56259/

# Putting Safety in the Software

**Embedded System Conference**
**Class #426**

**Martha S. Wetherholt**
**NASA Glenn Research Center**
Martha.S.Wetherholt@grc.nasa.gov

**Kalynnda M. Berens**
**S.A.I.C.**
Kalynnda.M.Berens@grc.nasa.gov

## *Software Safety, a Practical Approach*

Software is a vital component of nearly every piece of modern technology. It is not a "sub-system", able to be separated out from the system as a whole, but a "co-system" that controls, manipulates, or interacts with the hardware

and with the end user. Software has its fingers into all the pieces of the pie. If that "pie", the system, can lead to injury, death, loss of major equipment, or impact your business bottom line, then software safety becomes vitally important.

Learning to think about software from a safety perspective is the focus of this paper. We want you to think of software as part of the safety critical system, a major part. This requires "system thinking" – being able to grasp the whole picture. Software's contribution to modern technology is both good and potentially bad. Software allows more complex and useful devices to be built. It can also contribute to plane crashes and power outages. We want you to see software in a whole new light, see it as a contributor to system hazards, and also as a possible fix or mitigation to some of those hazards.

## *Why you really care about SW Safety*

Many people still claim that software can't hurt you, unless the paper work falls on you. When we even take a cursory look to see what software controls and monitors in our life, I'm not sure how those folks can be so blind. Software has become and integral part of all our lives, from transportation to medical devices to cash flow. The only thing you can do with an F-22 fighter without software is to take a picture of it.

Safety is obviously an issue when we are using software to control either a few thousand tons of metal, flying through the air and gliding along on some rails, or a pocket-watch sized heart pacer. However, software is also used to provide and display critical information (such as the various status information for controlling a nuclear power plant) or the heartbeat of an in utero infant. These are areas where software safety is a concern as well.

A system/product is ***Safe*** when there is little to no chance for it to blow up, break, malfunction, or otherwise fail in such a way as to potentially injure someone.

A system/product is ***Not*** Safe when someone could die or be seriously injured.

Something is ***Critical*** when there is a potential for:

- Serious injury or death
- Serious impact to the bottom line
- Bad publicity, public reputation
- Vital information is accessible to the wrong folks

NASA includes the possible destruction of vital equipment as part of the definition of safety. Some failures can impact your business (incorrect ATM functions, international bank transfers, etc.), your reputation, and can lead to liability. These may also be considered under your definition of Critical software, but they are not necessarily a "safety" concern. Of course, if your ATM doesn't deliver the money to pay-off Guido, your loan shark, that might be classified as "safety critical".

While we stress safety critical software, the processes and analyses can be used as good practice and good approaches for mission/business critical systems as well. Nowadays you need to worry about more than the obvious "safety" issue of injury and potentially death. In this make-it-or-break-it business world you must be aware of (and meet) any regulatory requirements. Perhaps your company's good reputation can make your system "critical". And there is always the chance of getting sued these days, even if the product doesn't injure some one – failure to perform as expected is enough cause in some arenas.

While there is no defense against stupidity and ignorance, you should try to be as safe as possible. We all know of the cases where some genius decides his lawnmower is a good hedge clipper as well.

## Definitions

- **Hazard.** The presence of a potential risk situation caused by an unsafe act or condition. Every hazard has at least one cause, which in turn can lead to a number of effects (i.e., damage, illness, etc.).
- **Hazard cause.** Typically a fault, error, or defect in either the hardware or software or it can be a human operator or programmer error.
- **Hazard control.** A method for preventing the hazard from occurring or lessening it's effect if it does occur.

- **Error.** Human mistake which could cause a failure. For example, a mistake in engineering and development of the system (requirement specification, design, implementation), or a mistake during operation.
- **Fault.** Any change in state of an item which is considered to be anomalous and may warrant some type of corrective action. A fault may or may not lead to a failure.
- **Failure.** The inability of a system or component to perform its required functions within specified performance requirements. A fault may or may not lead to a failure. One or more faults can become a failure. All failures are the result of one or more faults.

- **Mishap/Accident.** An unplanned event or series of events that results in death, injury, occupational illness, or damage to or loss of equipment, property, or damage to the environment; an accident.
- **Risk.** A measure of the severity and likelihood of an accident or mishap. The probability that a specific threat will exploit a particular vulnerability of the system.
- **Criticality.** The relative measure of the consequences of a failure.
- **Severity.** Just how bad is the failure? Similar to criticality.
- **Reliability.** Operation of the software as expected, within a specified environment, for a given period of time.

**Faults and Failures**

Faults are preliminary indications that a failure may have occurred or will occur. Examples of *faults* included:

- device errors reported by Built In Test
- out of limits conditions on sensor values
- loss of communication with devices
- loss of power to a device
- communication error on bus transaction
- software exceptions (e.g., divide by zero, file not found)
- rejected commands
- measured performance values outside of commanded or expected values
- an incorrect step, process, or data definition in a computer program

*Failures* are the result of one or more faults. Consider the situation where the position of the shuttle's robotic arm is transmitted to a processor controlling the movement of the arm. If that transmission is garbled, the controlling processor may not have the correct position of the arm. This would be a fault. If the controlling processor repositions the arm based on faulty data, the arm may be positioned into the side of the shuttle! This would be a failure.

**Dealing with Hazards**

Since we want to make software safer, or at least the systems they control, monitor and verify, we need to look at software and the system before it is designed, *when danger is only a risk.* But how risky is it?

- What is the potential or likelihood for an accident to occur?
- How bad of an accident could it cause?

The combination determines the 'risk', and from that we estimate the proper amount of effort necessary to assure the software operates safely.

Not all situations have the same potential for destruction. Either the extent of damage or the chance of the damage even occurring vary.

For NASA, first and foremost we try to eliminate hazards by design choices where possible. Those we can't eliminate (and there are quite a few in our business) we need to mitigate and control. When mitigation can't lessen either the likelihood of occurrence or reduce the potential impact of the hazard, we must provide safeguards to prevent inadvertent use or exposure. If that fails to be practical, we resort to warnings.

Of course, the control part of the software is then "safety critical" as well as any software routines and algorithms used to detect critical hardware and environment situations.

## Safety vs. Reliability

Software is reliable if it works as expected when the product/system is used within specified limits and over a specified time period. For software to be safe, it must also not lead to a hazard even if used in an unintended manner or for a longer than specified time. Whether the system is t

urned off or turned on, you expect it to be relatively safe. Safety and reliability are different but can work to help each other.

A system is:

- ➔ **safe** if it doesn't kill anyone, or the system itself, while either performing its normal operations or, when unable to perform correctly, "fails-safe".
- ➔ **reliable** if it performs the required functions within specified parameters/environment and within predicted working timeframe consistently

***Software is <u>very</u> reliable, in that it does just what its programmed to do, over and over and over again. Even if what you programmed was in error – the software will execute it! Software doesn't wear out or 'break'.***

While a reliable system is often safer, it is not necessarily so. However, if the system can be counted on to always perform a certain way, that way can be made safe. So a reliable system aids in creating a safe system. However, a safe system is not always reliable. But there is a better chance of it being safe if there does not have to be safeguards for numerous unpredictable possibilities.

