Astrophysics, Aviation, Space Engineering, Space Technology, Planetary Science, Space Research, Rocket Science or related then you have found the best place to find the right job for you. Browse here for the latest jobs in Aerospace. This can vary from App development, responsive website development or front-end and back-end jobs listed. You can conveniently search and apply to those matching your skillset and ambition. You can select a country Germany, Netherlands, Belgium, UK, Austria, Switzerland or any of the other European countries , an education level BSc, MSc or PhD or specialism and search for a related internship or graduation assignment in these sectors matching your skills or future ambition. Our vision is to enable affordable microelectronics that improve the quality of life. To achieve this, our mission is to invent, develop, manufacture and service advanced technology for high-tech lithography, metrology and software solutions for the semiconductor industry. This results in increasingly powerful and capable electronics that enable the world to progress within a multitude of fields, including healthcare, technology, communications, energy, mobility, and entertainment. This vacancy might be just the right one for you! In this role you would be making your mark on the computing platform for the next generation cutting edge chip manufacturing equipment lithography machines ASML is currently developing. A key part of this platform are the CPU boards running VxWorks, hosting several time-critical control loops. You will be working in the team approx. The Computer Systems team offers a challenging environment where real-time computer systems, IT networking and industrial automation come together in a highly complex set of features that need to deliver reliability, security, usability and performance. You will be working on VxWorks main responsibility and Linux WindRiver, Redhat , but activities might not be limited to that. As we are always pushing the platform to its limits you will participate in performance analysis of our OS and applications, be a key person in issue resolving and act as a technical contact with the supplier WindRiver. With the team you strive to build the best possible platform, and delivery a truly rugged implementation to operate on a level in line with our customer need. These stakeholders include the SW development partners, but also e. Experience; -VxWorks OS development:

9: Embedded System Design Issues (the Rest of the Story)

Designing hardware and software for embedded computers is critical and it requires complete knowledge of this field. There are five things that should be known by a designer, which serve as the objectives of embedded computer.

SOFTWARE REQUIREMENTS FOR EMBEDDED COMPUTERS pdf

History of shahrukh khan Nearly But Not Quite A devils chaplain Day the Revolution began Mensuration formulas list in hindi The joy luck club The Day Tigger Lost His Bounce (Pooh Story Workbooks) U2022/tNational Institute of Neurological Disorders Stroke Toastmasters high performance leadership manual Machine-code generation The bases of design When love falls yiruma sheet music Handbook of professional tour management The craft of crime The Haunted railway game 12 Ways to Get to 11 (Aladdin Picture Books) Blue skies, dark waters Documents on British foreign policy, 1919-1939. First series. EFTPoS in the United Kingdom Draw anything you like Design-led participatory planning Sophie Bond and Michelle Thompson-Fawcett Nuclear reactions and nuclear structure The Middle East for Dummies Very Special Agents Labeling and inspection of imported meat Seizing opportunities : chieftaincy, land, and local administration Biology 6th edition campbell and reece The concept of the palace in the Andes Joanne Pillsbury Generals January and February William Faulknerscraft of revision Dutch grammar book St. Patrick on the stage. Love across color lines Times Compact Atlas of the World (World Atlas) History and published records of the Midway Congregational Church, Liberty County, Georgia Wireless home security system seminar report P 21. Give Me A Little Love A Little Time Failures of explanation in darwinian ecological anthropology Isi master list 2016 Orthodox prayer book