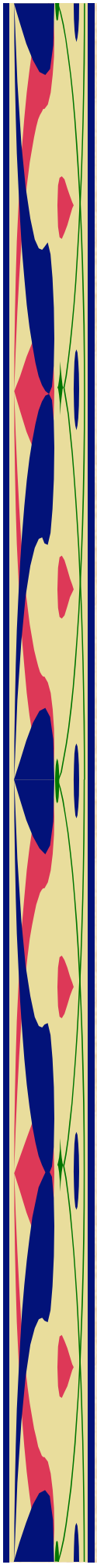
Joint Software System Safety Committee

SOFTWARE SYSTEM SAFETY HANDBOOK

A Technical & Managerial Team Approach

December 1999



This Handbook
was funded and developed by the

Joint Services Computer Resources Management Group,
U.S. Navy,
U.S. Army,
and the U.S. Air Force

Under the direction and guidance
of the

Joint Services Software Safety Committee

of the
Joint Services System Safety Panel
and the
Electronic Industries Association, G-48 Committee

AUTHORS

David Alberico	Contributing (Former Chairman)
John Bozarth	Contributing
Michael Brown	Contributing (Current Chairman)
Janet Gill	Contributing
Steven Mattern	Contributing and Integrating
Arch McKinlay VI	Contributing

Acknowledgements

This Handbook represents the cumulative effort of many people. It underwent several reviews by the technical community that resulted in numerous changes to the original draft. Therefore, the contributors are too numerous to list. However, the Joint Services Software System Safety Committee wishes to acknowledge the contributions of the contributing authors to the Handbook. Special thanks to **Lt. Col. David Alberico**, USAF (RET), Air Force Safety Center, Chairperson of the JSSSSC, from 1995 to 1998, for his initial guidance and contributions in the development of the Handbook.

The following authors wrote significant portions of the current Handbook:

John Bozarth, CSP, EG&G Technical Services, Dahlgren, VA
Michael Brown, Naval Surface Warfare Center, Dahlgren Division,
(Chairperson, JSSSSC, 1998 to Present)
Janet Gill, Naval Air Warfare Center, Aircraft Division, Patuxent River, MD
Steven Mattern, Science and Engineering Associates, Albuquerque, NM
Archibald McKinlay, Booz-Allen and Hamilton, St. Louis, MO

Other contributing authors:

Brenda Hyland, Naval Air Warfare Center, Aircraft Division, Patuxent River, MD
Lenny Russo, U.S. Army Communication & Engineering Command, Ft. Monmouth, NJ

The committee would also like to thank the following individuals for their specific contributions:

Edward Kratovil, Naval Ordnance Safety and Security Activity, Indian Head, MD
Craig Schilders, Naval Facilities Command, Washington, DC
Benny Smith, U.S. Coast Guard, Washington, DC
Steve Smith, Federal Aviation Administration, Washington, DC
Lud Sorrentino, Booz-Allen and Hamilton, Dahlgren, VA
Norma Stopyra, Naval Space and Warfare Systems Command, San Diego, CA
Dennis Rilling, Naval Space and Warfare Systems Command, San Diego, CA
Benny White, National Aeronautics and Space Administration, Washington, DC
Martin Sullivan, EG&G Technical Services, Dahlgren, VA

This Handbook is the result of the contributions of the above mentioned individuals and the extensive review comments from many others. The committee thanks all of the authors and the contributors for their assistance in the development of this Handbook.

TABLE OF CONTENTS

D. COTS AND NDI SOFTWARE

E. GENERIC REQUIREMENTS AND GUIDELINES

LIST OF FIGURES

LIST OF TABLES

1. Executive Overview

Since the development of the digital computer, software continues to play an important and evolutionary role in the operation and control of hazardous, safety-critical functions. The reluctance of the engineering community to relinquish human control of hazardous operations has diminished dramatically in the last 15 years. Today, digital computer systems have autonomous control over safety-critical functions in nearly every major technology, both commercially and within government systems. This revolution is primarily due to the ability of software to reliably perform critical control tasks at speeds unmatched by its human counterpart. Other factors influencing this transition is our ever-growing need and desire for increased versatility, greater performance capability, higher efficiency, and a decreased life cycle cost. In most instances, software can meet all of the above attributes of the system's performance when properly designed. The logic of the software allows for decisions to be implemented without emotion, and with speed and accuracy. This has forced the human operator out of the control loop; because they can no longer keep pace with the speed, cost effectiveness, and decision making process of the system.

Therefore, there is a critical need to perform system safety engineering tasks on safety-critical systems to reduce the safety risk in all aspects of a program. These tasks include the *software system safety (SSS) activities* involving the design, code, test, Independent Verification and Validation (IV&V), operation & maintenance, and change control functions of the software engineering development process.

The main objective (or definition) of system safety engineering, which includes SSS, is as follows:

"*The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle.*"

The ultimate responsibility for the development of a "safe system" rests with program management. The commitment to provide qualified people and an adequate budget and schedule for a software development program begins with the program director or program manager (PM). Top management must be a strong voice of safety advocacy and must communicate this personal commitment to each level of program and technical management. The PM must support the integrated safety process between systems engineering, software engineering, and safety engineering in the design, development, test, and operation of the system software.

Thus, the purpose of this document (hereafter referred to as the Handbook) is as follows:

Provide management and engineering guidelines to achieve a reasonable level of assurance that software will execute within the system context with an acceptable level of safety risk.

2. Introduction to the Handbook

2.1 Introduction

All members of the system development team should read section 2 of the Software System Safety Handbook (SSSH). This section discusses the following major subjects:

- The major purpose for writing this Handbook
- The scope of the subject matter that this Handbook will present
- The authority by which a SSS program is conducted
- How this Handbook is organized and the best procedure for you to use, to gain its full benefit.