Here's an example of reliable software: A relay contact is usually used to signal elevators to return to the bottom floor of a building and open the doors in the event of a fire alarm. This is reliable, but is it safe? Yes, unless the fire is on the ground level - then it is lethal!

Consider this scenario: A pipe works within specified parameters (i.e. pressures). The regulator is controlled by software and is responsible for maintaining the pressure in the pipe within specified limits. If the pressure were to go outside the pipe's range, then the pipe becomes a hazard which may rip open, explode, etc. The software has a fault that, under some circumstances (such as a heavy processing load), allows a valve to open, increasing the pressure in the pipe. The software is unreliable, because its behavior varies with time in an unpredictable way. The software is not safe, because that unreliability can lead to a hazard.

## What Makes Software Hazardous?

Software is hazardous (safety critical) if it directly or indirectly contributes to the occurrence of a hazardous system state:

- Controls hazardous or safety critical hardware
- Monitors safety critical hardware as part of a hazard control
- Provides information upon which a safety-related decision is made
- Performs analysis that impacts automatic or manual hazardous operations
- Verifies hardware hazard controls

- Is used to verify safety critical hardware and/or software
- Is used to model or simulate safety critical applications

Software can be the cause of a hazard, if it controls hazardous hardware and the software fails, or fails to perform as needed or expected. Here are some examples of ways software can lead to a failure:

- ✓ Failure to detect a hardware problem (MONITORING)
- ✓ Failure to perform a function (CONTROL)
- ✓ Performing a function at the wrong time (CONTROL)
- ✓ Executing a function out of sequence
- ✓ Performing a function when the program is in the wrong state
- ✓ Executing the wrong function (CONTROL)
- ✓ Performing a function incompletely (CONTROL)
- ✓ Failure to "pass along" information or messages (CONTROL/ MONITORING)
- ✓ Passes misleading or old information (MONITORING)
- ✓ Provide information too soon or too late
- ✓ Simulation incorrectly models hardware reactions (VERIFY)

When software controls a hazard, it is important that the software be located such that the failure of another hazard control, or failure of the hazardous equipment, will not cause the software to fail as well. If the hazard is a physical, hardware problem that is outside of the computer processing arena (e.g. valve, robotic arm), then the controlling software can reside on the main processor with the "regular" software. However, it should be partitioned or firewalled from the non-critical software, to prevent accidental corruption.

If the hazard cause is a software error, then another process must monitor the situation. This could be on the same processor (partitioned from the rest of the software) or a separate processor. If a CPU failure can lead to the hazard, then there must be a "hot standby" processor that monitors the first CPU and can step in to safe the system if the CPU should fail.

When looking at hazard causes and controls, always think about whether the software has enough information to do its job. Are there enough sensors? What if a sensor fails – can the software extrapolate from other sensors, or is it dead in the water? Will a second processor be needed to monitor or provide backup? Often a little extra hardware can make software's job much easier.

**Safety Critical Software**

"Software" includes more than just desktop applications or embedded programs. It includes firmware (software residing in PROMs or other programmable devices), Programmable Logic Controllers (PLCs), ASICs (Application Specific Integrated Circuits), and FPGAs (Field Programmable Gate Arrays). While some of the devices result in "hardware", they are coded and programmed like software. The process of creating the "code" must follow good software development practices.

Safety critical software is:

- Software which controls or monitors safety critical functions including mitigation of hazards
- Software which runs on the same system as safety critical software or impacts systems which run safety critical software
- Software which handles safety critical data
- Software used to verify and validate safety critical hardware and software

Software that is used to mitigate (reduce) a potential hazard is safety critical. Examples include: software designed for redundancy of hardware or software that is performing a safety critical function; software that detects fault trends and responds; automatic response to failure (automatic safing); and software used for Caution and Warning.

Safety critical software must be connected somehow to a critical function. The software must be capable of affecting a safety critical function, a function that "must work" or "must not work". To prove this is or is not the case may take some analysis. Only the ability to *affect* a function in a manner that causes a hazardous system state is necessary to classify the software as safety critical.

Remember to consider software that can potentially interact with the safety critical software. While your non-critical code does not affect the critical code during normal operations (you believe)

, what happens if it fails? Can it overwrite safety critical data if a pointer is corrupted? Can a corrupted stack lead to a jump into a safety critical routine? These are the kinds of issues you need to consider when reviewing the non-critical software.

**Categories of S/W Safety Functions**

Classifying the software into these categories often helps define the problem and helps in creating the "solution". While these may not be the only categories of safety critical software, they do cover a wide range of the operational software you need to consider.

- ➔ **Must work/must not work functions.** These functions may be mode and state dependent. A function may be a "must work" only in a particular state. Some functions are "must never work" functions, and some will be "must never fail to work". The collision avoidance function in air traffic control software must always work (never fail).
- ➔ **Fault tolerance.** What level (or number of levels) of redundancy are needed? What division between hardware and software redundancy is best for the system?
- ➔ **Multiple Commanding (Ready, Arm, Fire).** Multiple commanding reduces the chance of accidentally sending a hazardous command. The software should be aware of the command sequence and reject out-of-sequence ready/arm/fire commands. Acknowledgements need to be sent at each stage, so that everyone (human and software) is on the same page.
- ➔ **Caution And Warning Functions.** Some functions monitor potential hazardous conditions and report that information to a human operator.
- ➔ **Failure Detection, Isolation, and Recovery/Restart/Reduced Operation.** When a fault or failure occurs, there are multiple ways to get back to a safe operating state. What the best way is (restarting the system, reducing the functionality of the system, rollback to a previously safe state, etc.) should be thought out early in the design.
- ➔ **Automatic safing software.** What is needed to automatically safe the system, and what conditions trigger this response. Consider if backup systems are required, and whether they must be hot, warm, or cold.
- ➔ **Autonomous Decision Making & Operation.** Some software will have total control, without human intervention. The Remote Agent experiment on the NASA Deep Space 1 probe was such an intelligent autonomous system.

## *Hazard Severity Definitions and Probabilities*

Information on hazard severity, likelihood, and risk indices are included in the class presentation, and will not be reproduced here. They are NASAs "best guess" at defining and categorizing hazards. Consider them a starting point for thinking about hazards and probabilities in your area of interest.

Definitions of hazard severity will differ between organizations, standards, and even products. You should determine a set of severity levels that are appropriate to your domain. In the same manner, you need to define the likelihood of occurrence. Neither of these sets of definitions are easy to quantify. You have to use your best engineering judgment. Be prepared for arguments and discussion.

Creating the definitions may not be too difficult, but applying them to the system and software is guaranteed to produce heated discussions. Everyone wants to believe that their system/function/program **cannot** fail. Try to be realistic when assigning probabilities, especially to software. The more complex the software is, the higher the probability of failure should be. If necessary, get input from those outside your immediate project, who may see things a bit more dispassionately. Be aware that management may have a strong desire to not have anything classified with a "bad" risk number (low value), and may put pressure on those making the determination.

## *Creating Safer Software & Safer Systems*

So how can you create software that is safer (and therefore results in a safer system)? First and foremost, follow a good software development process! While good process does not guarantee good software, a chaotic process is very likely to develop buggy, unreliable, and potentially unsafe software.

Consider the tools you select with a "safety eye". This includes the operating system chosen, language, and development tools.

Conduct reviews with diverse teams, to get the most "system" input possible. Software does not exist in isolation, and software developers need to develop a systems engineering approach to their work. Use Formal Inspections on critical products (requirements, design, code) and more informal reviews on non-critical areas.

Communication is a vital tool in creating safer software. Develop a good working relationship with the hardware engineers, the users/operators, and even management. Everyone makes assumptions that are obvious to them, but not to others. Everyone also forgets to tell others about a decision, change, or idea that has a system-wide impact. They may not even realize that the change *does* have a system-wide impact. So talk to people! The more you know and understand about the problems and issues of other areas, the easier it is for you to anticipate problems or changes.

Conduct a set of development and safety analyses that are appropriate for your level of hazard severity and risk. If your system could kill someone, do a thorough and complete job of analysis. If the risk is low (not probable or the severity is negligible), you can get away with a minimal set of analyses. These analyses are discussed later in conjunction with the various lifecycle phases.

Eliminate analyses that are not appropriate, based on cost, benefit, or applicability.

## *Don't Panic - Scoping and Tailoring a Software Safety Program*

Your software safety program does not have to be huge, expensive, and time consuming. The scope of the safety program, and the specific analyses and procedures to perform, are tailorable. You don't have to do everything, but you should do "enough".

Also remember that not all your software will need to be created at the same level of effort (process and analysis), if you are careful how you design it. The more isolated you make your safety critical code, the less you need to analyze the non-critical components. Keep not only code separate, but data as well.

The most important factor in scoping a software safety effort is **risk.** A low risk system will require much less safety effort than a high risk system. So it's important to do the up-front analysis and determine just how risky your software and system are!

What are those "up front" analyses? First and foremost is the Preliminary Hazard Analysis (PHA), which identifies *system* hazards. The PHA is performed by the safety folk during the requirements phase. If you're not lucky enough to have a safety engineer, most system safety books describe the procedure. Other analyses include Fault Trees and Failure Modes and Effects Analyses, both of which are described in detail in the NASA Software Safety Guidebook.

The class presentation takes you through system and software hazard and risk tables, until you arrive at a software risk classification. From there you move to one of three levels of software safety effort: full, moderate, and minimum. Within each category, you can further tailor the development and analysis effort to match your risk and your project.

### Safety Critical Software

Earlier, you were introduced to many examples of software that is considered safety critical. To answer the question of "how critical", consider these factors:

- **Degree of control.** Does the software control the hazardous operation? Does the software provide hazard prevention (monitoring, automatic safing, etc.)? If the software malfunctions, can it cause a failure or fault? Does it provide the only source of information for an operator to make a safety-related decision?

- **Complexity.** Number of Interfaces between software elements, software/hardware, and software/user. How many subsystems are controlled by software? Does the software contain interacting, parallel executing processes

- **Timing Criticality.**
- How fast does the software have to respond to a hazardous condition? How many processes have real-time requirements (safety related or not)?

The military (MIL-STD-882C) set up software control categories based on the degree of control the software has over the hazard. Level 1 (1A) has complete control over potentially hazardous hardware and there is no time for human intervention. Level 2 (IIA and IIB) has control or monitoring capability, but there is sufficient time for a human to intervene. The difference between A and B is whether the software controls the hazard (A) or monitors it (B). Level 3 (IIIA and IIIB) has safety related issues, such as commanding of hazardous hardware, but there are redundant independent safety measures to protect from failure. Level 4 (IV) has no control and no monitoring of hazards. This is "non-critical" software, except if it can fail in such a way as to impact the other safety critical software. If you have a catastrophic or critical hazard, all software needs to be evaluated for unintended interactions and failure modes that can lead to the hazard.

Using the Military's categorizations and the System Risk index, we can create a matrix for a basic determination of the level of effort for software development and safety analyses .

| Software Category | Hazard Severity | | | |
|---|---|---|---|---|
| | Catastrophic | Critical | Moderate | Negligible |
| IA<br>System Risk Index 2 | ■ | ■ | ■ | Minimum |
| IIA & IIB<br>System Risk Index 3 | ■ | ■ | ■ | Minimum |
| IIIA & IIIB<br>System Risk Index 4 | Minimum | Minimum | Minimum | Minimum |

**Full Software Safety Effort**

Systems and subsystems in this category have severe hazards which can escalate to major failures in a very short period of time.

Example systems include life support, fire detection and control, propulsion/pressure systems, and pyrotechnics and ordnance systems.

What does a full safety effort entail? Formal, rigorous Software Development for all code. Safety design features in the software. All or most of the Safety and Software Safety Analyses performed. Independent Verification and Validation (IV&V) and in house Software QA. The software may need to meet specific standards, such as FAA, military, or medical.

**Moderate Software Safety Effort**

Systems or subsystems in this category have either a limited hazard potential or, if they control serious hazards, the response time for initiating hazard controls to prevent failures is long enough to allow for notification of and response from human operators.

For a moderate software safety effort, a rigorous development process and safety analyses are only applied to the safety critical code. Not all safety analyses are required. Monitor non-safety critical code for potential impacts on safety critical areas. In house Software QA oversight.

**Minimum Software Safety Effort**

Systems in this category have either a very low hazard potential, or control of the hazard is accomplished by non-software means.

A minimum safety effort includes normal software development process, with extra attention given to safety related code. Minimal Software Safety analyses. Software safety as part of the development process, not a separate effort. Normal Software QA effort.

**What can you Tailor?**

Within a level of safety effort, you have the ability to tailor to meet your needs. Items you can tailor include:

- **The process.** What development process is best for a particular system, environment, schedule, cost? What reviews or inspections make sense.
  - **Method.** What design method will be used (structured, OO)?
  - **Life-cycle.** Waterfall, spiral, rapid prototype, and modifications of these.
  - **Tools.** What tools will be used, including simulators, code generators, and CASE tools.
- **The organization.** Who makes up the team, and who does what part. For example, use senior members of team to handle flow down of requirements and safety issues. Make sure there is good communication between systems, hardware, safety , software engineering, software QA, and project management.
- **Analyses.** What analyses make sense, when you consider your whole system and project? Eliminate analyses that are not appropriate, based on cost, benefit, or applicability. Decide early which ones you will do. Document your decisions. If the situation changes, you may want to consider doing some of the previously skipped analyses.
- **Inspections.** Determine how many inspections to have. If choosing to have only a few inspections, select the most critical documents or code to be inspected.
- **Reviews.** Select the number of formal reviews necessary to get adequate insight into the software development process.
- **Testing and Verification.** Modify the level of testing based on the criticality of the software under test.