As a member of the software development team, the safety engineer is critical in the design, and redesign, of modern systems. Whether a hardware engineer, software engineer, "safety specialist," or safety manager, it is his/her responsibility to ensure that an acceptable level of safety is achieved and maintained throughout the life cycle of the system(s) being developed. This Handbook provides a rigorous and pragmatic application of SSS planning and analysis to be used by the safety engineer.

SSS, an element of the total system safety and software development program, cannot function independently of the total effort. Nor can it be ignored. Systems, both "simple" and highly integrated multiple subsystems, are experiencing an extraordinary growth in the use of computers and software to monitor and/or control safety-critical subsystems and functions. A software specification error, design flaw, or the lack of initial safety requirements can contribute to or cause a system failure or erroneous human decision. Preventable death, injury, loss of the system, or environmental damage can result. To achieve an acceptable level of safety for software used in critical applications, software safety engineering must be given primary emphasis early in the requirements definition and system conceptual design process. Safety-critical software must then receive a continuous emphasis from management as well as a continuing engineering analysis throughout the development and operational life cycles of the system.

This SSSH is a joint effort. The U.S. Army, Navy, Air Force, and Coast Guard Safety Centers, with cooperation from the Federal Aviation Administration (FAA), National Aeronautics and Space Administration (NASA), defense industry contractors, and academia are the primary contributors. This extensive research captures the "best practices" pertaining to SSS program management and safety-critical software design. The Handbook consolidates these contributions into a single, user-friendly resource. It aids the system development team in understanding their SSS responsibilities. By using this Handbook, the user will appreciate the need for all disciplines to work together in identifying, controlling, and managing software-related hazards within the safety-critical components of hardware systems.

To summarize, this Handbook is a "how-to" guide for use in the understanding of SSS and the contribution of each functional discipline to the overall goal. It is applicable to all types of systems (military and commercial), in all types of operational uses.

2.2 Purpose

The purpose of the SSSH is to provide management and engineering guidelines to achieve a reasonable level of assurance that the software will execute within the system context with an acceptable level of safety risk[1].

2.3 Scope

This Handbook is both a reference document and management tool for aiding managers and engineers at all levels, in any government or industrial organization. It demonstrates "how to" in the development and implementation of an effective SSS process. This process minimizes the likelihood or severity of system hazards caused by poorly specified, designed, developed, or operation of software in safety-critical applications.

The primary responsibility for management of the SSS process lies with the system safety manager/engineer in both the developer's (supplier) and acquirer's (customer) organization. However, nearly every functional discipline has a vital role and must be intimately involved in the SSS process. The SSS tasks, techniques, and processes outlined in this Handbook are basic enough to be applied to any system that uses software in critical areas. It serves the need for all contributing disciplines to understand and apply qualitative and quantitative analysis techniques to ensure the safety of hardware systems controlled by software.

This Handbook is a guide and is not intended to supersede any Agency policy, standard, or guidance pertaining to system safety (MIL-STD-882C) or software engineering and development (MIL-STD-498). It is written to clarify the SSS requirements and tasks specified in governmental and commercial standards and guideline documents. The Handbook is not a compliance document but a reference document. It provides the system safety manager and the software development manager with sufficient information to perform the following:

- Properly scope the SSS effort in the Statement of Work (SOW),
- Identify the data items needed to effectively monitor the contractor's compliance with the contract system safety requirements, and
- Evaluate contractor performance throughout the development life cycle.

The Handbook is not a tutorial on software engineering. However, it does address some technical aspects of software function and design to assist with understanding software safety. It is an objective of this Handbook to provide each member of the SSS Team with a basic understanding of sound systems and software safety practices, processes, and techniques.

[1] The stated purpose of this Handbook closely resembles Nancy Leveson's definition of Software System Safety. The authors would like to provide the appropriate credit for her implicit contribution.

Another objective is to demonstrate the importance of each technical and managerial discipline to work hand-in-hand in defining software safety requirements (SSR) for the safety-critical software components of the system. A final objective is to show where safety features can be designed into the software to eliminate or control identified hazards.

2.4 Authority/Standards

Numerous directives, standards, regulations, and regulatory guides establish the authority for system safety engineering requirements in the acquisition, development, and maintenance of software-based systems. Although the primary focus of this Handbook is targeted toward military systems, much of the authority for the establishment of Department of Defense (DOD) system safety, and software safety programs, is derived from other governmental and commercial standards and guidance. We have documented many of these authoritative standards and guidelines within this Handbook. First, to establish their existence; second, to demonstrate the seriousness that the government places on the reduction of safety risk for software performing safety-critical functions; and finally, to consolidate in one place all authoritative documentation. This allows a PM, safety manager, or safety engineer to clearly demonstrate the mandated requirement and need for a software safety program to their superiors.

2.4.1 Department of Defense

Within the DOD and the acquisition corps of each branch of military service, the primary documents of interest pertaining to system safety and software development include DOD Instruction 5000.1, Defense Acquisition; DOD 5000.2R, Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs; MIL-STD-498, Software Development and Documentation; and MIL-STD-882D, Standard Practice for System Safety. The authority of the acquisition professional to establish a software safety program is provided in the following paragraphs. These paragraphs are quoted or summarized from various DOD directives and military standards. They clearly define the mandated requirement for all DOD systems acquisition and development programs to incorporate safety requirements and analysis into the design, development, testing, and support of software being used to perform or control critical system functions. The DOD documents also levy the authority and responsibility for establishing and managing an effective software safety program to the highest level of program authority.

2.4.1.1 DODD 5000.1

DODD 5000.1, Defense Acquisition, March 15, 1996; Paragraph D.1.d, establishes the requirement and need for an aggressive risk management program for acquiring quality products.