## *Requirements phase*

Requirements are the "contract" between the software developer and the system/project/customer that if the software does X,Y, and Z, then everyone will be happy. They specify what the software/system must do; what it must not do; and performance, environmental, hardware and other constraints. Requirements should not include design solutions, but may contain restrictions that preclude certain possible designs.

Getting the requirements right (correct, complete, and verifiable) is the first step to creating safe and reliable software, but it is not an easy task. Sometimes just getting the requirements at all is difficult! But the majority of critical errors found in projects can be traced back to the requirements. The cost of fixing errors goes up exponentially the longer the error remains in the system.

In a "worst case" project, all requirements are given verbally, are vague, and change regularly. No one has even thought about whether there are any performance or safety issues. The hardware design changes every Monday. You have no idea who will be using the system or what is desired for a user interface. If you have a requirements document, it's so out-of-date that no one looks at it.

What can you do on your own to help correct the requirements problem? *You need to create your own requirements document.* **Don't panic** – this is not rocket science! You do not need to have a formal document structure. At a minimum, have a title, author (that's you) date, and a short description of the system that the software will control. Put this document under configuration management/version control. A requirements "track record" can be helpful in educating your boss in the cost of frequent changes.

- *Write down what you know!* Everything your boss, the systems team, or the engineers told you is a good place to start. Keep copies of appropriate emails. Take notes during meetings. Put your understanding of what they want down on paper.

* *Add your assumptions.* There are always assumptions, and they may well come back to bite you! Put everything you can think of on paper.
* *Research the hardware.* As embedded programmers, you need an awareness of the hardware idiosyncrasies. These will act as constraints on your software.
* *Think about the users (or better yet, get their help).* Ask yourself who the users will be. Are they expecting a fancy interface? What information do they need to see, and how can you make it obvious. Sketch out your interface or throw together a quick prototype.
* *Take the next step...think about safety.* How can your device/system mess up. How can the software help it mess up, and how can it prevent the system from screwing up. Error detection and handling. Safety requirements – standards, organizational. System hazards.
* *Get "sign off" or approval on your requirements from all parties involved.* Don't just show the requirements to your manager. Make sure hardware, operations, the customer, anyone else involved, agrees.

When requirements change, it is important to document the modifications and the reasons behind them. Make sure you get "sign off" on the change from the initiator. Update the requirements document, so that anyone on the project can understand what is being created. Try to keep interfaces unaltered to prevent changes propagating throughout the system. Make sure your management understands the true cost of the change, and the risks it creates. And *re-evaluate the software safety!* Changes may lead to new hazards, or may impact safety critical code.

A series of analyses should be performed during the requirements phase. *Don't Panic* when you see "Analysis". Many analyses are just applying common sense in an organized manner. You don't have to be a mathematical genius to perform these analyses. For example, "flow down analysis" is simply a structured method to make sure requirements from a variety of sources are included in the software requirements.

The Preliminary Hazard Analysis (PHA) is a way of determining what aspects of the system could cause hazards. Software Hazard Analysis looks at how software relates to the system hazards, as a cause, control, mitigation, warning, etc. Procedures for the PHA and other hazard analyses are found in system safety books.

**Software Requirements Flow-down Analysis** is a method of making sure that all necessary requirements are included in the software requirements. Software requirements come from many sources, including the system specification, safety standards, regulatory requirements, hazard analyses, system and software risk analyses, and software standards and guidelines. Other sources are organizational or program requirements (e.g. International Space Station requirements), hardware and environmental constraints, customer input, and generic software safety requirements. Its important to know how to get hold of these requirements and analyses. This means knowing where to go, who to contact.

The purpose of a software requirements flow-down analysis is to make sure that **all** software requirements are captured, and that they are **correct, consistent, unambiguous** and **verifiable**. It doesn't do any good to have a lot of vague software requirements that no one really understands. If the requirement isn't clear, various people will interpret it differently. If it is not verifiable, how will you know if it has been met?

Consider the characteristics of the system and make sure they are included in the software requirements. Characteristics to be considered include at a minimum:

- specific limit ranges
- out of sequence event protection requirements
- timing
- relationship logic for limits. Allowable limits for parameters might vary depending on operational mode or mission phase. (Expected pressure in a tank varies with temperature.)
- voting logic
- hazardous command processing requirements
- fault response
- fault detection, isolation, and recovery
- redundancy management/switchover logic. What to switch and under what circumstances.

Generic software safety requirements are a distillation from various sources, including common software safety problems, lessons learned, and problems and issues with similar systems. Generic requirements prevent costly duplication of effort by taking advantage of existing proven techniques and lessons learned rather than reinventing techniques or repeating mistakes. There are many sources of generic software safety requirements: UL, NSTS 19943 Command Requirements And Guidelines for NSTS Customers, Marshall Spaceflight Center, the Food & Drug Administration, etc.

While you should start with a generic list, modify it (add, delete, change) for your specific domain. Think of them as generic shopping lists where you choose what is needed for a particular project. When you pick and choose what is needed for a particular project, document the reasons for selection/exclusion. Requirements can be tailored/scaled back to the level of the project.

The **Requirements Criticality Analysis** determines how critical each requirement is. It identifies additional potential system hazards that the system PHA did not reveal and potential areas where system requirements were not correctly flowed to the software. Identified potential hazards are then addressed by adding or changing the system requirements and reflowing them to hardware, software and operations as appropriate.

You perform the Requirements Criticality Analysis by doing the following:

- All software requirements are analyzed to identify additional potential system hazards that the system PHA did not reveal. A checklist of PHA hazards makes it easier to identify PHA-designated hazards that are not reflected in the software requirements, and new hazards missed by the PHA. In addition, look for areas where system requirements were not correctly flowed to the software.
- Review the *system* requirements to identify hardware or software functions that receive, pass, or initiate critical signals or hazardous commands. Also FDIR (fault detection, isolation, recovery) and fault tolerance requirements.
- Review the *software* requirements to verify that the functions from the system requirements are included. In addition, look for any new software functions or objects that receive/pass/initiate critical signals or hazardous commands.
- Look through the *software* requirements for conditions that may lead to unsafe situations. Consider conditions such as out-of-sequence, wrong event, inappropriate magnitude, incorrect polarity, inadvertent command, adverse environment, deadlocking, and failure-to-command modes.
- Analyze the *software* requirements for **must work** and **must not work** functions, especially those with safety implications.

The evaluation will determine whether the requirement has safety implications. Make sure you have agreement between safety, software, and management for a "safety critical" designation. These requirements are then placed into a tracking system to ensure traceability of software safety requirements throughout the software development cycle from the highest level specification all the way to the code and test documentation.

Keep in mind that not all "safety critical" requirements are created equal. At this stage, however, it's best to look at everything that can cause a safety problem, even a trivial one. You can remove a requirement with less effort and cost than adding one in later.

The Criticality analysis will not be able to be completed at this time. Only the requirements can be analyzed. As the design progresses, it is important to revisit this analysis.

**Checklists** are a tool for making sure you haven't forgotten anything important, while doing an analysis or reviewing a document. They are a way to put the collective experience of those who created and reviewed the checklist to work on your project. They are a starting point, and should be reviewed for relevance for each project.

**Cross references** are matrices that list related items. A matrix that shows the software related hazards, hazard controls and their corresponding safety requirements should be created and maintained. This should be a living document, reviewed and updated periodically. Refreshing your mind on the hazards that software must control while working on the software design, for example, increases the likelihood that the hazard controls will be designed in correctly. Another cross reference matrix would list each requirement and the technique that will verify it (analysis, test, etc.).

A hazard requirements flowdown matrix should be developed. This matrix should map safety requirements and hazard controls to system/software functions and to software modules and components. Where components are not yet defined, map to the lowest level possible and tag for future flowdown.

**Timing, throughput and sizing analysis** should be performed for all functions, not just safety critical functions. This analysis evaluates software requirements that relate to execution time, transmission time and amount, and memory/disk allocation. Typical constraint requirements are maximum execution time and maximum memory usage.

You should evaluate the adequacy and feasibility of safety critical timing, throughput and sizing requirements. For example, are the low level temperature readings likely to overload the system so that it can't see or process safety critical functions? Are adequate resources allocated in each case, under worst case scenarios. Will I/O channels be overloaded by many low level or error messages, preventing safety critical features from operating (a non-critical function may be causing this problem). Evaluate resource conflicts between science data collection and safety critical data availability.

Investigate time variations of CPU load, determine circumstances of peak load and whether it is acceptable. Consider multi-tasking effects. Note that excessive multi-tasking can result in system instability leading to "crashes".

**Formal Inspections** were developed by Michael Fagan at IBM to improve software quality and increase programmer productivity. As such, the formal inspection process involves the interaction of the following five elements: 1) Well-defined inspection steps and rules; 2) Well-defined roles for participants (moderator, recorder, reader author, inspector); 3) Formal collection of process and product data (metrics); 4) The product being inspected must meet certain input criteria; and 5) A supporting infrastructure of management and standards is needed.

Formal inspections are not 'formal' milestone reviews, such as a "Critical Design Review". They are conducted by peers, on partial or complete products, such as documents, processes, procedures, design, or code. Each member of a team has a real interest in the software product, and are involved because of their unique perspective. The inspection is a tool to help the author identify problems, not a critique of style or design.

Walkthroughs or informal reviews are similar, but they are usually unstructured and only include software developers. The strengths of formal inspections is that a wide variety of areas are included and that the formal process keeps everyone focused and on track.

While inspections can be used on anything and everything, they are time and resource consuming. Wise choices of what to inspect, and when, is important.

## *Design phase*

The design process includes identification of design features and methods (object/class choice, data hiding, functional distribution, etc.), including safety features (e.g., inhibits, traps, interlocks and assertions) that will be used throughout the software to implement the software requirements. Safety Critical Computer Software Components (SCCSCs) can now be identified. These SCCSCs contain safety critical code or interact with other safety critical components. Safety Critical components should be flagged for additional design attention.

During this phase, test plans and procedures are created. Make sure that safety testing is included, either as separate tests or as part of a system or subsystem test. Look at the information necessary to verify the safety requirement to determine when that requirement should be tested.

**Tools** (language, compilers, operating systems, CASE, etc.) are usually selected during the design phase. Keep safety in mind when choosing. For languages, look at features that help make it "safer", such as strong data typing, control of pointers (or no pointers) and support for modularity. What error handling features does the language support. Being comfortable with the idiosyncrasies of a language is important. Find or create checklists of common errors for the language chosen, and review the list regularly. Use the checklist during inspections or reviews. Add your own "personal" bugs to the checklist.

When choosing an **Operating System** (OS), consider the issues of memory management (MMU usage, memory protection between tasks), determinism (meet deadlines, minimize time jitter, bounding of priority inversion), priority inversion (a temporary state used to resolve resource conflicts that can lead to a low priority task keeping a high priority task from running), context switching time (time to save task, make next task ready to run), interrupt latency (how fast can interrupts be serviced), scheduling method (priority-based preemptive, "Round Robin" time slice, or "cooperative"), synchronization methods (semaphores, messages, etc. – consider the time each method takes to complete), error handling in OS system calls (error code, exceptions, user-created error handler), safety certification.

When deciding on a **compiler**, consider: is there a list of defects (bugs) in the compiler? Can it treat warnings as errors, and can you set that up as the default? Can you use "stubs" to eliminate code you don't need? Is the source code available, so that unneeded code can be stripped out, or for formal verification or inspection?

In addition to "the basics", these tools are very useful in creating good (and safe) code:

- Lint – finds problems in the code that compilers might miss. Not everything is a true problem, but should be evaluated. If it's "non standard", treat it as an error!
- Profiler – checks speed of program. Good for finding routines that take the most time. Points to areas to where optimization may be useful.
- Memory check programs – find memory leaks, writing outside of bounds.

**CASE** tools can be very helpful during the design and coding phases. Some can even generate code from the design. **Don't trust them for safety critical code!** Review automatically generated code closely, both safety-critical sections and non-critical sections (for interactions with critical). The maturity of these CASE tools varies.

**Configuration Management** (CM) is a very important process. It involves more than just code control and versioning. All documentation (requirements, plans, procedures, design, etc.) should also be under CM control. Select CM tools that can handle binary files (documents) as well as code. Or have two separate systems (one for code, one for everything else). For future understanding and for traceability, it is very important to document what you changed when you check something in. "Just compare to the previous version" is a pain in the behind when you have to trace back several changes to see what lead to the bug you've found.

Everyone makes mistakes. But once a bug is found, it should not come back to bite you later in the development cycle! **Defect tracking** makes sure you haven't forgotten a problem. With unresolved problems, it gives you information if a similar problem occurs down the road or in a similar program. Keeping track of the defects helps you see patterns that can be used to improve the design, coding, or development process.

**COTS software** comes with a lot of baggage, most of it unknown. When safety or reliability is an issue, tread lightly. Consider carefully before committing to COTS software. This includes your OS. Remember that s

ystems vary. The COTS software may work perfectly in one system and fail miserably in another. Software with bug-fixes every week is definitely not stable enough for safety-critical systems. Also consider upgrades - how different are the versions? Can you stick with the tried-and-true?