> d. Risk Assessment and Management. PMs and other acquisition managers shall continually assess program risks. Risks must be well understood, and risk management approaches developed, before decision authorities can authorize a program to proceed into the next phase of the acquisition process. To assess and manage risk, PMs and other acquisition managers shall use a variety of techniques, including technology demonstrations, prototyping, and test and evaluation. Risk management encompasses

identification, mitigation, and continuous tracking, and control procedures that feed back through the program assessment process to decision authorities. To ensure an equitable and sensible allocation of risk between government and industry, PMs and other acquisition managers shall develop a contracting approach appropriate to the type of system being acquired.

2.4.1.2 DOD 5000.2R

DOD 5000.2R, Mandatory Procedures for MDAPs and MAIS Acquisition Programs, March 15, 1996, provides the guidance regarding system safety and health.

> **4.3.7.3 System Safety and Health**: The PM shall identify and evaluate system safety and health hazards, define risk levels, and establish a program that manages the probability and severity of all hazards associated with development, use, and disposal of the system. All safety and health hazards shall be managed consistent with mission requirements and shall be cost-effective. Health hazards include conditions that create significant risks of death, injury, or acute chronic illness, disability, and/or reduced job performance of personnel who produce, test, operate, maintain, or support the system.
>
> Each management decision to accept the risks associated with an identified hazard shall be formally documented. The Component Acquisition Executive (CAE) shall be the final approval authority for acceptance of high-risk hazards. All participants in joint programs shall approve acceptance of high-risk hazards. Acceptance of serious risk hazards may be approved at the Program Executive Officer (PEO) level.

2.4.1.3 Military Standards

2.4.1.3.1 MIL-STD-882B, Notice 1

MIL-STD-882B, System Safety Program Requirements, March 30, 1984 (Notice 1, July 1, 1987), remains on numerous government programs which were contracted during the 1980s prior to the issuance of MIL-STD-882C. The objective of this standard is the establishment of a System Safety Program (SSP) to ensure that safety, consistent with mission requirements, is designed into systems, subsystems, equipment, facilities, and their interfaces. The authors of this standard recognized the safety risk that influenced software presented in safety-critical systems. The standard provides guidance and specific tasks for the development team to address the software, hardware, system, and human interfaces. These include the 300-series tasks. The purpose of each task is as follows:

> **Task 301, Software Requirements Hazard Analysis**: The purpose of Task 301 is to require the contractor to perform and document a Software Requirements Hazard Analysis. The contractor shall examine both system and software requirements as well as design in order to identify unsafe modes for resolution, such as out-of-sequence, wrong event, inappropriate magnitude, inadvertent command, adverse environment, deadlocking, failure-to-command, etc. The analysis shall examine safety-critical computer software components at a gross level to obtain an initial safety evaluation of the software system.

Task 302, Top-level Design Hazard Analysis: The purpose of Task 302 is to require the contractor to perform and document a Top-level Design Hazard Analysis. The contractor shall analyze the top-level design, using the results of the Safety Requirements Hazard Analysis if previously accomplished. This analysis shall include the definition and subsequent analysis of safety-critical computer software components, identifying the degree of risk involved, as well as the design and test plan to be implemented. The analysis shall be substantially complete before the software-detailed design is started. The results of the analysis shall be present at the Preliminary Design Review (PDR).

Task 303, Detailed Design Hazard Analysis: The purpose of Task 303 is to require the contractor to perform and document a Detailed Design Hazard Analysis. The contractor shall analyze the software detailed design using the results of the Software Requirements Hazard Analysis and the Top-level Design Hazard Analysis to verify the correct incorporation of safety requirements and to analyze the safety-critical computer software components. This analysis shall be substantially complete before coding of the software is started. The results of the analysis shall be presented at the Critical Design Review (CDR).

Task 304, Code-level Software Hazard Analysis: The purpose of Task 304 is to require the contractor to perform and document a Code-level Software Hazard Analysis. Using the results of the Detailed Design Hazard Analysis, the contractor shall analyze program code and system interfaces for events, faults, and conditions that could cause or contribute to undesired events affecting safety. This analysis shall start when coding begins, and shall be continued throughout the system life cycle.

Task 305, Software Safety Testing: The purpose of Task 305 is to require the contractor to perform and document Software Safety Testing to ensure that all hazards have been eliminated or controlled to an acceptable level of risk.

Task 306, Software/User Interface Analysis: The purpose of Task 306 is to require the contractor to perform and document a Software/User Interface Analysis and the development of software user procedures.

Task 307, Software Change Hazard Analysis: The purpose of Task 307 is to require the contractor to perform and document a Software Change Hazard Analysis. The contractor shall analyze all changes, modifications, and patches made to the software for safety hazards.

2.4.1.3.2 MIL-STD-882C

MIL-STD-882C, System Safety Program Requirements, January 19, 1993, establishes the requirement for detailed system safety engineering and management activities on all system procurements within the DOD. This includes the integration of software safety within the context of the SSP. Although MIL-STD-882B and MIL-STD-882C remain on older contracts within the DOD, MIL-STD-882D is the current system safety standard as of the date of this handbook.

Paragraph 4, General Requirements, 4.1, System Safety Program: The contractor shall establish and maintain a SSP to support efficient and effective achievement of overall system safety objectives.

Paragraph 4.2, System Safety Objectives: The SSP shall define a systematic approach to make sure that:..(b.) Hazards associated with each system are identified, tracked, evaluated, and eliminated, or the associated risk reduced to a level acceptable to the Procuring Authority (PA) throughout entire life cycle of a system.

Paragraph 4.3, System Safety Design Requirements: "...Some general system safety design requirements are:..(j.) Design software controlled or monitored functions to minimize initiation of hazardous events or mishaps."

Task 202, Preliminary Hazard Analysis (PHA), Section 202.2, Task Description: "...The PHA shall consider the following for identification and evaluation of hazards as a minimum: (b.) Safety related interface considerations among various elements of the system (e.g., material compatibilities, electromagnetic interference, inadvertent activation, fire/explosive initiation and propagation, and hardware and software controls.) This shall include consideration of the potential contribution by software (including software developed by other contractors/sources) to subsystem/system mishaps. Safety design criteria to control safety-critical software commands and responses (e.g., inadvertent command, failure to command, untimely command or responses, inappropriate magnitude, or PA-designated undesired events) shall be identified and appropriate actions taken to incorporate them in the software (and related hardware) specifications."