Do you get more than a users manual? Can you get design info, testing reports, or source code? Can you talk to the team that built it, if you get into deep trouble? Source code may be needed for inspection or analyses. You may also need to modify the software to work in the system, or for safety reasons. Will you still be able to get support if you do so? Do you know anything about the team that created the software? Did they follow a standard process? What is their track record for quality software?

What error handling does the software have (error codes, exceptions). Is there information on how it can fail or what its limits are (stress, load, etc.)? Can you get a list of known defects? Check user groups and web-sites, though always consider the source.

Do you need to add functionality (glueware)? How will extra functionality affect the system resources? What happens if any of those functions are accidentally executed?

**A Good Design Process**

starts with communication. It is critical to assure that the software design isn't built around missing, misunderstood, or changing hardware or operational specifications. Have Formal Inspections, reviews, or walkthroughs of design products. Update the list of risks, now that more details are present. Use simulations and prototypes to "nail down" performance issues or interface designs.

Implement a problem tracking and reporting system (defect tracking). Use configuration management on your design products. Follow a standard process (company, IEEE, military, etc.). Remember to document changes, and propagate requirement changes back to the software specification.

When designing **Safety Critical Systems**

, consider the following areas:

- Fault/Failure Detection, Isolation and Recovery (FDIR)
- Fault Tolerance/Defensive Programming
- Must Not Work Functions
- Must Work Functions
- Critical Interfaces/Commands
- Critical Data
- Coding Standards

Fault/Failure Detection, Isolation, and Recovery strategies determine when there is a problem and provide an appropriate response to the situation. "Fault" FDIR stops the problem at a low level before it cascades into a failure. "Failure" FDIR recovers the system to a safe state (if possible). To detect faults or failures, the system needs a way to determine there is a problem! Consider whether the software is getting enough information (sensors, etc.) to detect faults or failures. Isolation prevents the fault/failure from propagating. During design, consider what level to contain the fault/failure at. To initiate a recovery scheme, the system must be "smart" enough to know what to do. Recovery could be to request retransmission, switch to alternate storage location if the first is full, or return to a known safe state if the failure is severe enough. The recovery mode could also be restoration to a properly operating state, or reduced functionality in the problem area until human intervention.

It is important to define what functions will be Must Work and which will be Must Not Work, and for what states. A function could be MWF in one state (valve closed during experiment) and MNWF in another state (valve must not be closed when venting in post-experiment state), or vice versa. States and/or modes of operation need to be identified before the MW and MNW functions can be determined.

What interfaces allow software to influence safety critical hardware or to pass safety critical information? Can the state of a critical interface be saved between reset/power cycles (preserve state)? Consider hardware interfaces - how can the software recognize and deal with failed sensors, malfunctioning actuators, or communications problems. What types of data collision must be guarded against? What happens when things get mixed up?

What commands can initiate or stop actions that involve hazard causes or controls? Consider ready/arm/fire sequences for safety critical commands. Independent commands, coming from different sources that must give the command to proceed within a specified time, is another way to prevent accidental initiation of a hazardous command. Make sure operator verification dialogs ("do you really want to...") default to **no** for critical commands, so that the operator must *think* before sending the command. Make sure one keystroke cannot initiate the command. Consider protection from out-of-sequence critical command. "Commands" also include function calls as well as operator initiated instructions.

Determine what is critical data and isolate it from "regular" data. Consider maintaining multiple copies, storing it in memory that is not cleared by reset/powerup, and performing reasonableness checks before use.

Other critical items to consider during the design are input/output timing; the effects of multiple events, out-of-sequence events, failure of event, and wrong event; inappropriate magnitude of value; incorrect polarity; adverse environment; deadlocking in a multitasking system, and hardware failures.

Coding standards (not style) can be used to limit certain constructs, library functions, or language elements that should not be used. Create them from lessons learned, previous projects, and information on the chosen language/compiler.

When designing your software, always

***Keep Safety in Mind.*** Don't wait until the design is complete to incorporate safety features. Look at the safety requirements and what the safety analyses tell you about the system, and feed this back into your design. Write good verification plans that target the safety requirements and thoroughly test the safety critical modules, interfaces, and data. However, i

n today's development environment, where cost is everything, you can't do it all! You need to focus the safety effort to the areas where it is most needed.

**Design Logic analysis** determines the best flow for the control logic, what equations and algorithms are needed to bring this about, and what the control hierarchy must be.

**Design Data Analysis** considers: Data needed within and without the system needs to be identified by source, destination, manipulation required, temporary and permanent storage, volume, speed, etc. Check for safety critical data sharing the same memory area as non-critical data, or being accessed by functions that are designated non-critical.

**Design Interface Analysis**: Look at all interfaces (software to software, software to hardware, hardware to software, software to humans). Interface characteristics to be addressed should include interprocess communication methods, data encoding, error checking and synchronization. The analysis should consider the validity and effectiveness of checksums, CRC's, and error correcting code. The sophistication of error checking or correction that is implemented should be appropriate for the predicted bit error rate of the interface.

**Design Constraint Analysis:** where the timing, sizing, throughput factors are determined for the system hardware and required performance levels. Also consider equations and algorithms limitations, input and output data limitations (e.g., range, resolution, accuracy), sensor/actuator accuracy and calibration, noise, EMI, digital word length (quantization/roundoff noise/errors), human factors, human capabilities and limitations, physical time constraints and response times, off nominal environments (fail safe response).

**Design Traceability Analysis:** assure the design incorporates requirements and map all safety critical requirements to a design element.

**Complexity:** understand the level of complexity of your design. Less complexity is usually better (less errors), but not always. Look at the complexity of individual components, and the complexity of the interactions.

**Independence/Interdependence:** Look at the design to verify that safety critical functions are independent of non-critical functions. Conversely, look at the non-critical functions to see how they may inadvertently affect the safety critical functions, via shared data, potential function calling sequence, communication, or other mechanisms.

**Fault Tree:** A top-down analysis applied to the system or software. Looks at potential failures, and traces back to find what could cause the problem. Can be started with the architectural design and updated as the design progresses.

**FMEA:** A bottom-up analysis that looks at each component and how it can fail, then follows the failure through the system to see if a hazard/failure can occur. Needs at least the detailed design. Time consuming, so don't use for all components! Can be combined with FTA.

**Software Safety Tasks** include updating the hazard analyses, checking for new hazards, proper implementation of hazard controls, and any areas of concern. The software design should be looked at by a safety engineer, or a software developer wearing a "safety hat". Verify that all safety requirements are included in the design. This is where a traceability matrix really helps! Also create a list/matrix of the Safety Critical Computer Software Components. They warrant further analysis throughout the design, implementation and testing phases.