Task 202 is included as a representative description of tasks integrating software safety. The general description is also applicable to all the other tasks specified in MIL-STD-882C. The point is that software safety must be an integral part of system safety and software development.

2.4.1.3.3 MIL-STD-882D

MIL-STD 882D, Standard Practice of System Safety, replaced MIL-STD-882C in September 1999. Although the new standard is radically different than its predecessors, it still captures their basic tenets. It requires that the system developers document the approach to produce the following:

- Satisfy the requirements of the standard,
- Identify hazards in the system through a systematic analysis approach,
- Assess the severity of the hazards,
- Identify mitigation techniques,
- Reduce the mishap risk to an acceptable level,
- Verify and validate the mishap risk reduction, and

- Report the residual risk to the PM.

This process is identical to the process described in the preceding versions of the standard without specifying programmatic particulars. The process described in this handbook meets the requirements and intent of MIL-STD-882D.

Succeeding paragraphs in this Handbook describe its relationship to MIL-STDs-882B and 882C since these invoke specific tasks as part of the system safety analysis process. The tasks, while no longer part of MIL-STD-882D, still reside in the Defense Acquisition Deskbook (DAD). The integration of this Handbook into DAD will include links to the appropriate tasks.

A caveat for those managing contracts: A PM should not blindly accept a developer's proposal to make a "no-cost" change to replace earlier versions of the 882 series standard with MIL-STD 882D. This could have significant implications in the conduct of the safety program preventing the PM and his/her safety team from obtaining the specific data required to evaluate the system and its software.

2.4.1.3.4 DOD-STD-2167A

Although MIL-STD-498 replaced DOD-STD-2167A, Military Standard Defense System Software Development, February 29, 1988, it remains on numerous older contracts within the DOD. This standard establishes the uniform requirements for software development that are applicable throughout the system life cycle. The requirements of this standard provide the basis for government insight into a contractor's software development, testing, and evaluation efforts. The specific requirement of the standard, which establishes a system safety interface with the software development process, is as follows:

> **Paragraph 4.2.3, Safety Analysis**: The contractor shall perform the analysis necessary to ensure that the software requirements, design, and operating procedures minimize the potential for hazardous conditions during the operational mission. Any potentially hazardous conditions or operating procedures shall be clearly defined and documented.

2.4.1.3.5 MIL-STD-498

MIL-STD-498[2], Software Development and Documentation, December 5, 1994, Paragraph 4.2.4.1, establishes an interface with system safety engineering and defines the safety activities which are required for incorporation into the software development throughout the acquisition life cycle. This standard merges DOD-STD-2176A and DOD-STD-7935A to define a set of activities and documentation suitable for the development of both weapon systems and automated information systems. Other changes include improved compatibility with incremental and evolutionary development models; improved compatibility with non-hierarchical design methods; improved compatibility with Computer-Aided Software Engineering (CASE) tools; alternatives to, and more flexibility in, preparing documents; clearer requirements for incorporating reusable software; introduction of software management indicators; added

[2] IEEE 1498, Information Technology - Software Development and Documentation is the demilitarized version of MIL-STD-498 for use in commercial applications

emphasis on software support; and improved links to systems engineering. This standard can be applied in any phase of the system life cycle.

> **Paragraph 4.2.4.1, Safety Assurance:** The developer shall identify as safety-critical those Computer Software Configuration Items (CSCI) or portions thereof whose failure could lead to a hazardous system state (one that could result in unintended death, injury, loss of property, or environmental harm). If there is such software, the developer shall develop a safety assurance strategy, including both tests and analyses, to assure that the requirements, design, implementation, and operating procedures for the identified software minimize or eliminate the potential for hazardous conditions. The strategy shall include a software safety program that shall be integrated with the SSP if one exists. The developer shall record the strategy in the software development plan (SDP), implement the strategy, and produce evidence, as part of required software products, that the safety assurance strategy has been carried out.

In the case of reusable software products [this includes Commercial Off-The-Shelf (COTS)], MIL-STD-498 states that:

> **Appendix B, B.3, Evaluating Reusable Software Products, (b.):** General criteria shall be the software product's ability to meet specified requirements and to be cost effective over the life of the system. Non-mandatory examples of specific criteria include, but are not limited to:..b. Ability to provide required safety, security, and privacy.

2.4.2 Other Government Agencies

Outside the DOD, other governmental agencies are not only interested in the development of safe software, but are aggressively pursuing the development or adoption of new regulations, standards, and guidance for establishing and implementing software SSPs for their developing systems. Those governmental agencies expressing an interest and actively participating in the development of this Handbook are identified below. Also included is the authoritative documentation used by these agencies which establish the requirement for a SwSSP.

2.4.2.1 Department of Transportation

2.4.2.1.1 Federal Aviation Administration

FAA Order 1810 "ACQUISITION POLICY" establishes general policies and the framework for acquisition for all programs that require operational or support needs for the FAA. It implements the Department of Transportation (DOT) Major Acquisition Policy and Procedures (MAPP) in its entirety and consolidates the contents of more than 140 FAA Orders, standards, and other references. FAA Order 8000.70 "FAA SYSTEM SAFETY PROGRAM" requires that the FAA SSP be used, where applicable, to enhance the effectiveness of FAA safety efforts through the uniform approach of system safety management and engineering principles and practices.[3]

[3] FAA System Safety Handbook, Draft, December 31, 1993

A significant FAA safety document is (RTCA)/DO-178B, Software Considerations In Airborne Systems and Equipment Certification. Important points from this resource are as follows:

> **Paragraph 1.1, Purpose**: The purpose of this document is to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements.
>
> **Paragraph 2.1.1, Information Flow from System Processes to Software Processes**: The system safety assessment process determines and categorizes the failure conditions of the system. Within the system safety assessment process, an analysis of the system design defines safety-related requirements that specify the desired immunity from, and system responses to, these failure conditions. These requirements are defined for hardware and software to preclude or limit the effects of faults, and may provide fault detection and fault tolerance. As decisions are being made during the hardware design process and software development processes, the system safety assessment process analyzes the resulting system design to verify that it satisfies the safety-related requirements.

The safety-related requirements are inputs to the software life cycle process. To ensure that they are properly implemented, the system requirements typically include or reference:

- The system description and hardware definition;
- Certification requirements, including Federal Aviation Regulation (United States), Joint Aviation Regulations (Europe), Advisory Circulars (United States), etc.;
- System requirements allocated to software, including functional requirements, performance requirements, and safety-related requirements;
- Software level(s) and data substantiating their determination, failure conditions, their Hazard Risk Index (HRI) categories, and related functions allocated to software;
- Software strategies and design constraints, including design methods, such as, partitioning, dissimilarity, redundancy, or safety monitoring; and
- If the system is a component of another system, the safety-related requirements and failure conditions for that system.

System life cycle processes may specify requirements for software life cycle processes to aid system verification activities.

2.4.2.1.2 Coast Guard

COMDTINST M41150.2D, Systems Acquisition Manual, December 27, 1994, or the "SAM" establishes policy, procedures, and guidance for the administration of Coast Guard major acquisition projects. The SAM implements the DOT MAPP in its entirety. The "System Safety Planning" section of the SAM requires the use of MIL-STD-882C in all Level I, IIIA, and IV

acquisitions. The SAM also outlines system hardware and software requirements in the "Integrated Logistics Support Planning" section of the manual.

Using MIL-STD-498 as a foundation, the Coast Guard has developed a "Software Development and Documentation Standards, Draft, May 1995" document for internal Coast Guard use. The important points from this document are as follows:

> **Paragraph 1.1, Purpose**: The purpose of this standard is to establish Coast Guard software development and documentation requirements to be applied during the acquisition, development, or support of the software system.
>
> **Paragraph 1.2, Application**: "This standard is designed to be contract specific applying to both contractors or any other government agency(s) who would develop software for the Coast Guard."
>
> **Paragraph 1.2.3, Safety Analysis**: "Safety shall be a principle concern in the design and development of the system and it's associated software development products." This standard will require contractors to develop a software safety program, integrating it with the SSP. This standard also requires the contractor to perform safety analysis on software to identify, minimize, or eliminate hazardous conditions that could potentially affect operational mission readiness.

2.4.2.1.3 Aerospace Recommended Practice

"The Society of Automotive Engineers provides two standards representing Aerospace Recommended Practice (ARP) to guide the development of complex aircraft systems. ARP4754 presents guidelines for the development of highly integrated or complex aircraft systems, with particular emphasis on electronic systems. While safety is a key concern, the advice covers the complete development process. The standard is designed for use with ARP4761, which contains detailed guidance and examples of safety assessment procedures. These standards could be applied across application domains but some aspects are avionics specific."[4]

The avionics risk assessment framework is based on Development Assurance Levels (DAL), which are similar to the Australian Defense Standard Def(Aust) 5679 Safety Integrity Levels (SIL). Each functional failure condition identified under ARP4754 and ARP4761 is assigned a DAL based on the severity of the effects of the failure condition identified in the Functional Hazard Assessment. However, the severity corresponds to levels of aircraft controllability rather than direct levels of harm. As a result, the likelihood of accident sequences is not considered in the initial risk assessment.

The DAL of an item in the design may be reduced if the system architecture:

- Provides multiple implementations of a function (redundancy),
- Isolates potential faults in part of the system (partitioning),

[4] International Standards Survey and Comparison to Def(Aust) 5679 Document ID: CA38809-101 Issue: 1.1, Dated 12 May 1999, pg 3.

- Provides for active (automated) monitoring of the item, or
- Provides for human recognition or mitigation of failure conditions.

Detailed guidance is given on these issues. Justification of the reduction is provided by the preliminary system safety assessment.

DALs are provided with equivalent numerical failure rates so that quantitative assessments of risk can be made. However, it is acknowledged that the effectiveness of particular design strategies cannot always be quantified and that qualitative judgments are often required. In particular, no attempt is made to interpret the assurance levels of software in probabilistic terms. Like Def(Aust) 5679, the software assurance levels are used to determine the techniques and measures to be applied in the development processes.

When the development is sufficiently mature, actual failure rates of hardware components are estimated and combined by the System Safety Assessment (SSA) to provide an estimate of the functional failure rates. The assessment should determine if the corresponding DAL has been met. To achieve its objectives, the SSA suggests Failure Modes and Effects Analysis and Fault Tree Analysis (FTA), which are described in the appendices of ARP4761.[5]

2.4.2.2 National Aeronautics and Space Administration

NASA has been developing safety-critical, software-intensive aeronautical and space systems for many years. To support the required planning of software safety activities on these research and operational procurements, NASA published NASA Safety Standard (NSS) 1740.13, Interim, Software Safety Standard, in June 1994. "The purpose of this standard is to provide requirements to implement a systematic approach to software safety as an integral part of the overall SSPs. It describes the activities necessary to ensure that safety is designed into software that is acquired or developed by NASA and that safety is maintained throughout the software life cycle." Several DOD and Military Standards including DOD-STD-2167A, Defense System Software Development, and MIL-STD-882C, System Safety Program Requirements influenced the development of this NASA standard.

The defined purpose of NSS 1740.13 is as follows:

- To ensure that software does not cause or contribute to a system reaching a hazardous state,
- That it does not fail to detect or take corrective action if the system reaches a hazardous state, and
- That it does not fail to mitigate damage if an accident occurs.