## Coding for Safety

You've created (and maintained) requirements that include safety features, and those features have been designed into the software system. Now you get to translate that wonderful design into actual code. Your primary goal at this phase is to develop complete and correct code. Complete means that all the design elements, including safety features, are actually implemented. Correct means that you did not add any bugs or faults.

Managers and designers must communicate all issues relating to the program and modules they assign to programmers. Safety-critical designs and coding assignments should be clearly identified. Programmers must recognize not only the explicit safety-related design elements but should also be cognizant of the types of errors which can be introduced into non-safety-critical code which can compromise safety controls. Coding checklists should be provided to alert for these common errors.

*Coding Checklists* are a tool to keep from making stupid mistakes. They are a way to put the collective experience of those who created and reviewed the checklist to work on your project. Coding Checklists should include common mistakes (both generic and language-specific), library or language elements to avoid using, and proactive programming techniques to consider (such a defensive programming). The coding standards (or at least the most important elements of the standards) should be included in this checklist. Individuals can customize a project coding checklist with their personal common errors.

A Safety Checklist tracks the safety requirements, and is used to verify that safety requirements identified earlier in the design process have, in fact, been flowed into the software code. This is not the time to "drop the ball" and forget about a safety requirement! The Safety Checklist also helps the programmer find possible deficiencies in the design, where a safety feature was not implemented or implemented poorly.

**Updating requirements**

Often during the development phase, missing or incorrect requirements are identified, or new system requirements are added to software. Hopefully, fault detection and recovery have already been included, though these requirements are known to sometimes appear at this phase of development. It may become apparent that various capabilities were assumed, but not explicitly required, and so were not implemented. Hardware may have been identified that couldn't perform a function it was originally supposed to. This is called requirements creep.

It is prudent to make sure there is a control board which agrees to changes. This is where a lot of problems/hazards can insert themselves. An elegant design may be crippled by various unexpected design changes that don't allow time to rework the entire design properly. Instead, coding kluges are put in to accommodate these changes.

Once missing or misunderstood requirements are identified, they must be incorporated by "back-filling" (updating) the requirements specifications prior to implementation in order to maintain proper configuration control. Too often the design is updated but not the requirements, because there isn't enough time or it's too much work. "The design says it all" may be true for the designers, but it is certainly *not* true for the testers or maintainers who use the requirements as their starting point. If the testing group defines system tests based on out-of-date requirements, important elements of the system may not be properly tested. Spend the minutes and keep the requirements in sync with the design and code.

**Defensive Programming**

Defensive Programming is a set of techniques to limit the possibility of faults becoming failures. They incorporate a degree of fault/failure tolerance into the code, sometimes by using software redundancy or stringent checking of input and output data and commands. An example of defensive programming is sometimes called "come from" checks. Critical routines have multiple checks in them to test whether they should be executing at some particular instant. One method is for each preceding process to set a flag in a word. If all the proper criteria are met then the routine in question is authorized to execute.

Some of this should have been required and designed in, but it is unfortunately often left to the programmers to come up with on their own. If you need to add some fault/failure tolerance into your code, consider the following:

- **Range and Limit Checking.** Check that a variable falls within the acceptable range of values. You need to decide how to handle values that are outside the limits. What you decide to do may have safety implications.
- **Reasonableness Checks.** A value may be in range but still be unreasonable. If there are multiple sensors reading the same physical component, they should all agree within a certain amount.
- **Input verification/validation (per mode/state).** What an input value or sequence should be may vary by the state or mode of the program.
- **Communication checks (ack/nak, CRC, etc.)**
- **Redundancy w. functions, data, sensors, etc.** Make sure you are not relying on one element (function, copy of the data, single sensor) for safety critical decisions. Sensors can go bad, data can be corrupted, and functions can have bugs.
- **Background memory checks.** Check both the physical memory (is it working correctly) and look for corruption by malfunctioning software. You may want to limit this to checking safety critical data areas only.
- **Start-up self checks.** Make sure your CPU is functioning properly and all hardware is up-to-speed before committing to operation.
- **Inhibits.** Prevent certain functions from being accessed. The inhibits will likely vary by mode or state. You don't want the vent valve open when the chamber is filled with toxic gas, for instance.
- **Multiple commanding** (e.g. Ready/Arm/Fire).

**Unit Testing**

Unit level testing begins during the software implementation phase. Units and modules often cannot be thoroughly tested during integration because individual module level inputs and outputs are no longer accessible. Unit level testing can identify implementation problems which require changes to the software. For these reasons, unit level testing must be mostly completed prior to software integration. Unit testing is covered in more detail in the testing section

**Coding Analyses**

Development and Safety analyses need to be done on the completed (or nearly so) code. These analyses make sure that the code has implemented the design and requirements correctly. They also check constraints that may not have been testable earlier. In addition, safety analyses verify that no new hazards have been added and that the code correctly implements the required safety features.

**Code logic analysis** evaluates the sequence of operations represented by the coded program. It detects logic errors in the coded software. This analysis is conducted by performing *logic reconstruction*, *equation reconstruction* and *memory decoding*. *Logic reconstruction* entails the preparation of flow charts from the code and comparing them to the design material descriptions and flow charts. *Equation reconstruction* is accomplished by comparing the equations in the code to the ones provided with the design materials. *Memory decoding* identifies critical instruction sequences even when they may be disguised as data.

**Code Data Analysis:** Data analysis focuses on how data items are defined and organized. This is accomplished by comparing the usage and value of all data items in the code with the descriptions provided in the design materials. Of particular concern to safety is ensuring the integrity of safety critical data against being inadvertently altered or overwritten.

**Code interface analysis** verifies the compatibility of internal and external interfaces of a software component. Each of these interfaces is a source of potential problems. Check that parameters are properly passed across interfaces. Interface characteristics to be addressed should include data encoding, error checking and synchronization. The analysis should consider the validity and effectiveness of checksums and CRCs.

**Code Constraint Analysis.** What are the real constraints on the system, particularly with regards to timing, sizing, and throughput. It is vital to ensure that computing resources and memory available are adequate for safety critical functions and processes.

**Unused Code Analysis.** A common real world coding error is generation of code which is logically excluded from execution. Such code is undesirable for three reasons; a) it is potentially symptomatic of a major error in implementing the software design; b) it introduces unnecessary complexity and occupies memory or mass storage; and c) the unused code might contain routines which would be hazardous if they were inadvertently executed (e.g., by a hardware failure or by a Single Event Upset).

**Interrupt Analysis.** Interrupt Analysis looks at the effect of interrupts on program flow and data corruption. Can interrupts lead to priority inversion and prevent a high priority or safety critical task from completing? If interrupts are locked out for a period of time, can the system stack incoming interrupts to prevent their loss? Can a low-priority process interrupt a high-priority process and change critical data?

**Measurement of Complexity.** As a goal, software complexity should be minimized to reduce likelihood of errors. Complex software also is more likely to be unstable, or suffer from unpredictable behavior. Complexity can be measured via McCabe's metrics and similar techniques. Refactoring is sometimes used to reduce complexity while maintaining code that is functionally identical. Be careful, however, as the refactoring may invalidate previous analyses or inspections.