2.4.3 Commercial

Unlike the historical relationship established between DOD agencies and their contractors, commercial companies are not obligated to a specified, quantifiable level of safety risk

[5] Ibid. page 27-28.

management on the products they produce (unless contractually obligated through a subcontract arrangement with another company or agency). Instead, they are primarily motivated by economical, ethical, and legal liability factors. For those commercial companies that are motivated or compelled to pursue the elimination or control of safety risk in software, several commercial standards are available to provide them guidance. This Handbook will only reference a few of the most popular. While these commercial standards are readily accessible, few provide the practitioner with a defined software safety process or the "how-to" guidance required to implement the process.

2.4.3.1 Institute of Electrical and Electronic Engineering

The Institute of Electrical and Electronic Engineers (IEEE) published IEEE STD 1228-1994, IEEE Standard for Software Safety Plans, for the purpose of describing the minimum acceptable requirements for the content of a software safety plan. This standard contains four clauses. Clause 1 discusses the application of the standard. Clause 2 lists references to other standards. Clause 3 provides a set of definitions and acronyms used in the standard. Clause 4 contains the required content of a software safety plan. An informative annex is included and discusses software safety analyses. IEEE STD 1228-1994 is intended to be "wholly voluntary" and was written for those who are responsible for defining, planning, implementing, or supporting software safety plans. This standard closely follows the methodology of MIL-STD-882B, Change Notice 1.

2.4.3.2 Electronic Industries Association

The Electronic Industries Association (EIA), G-48 System Safety Committee published the Safety Engineering Bulletin No. 6B, System Safety Engineering In Software Development, in 1990. The G-48 System Safety Committee has as its interest, the procedures, methodology, and development of criteria for the application of system safety engineering to systems, subsystems, and equipment. The purpose of the document is "...to provide guidelines on how a system safety analysis and evaluation program should be conducted for systems which include computer-controlled or -monitored functions. It addresses the problems and concerns associated with such a program, the processes to be followed, the tasks which must be performed, and some methods which can be used to effectively perform those tasks."

2.4.3.3 International Electrotechnical Commission

The International Electrotechnical Commission (IEC) has submitted a draft International Standard (IEC-61508) December 1997, which is primarily concerned with safety-related control systems incorporating Electrical/Electronic/Programmable Electronic Systems (E/E/PES). It also provides a framework which is applicable to safety-related systems irrespective of the technology on which those systems are based (e.g., mechanical, hydraulic, or pneumatic). Although some parts of the standard are in draft form, it is expected to be approved for use in 1999. "The draft International Standard has two concepts which are fundamental to its application - namely, a Safety Life Cycle and SIL. The Overall Safety Life Cycle is introduced in Part 1 and forms the

central framework which links together most of the concepts in this draft International Standard."[6]

This draft International Standard (IEC-61508) consists of seven parts:

Part 1: General Requirements

Part 2: Requirements for E/E/PES

Part 3: Software Requirements

Part 4: Definitions

Part 5: Guidelines on the Application of Part 1

Part 6: Guidelines on the Application of Part 2 and Part 3

Part 7: Bibliography of Techniques

The draft standard addresses all relevant safety life cycle phases when E/E/PES are used to perform safety functions. It has been developed with a rapidly developing technology in mind. The framework in this standard is considered to be sufficiently robust and comprehensive to cater to future developments.

2.5 International Standards

2.5.1 Australian Defense Standard DEF(AUST) 5679

DEF AUST 5679, published by the Australian Department of Defense in March 1999, is a standard for the procurement of safety-critical systems with an emphasis on computer based systems. It focuses on safety management and the phased production of safety assurance throughout the system development lifecycle, with emphasis on software and software-like processes. A safety case provides auditable evidence of the safety assurance argument.[7]

"Software risk and integrity assessment is based on the concept of development integrity levels. Probabilistic interpretations of risk are explicitly excluded because of the scope for error or corruption in the quantitative analysis process, and because it is currently impossible to interpret or assess low targets of failure rates for software or complex designs.

For each potential accident identified by the PHA, a severity category (catastrophic, fatal, severe, and minor) is allocated, based on the level of injury incurred. Sequences of events that could lead to each accident are identified, and assigned a probability where estimation is possible.

One of seven Levels of Trust (LOT) is allocated to each system safety requirement, depending on the severity category of the accidents that may result from the corresponding system hazard. The LOT may be reduced if each accident sequence can be shown to be sufficiently improbable.

[6] IEC 1508-1, Ed. 1, (DRAFT), Functional Safety; Safety Related Systems, June 1995
[7] International Standards Survey and Comparison to DEF(AUST) 5679 Document ID: CA38809-101 Issue: 1.1, Dated 12 May 1999, pg 3

Each LOT defines the desired level of confidence that the corresponding system safety requirement will be met.

Next, one of seven SILs is assigned to each Component Safety Requirement (CSR), indicating the level of rigor required meeting the CSR. By default, the SIL level of the CSR is the same as the LOT of the system safety requirement corresponding to the CSR. However, the default SIL may be reduced by up to two levels by implementing fault-tolerant measures in the design to reduce the likelihood of the corresponding hazard. As the standard prohibits allocation of probabilities to hazards, this is based on a qualitative argument."[8]

2.5.2 United Kingdom Defense Standard 00-55 & 00-54

"United Kingdom (UK) DEF STAN 00-55 describes requirements and guidelines for procedures and technical practices in the development of safety-related software. The standard applies to all phases of the procurement lifecycle. Interim UK DEF STAN 00-54 describes requirements for the procurement of safety-related electronic hardware, with particular emphasis on the procedures required in various phases of the procurement lifecycle. Both standards are designed to be used in conjunction with DEF STAN 00-56."[9]

"DEF STANs 00-55 and 00-54 require risk assessment to be conducted in accordance with DEF STAN 00-56. DEF STAN 00-55 explicitly mentions that software diversity may, if justified, reduce the required SIL of the application being developed."[10]

2.5.3 United Kingdom Defense Standard 00-56

"UK DEF STAN 00-56 provides requirements and guidelines for the development of all defense systems. The standard applies to all systems engineering phases of the project lifecycle and all systems, not just computer-based ones."[11]

"In DEF STAN 00-56, accidents are classified as belonging to one of four severity categories and one of six probability categories. The correspondence between probability categories and actual probabilities must be stated and approved by the Independent Safety Auditor. Using these classifications, a risk class is assigned to each accident using a matrix approved by the Independent Safety Auditor before hazard analysis activities begin.

For systematic (as opposed to random) failures, the SIL (or actual data if available) determines the minimum failure rate that may be claimed of the function developed according to the SIL; such failure rates must be approved by the Independent Safety Auditor (ISA). Accidents in the highest risk class (A) are regarded as unacceptable, while probability targets are set for accidents in the next two risk classes (B and C). Accidents in the lowest risk class are regarded as tolerable. Accident probability targets are regarded as having a systematic and a random component. The consideration of accident

[8] International Standards Survey and Comparison to Def(Aust) 5679 Document ID: CA38809-101 Issue: 1.1, Dated 12 May 1999, pg 26-27.
[9] Ibid., pg 3.
[10] Ibid., Page 27
[11] Ibid., Page 3.

probability targets and accident sequences determines the hazard probability targets with systematic and random components. These hazard probability targets must be approved by the Independent Safety Auditor."

DEF STAN 00-56 recommends conducting a "Safety Compliance Assessment using techniques such as FTA. If the hazard probability target cannot be met for risk class C, then risk reduction techniques such as redesign, safety or warning features, or special operator procedures must be introduced. If risk reduction is impracticable, then risk class B may be used with the approval of the Project Safety Committee."[12]

2.6 Handbook Overview

2.6.1 Historical Background

The introduction of software-controlled, safety-critical systems has caused considerable ramifications in the managerial, technical, safety, economic, and scheduling risks of both hardware and software system developments. Although this risk is discussed extensively in Section 3, the primary focus of this Handbook is documented in Section 4. It includes the identification; documentation (to include evidence through analyses); and elimination, or control, of the safety risk associated with software in the design, requirements, development, test, operation, and support of the "system."

A software design flaw or run-time error within safety-critical functions of a system introduces the potential of a hazardous condition that could result in death, personal injury, loss of the system, or environmental damage. Appendix F provides abstracts of numerous examples of software-influenced accidents and failures. The incident examples in Appendix F include the following:

F.1 - Therac Radiation Therapy Machine Fatalities

F.2 - Missile Launch Timing Error Causes Hang-Fire

F.3 - Reused Software Causes Flight Controls Shut Down

F.4 - Flight Controls Fail at Supersonic Transition

F.5 - Incorrect Missile Firing Due to Invalid Setup Sequence

F.6 - Operator Choice of Weapon Release Over-Ridden by Software Control

2.6.2 Problem Identification

Since the introduction of digital controls, the engineering community has wrestled (along with their research brethren) with processes, methods, techniques, and tools for the sole purpose of reducing the safety risk of software-controlled operations. Each engineering discipline viewed

[12] International Standards Survey and Comparison to DEF(AUST) 5679 Document ID: CA38809-101 Issue: 1.1, Dated 12 May 1999, pg 27.

the problem from a vantage point and perspective from within the confines of their respective area of expertise. In many instances, this view was analogous to the view seen when looking down a tunnel. The responsibilities of, and the interfaces with, other management and engineering functions were often distorted due to individual or organizational biases.

Part of the problem is that SSS is still a relatively new discipline with methodologies, techniques, and processes that are still being researched and evaluated in terms of logic and practicality for software development activities. As with any new discipline, the problem must be adequately defined prior to the application of recommended practices.

2.6.2.1 Within System Safety

From the perspective of most of the system safety community, digital control of safety-critical functions introduced a new and unwanted level of uncertainty to a historically sound hazard analysis methodology for hardware. Many within system safety were unsure of how to integrate software into the system safety process, techniques, and methods that were currently being used. System safety managers and engineers, educated in the 1950s, 60s, and 70s, had relatively no computer-, or software-related education or experience. This compounded their reluctance to, or in many cases their desire or ability to, even address the problem.

In the late 1970s and early 1980s, bold individuals within the safety, software, and research (academia) communities took their first steps in identifying and addressing the safety risks associated with software. Although these individuals may not have been in total lock step and agreement, they did, in fact, lay the necessary foundation for where we are today. It was during this period that MIL-STD-882B was developed and published. This was the first military standard to require that the developing contractor perform SSS engineering and management activities and tasks. However, due to the distinct lack of cooperation or communication between the system safety and software engineering disciplines in defining a workable process for identifying and controlling software-related hazards in developing systems, the majority of system safety professionals waited for academia, or the software engineering community to develop a "silver bullet" analysis methodology or tool. It was their hope that such an analytical technique or verification tool could be applied to finished software code to identify any fault paths to hazard conditions which could then be quickly corrected prior to delivery. This concept did not include the identification of system hazard and failure modes caused (or influenced) by software inputs, or the identification of safety-specific requirements to mitigate these hazards and failure modes. Note that there is yet no "silver bullet," and there will probably never be one. Even if a "silver bullet" existed, it would be used too late in the system development life cycle to influence design.