**Underflows/Overflows.** Underflows/overflows in certain languages (e.g.., ADA) give rise to "exceptions" or error messages generated by the software. These conditions should be eliminated by design if possible; if they cannot be precluded, then error handling routines in the application must provide appropriate responses, such as retry, restart, etc.

**Update previous analyses.** Previous analyses, including FMEA (Failure Modes and Effects Analysis) should be updated, now that the "real thing" is available.

## *Testing for Problems (Bugs, Errors, and Faults)*

**Unit Testing** is usually performed by the developer, or another member of the development team. A "unit" is a flexible and variable term. Units can be modules, classes, collection of modules, tasks, CSCI, etc. A unit should be a logical division that is "big enough" to be tested but "small enough" to allow access to the innards of the software. However, don't make a "unit" the entire program!

Unit testing uses stubs and drivers to simulate the rest of the software. Various types of tests can be performed, including White Box testing (path, statement, memory, safety), Black Box testing (typical, extreme, combinations of inputs), and Safety testing (e.g. Error Handling - make sure this works).

White box includes:

* **Path testing** - verify each path through the unit is tested.
* **Branch testing** - each branch (case, if) is taken. Usually part of path testing.
* **Loop testing** - each loop is exercised, especially at the boundaries (0, 1, typical,max-1, max, max+1 times).
* **Statement execution test** - verify every statement is executed at least once.
* **Memory locations** - verify that data is where it is supposed to be
* **Safety tests** - verify that safety critical units meet the safety requirements. Perform any safety tests that require access to the code (instrumentation) at this time.

Black Box tests include typical sets of inputs, extreme values of inputs, combinations of inputs (some at extremes, others typical; or all at extremes), and all sensor modes tested

**Unit Test "Best Practices"**

Design the test and write down the procedure, including the files used for stubs and drivers, *before* you test. Don't "wing it", because you are sure to miss something you really needed to test. You may also test more than you need to in certain areas. Make sure the stubs and drivers are under some sort of configuration control.

Review your software using a coding checklist prior to testing it. Keep a list of errors found in your code. Feed this information back into your coding checklist. Share your bug list with your team, if you won't be penalized for it.

**Testing COTS Software**

Commercial Off-the-shelf (COTS) software comes with a lot of questions attached. Whatever its pedigree or amount of testing, remember that the software was not specifically designed for your system. Ariane 5 demonstrated quite dramatically that perfectly good software (from Ariane 4) failed to function correctly *in the new system.*

Test for ways COTS software can fail. How does your software respond to those failures? Look for ways that the OTS software can influence the safety critical code, either during normal functioning or in a failure mode. Do as much stand-alone testing prior to integration to characterize the COTS software. Ways that COTS software can affect the system include: overwriting a memory location where a safety critical variable is stored; getting into a failure mode where the OTS software uses all the available resources and prevents the safety critical code from executing; inadvertent execution of a dormant, undesired function; or clogging the message queue so that safety critical messages do not get through in a timely manner.

**Integration Testing**

Software integration is putting the "puzzle" together. Integration tests are primarily "black box" tests. This is the time to test the interfaces, functionality, error handling, and performance. Safety tests that need access to software details should be done now (if not at unit testing). *Test for interactions between modules, especially those designated safety critical.*

**System Tests**

System tests comprise a collection of tests that exercise various aspects of the completed system (hardware and software).

- **Functional/Acceptance Test.** A functional test verifies that the software works as expected (meets the requirements) and verifies "must work" and "must not work" functions. The system is operated in nominal manner and environment. Unclear or incomplete requirements are often caught here, when the customer first sees the system. A Functional test is often used as the customer Acceptance Test.

- **Performance/Load Testing.** Performance tests include *capacity* (number of records, users, disk space, memory usage, etc.), *accuracy* (verification of algorithm results and precision), *response time, availability* (Percentage of time the system can be used), *throughput*, and *load* (can it handle the predicted load for the required period of time) tests.
- **Stress Testing.** See how much the system can handle before it "breaks". Aspects that can be stressed include: CPU usage, sensor input or output rates, telemetry rates, memory utilization, amount of available memory, and network utilization. Adequate margin between the system's peak usage and the breakdown point is desired.
- **Stability/Reliability Testing.** How well does the system operate for extended periods of time? The system is run in normal mode of operation, though occasional peak performance is allowed. You are not stress testing the system. Check if the system can handle intermittent bad data. Is there a sensitivity to event sequences? Does memory leakage cause problems after a period of time?
- **Safety Verification Testing.** Safety tests are designated for each hazard control. Verify partitions, firewalls, or other software constructs that isolate safety critical code. "Fail" the hazard controls in a multi-tolerant system. For example, in a two-fault-tolerant system (three controls), try all combinations of two failures. Verify hazardous commands. Verify software correctly handles out of sequence commands, hazardous commands issued in an incorrect state, and other possible errors. Software Safety (usually SQA) should witness all software safety testing.
- **"Oops" Testing.** Tests include *resistance to failure* (how gracefully does the system respond to errors?), *disaster* (What happens if you pull the plug? Does the system "fail safe"?), and *"Red Team"* (unstructured test where users try to break the system).
- **Other tests specific to environment.** Installation, Compatibility, and Conformance to standards.

## *Where do you go from here?*

The class presentation contains a section on programming "best practices" that lists many *Things to Do, Things **not** to Do*, and *Things to Think About* when creating safety critical software. The lists are drawn from a variety of sources, including NASA International Space Station standards. But the most important factors in creating safe software include:

- Get the requirements right!
- Use Configuration Management!
- Always think of the whole system, not just your little piece of it.
- Think Safety.
- Document *what* you are doing and *why* you are doing it! You will not remember when asked 6 months from now!
- Document interfaces and keep them up-to-date!
- Understand your hardware and purchased software (OS). Know how they can fail, throw exceptions, or impact your software.
- Design error/failure detection and handling in from the start - not as an afterthought
- Track your bugs, so they don't come back and bite you later. This also helps you remember your mistakes, so you won't make them again on the next project.
- Plan and script all tests. Don't just bang at the code. Know what you are testing, how much to test, and when to stop.
- Analyze your test outputs. Don't get complacent, or you may miss something vital. Automate the analysis if possible.
- Perform load, performance and stress tests. Know that your system can handle what it needs to, work as fast as necessary, and handle all the input/output it must deal with. Also know where the system's limits are so you can keep away from them.

The class presentation also contains several pages of **resources**, including standards, books, and websites.

The NASA Software Safety Guidebook, upon which much of this class is based, is currently out for review. Please contact the presenters for a copy of the Guidebook.

**Software Safety is everyone's business. As developers and engineers, you can make a positive difference. Help solve the "software safety crisis" one project at a time.**