To further obscure the issue, the safety community within DOD finally recognized that contractors developing complex hardware and software systems must perform "software safety tasks." As a result contracts from that point forward included tasks that included software in the system safety process. The contractor was now forced to propose, bid, and perform software safety tasks with relatively little guidance. Those with software safety tasks on contract were in a desperate search for *any* tool, technique, or method that would assist them in meeting their contractual requirements. This was demonstrated by a sample population survey conducted in

1988 involving software and safety engineers and managers[13]. When these professionals were asked to identify the tools and techniques that they used to perform contractual obligations pertaining to software safety, they provided answers that were wide and varied across the analytical spectrum. Of 148 surveyed, 55 provided responses. These answers are provided in Table 2-1. It is interesting to note that of all respondents to the survey, only five percent felt that they had accomplished anything meaningful in terms of reducing the safety risk of the software analyzed.

Table 2-1: Survey Response

Software Hazard Analysis Tools

No.	Tool/Technique	No.	Tool/Technique
8	Fault Tree Analysis	1	Hierarchy Tool
4	Software PrelimHazard Analysis	1	Compare & Certification Tool
3	Traceability Analysis	1	System Cross Check Matrices
3	Failure Modes & Effects Analysis	1	Top-Down Review of Code
2	Requirements Modeling/Analysis	1	Software Matrices
2	Source Code Analysis	1	Thread Analysis
2	Test Coverage Analysis	1	Petri-Net Analysis
2	Cross Reference Tools	1	Sofware Hazard List
2	Code/Module Walkthrough	1	BIT/FIT Plan
2	Sneak Circuit Analysis	1	Nuclear Safety Cross-Check Anal.
2	Emulation	1	Mathematical Proof
2	SubSystem Hazard Analysis	1	Software Fault Hazard Analysis
1	Failure Mode Analysis	1	MIL-STD 882B, Series 300 Tasks
1	Prototyping	1	Topological Network Trees
1	Design and Code Inspections	1	Critical Function Flows
1	Checklist of Common SW Errors	1	Black Magic
1	Data Flow Techniques		

NOTE: No. = Cumulative total from those responding to the 1988 Survey

The information provided in Table 2-1 demonstrated that the lack of any standardized approach for the accomplishment of software safety tasks that were levied contractually. It also appeared as if the safety engineer either tried to accomplish the required tasks using a standard system safety approach, or borrowed the most logical tool available from the software development group. In either case, they remained unconvinced of their efforts' utility in reducing the safety risk of the software performing in their system.

2.6.2.2 Within Software Development

Historically, the software development and engineering community made about as much progress addressing the software safety issue as did system safety. Although most software development managers recognized the safety risk potential that the software posed within their systems, few possessed the credible means or methods for both minimizing the risk potential and verifying that safety specification requirements had been achieved in the design. Most failed to include system safety engineering in software design and development activities, and many did not recognize that this interface was either needed or required.

[13] Mattern, Steven F., Software System Safety, Masters Thesis, Department of Computer Resource Management, Webster University, December 1988

A problem, which still exists today, is that most educational institutions do not teach students in computer science and software engineering that there is a required interface with safety engineering when software is integrated into a potentially hazardous system. Although the software engineer may implement a combination of fault avoidance, fault removal, and/or fault tolerance techniques in the design, code, or test of software, they usually fail to tie the fault or error potential to a specific system hazard or failure mode. While these efforts most likely increase the overall reliability of the software, many fail to verify that the safety requirements of the system have been implemented to an acceptable level.

It is essential that the software development community understand the needed interface with system safety and that system safety understands their essential interface with software development.

2.6.3 Management Responsibilities

The ultimate responsibility for the development of a "safe system" rests with program management. The commitment of qualified people and an adequate budget and schedule for a software development program must begin with the program director or PM. Top management must be a strong voice of safety advocacy and must communicate this personal commitment to each level of program and technical management. The PM must be committed to support the integrated safety process within systems engineering and software engineering in the design, development, test, and operation of the system software. Figure 2-1 graphically portrays the managerial element for the integrated team.

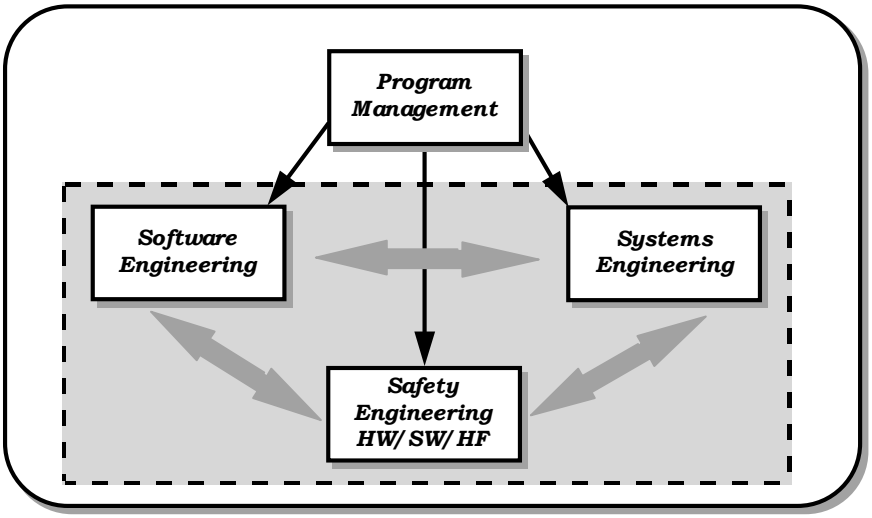


Figure 2-1: Management Commitment to the Integrated Safety Process

2.6.4 Introduction to the "Systems" Approach

System safety engineering has historically demonstrated the benefits of a "systems" approach to safety risk analysis and mitigation. When a hazard analysis is conducted on a hardware subsystem as a separate entity, it will produce a set of unique hazards applicable only to that subsystem. However, when that same subsystem is analyzed in the context of its physical,

functional, and zonal interfaces with the rest of the "system components," the analysis will likely produce numerous other hazards which were not discovered by the original analysis. Conversely, the results of a system analysis may demonstrate that hazards identified in the subsystem analysis were either reduced or eliminated by other components of the system. Regardless, the identification of critical subsystem interfaces (such as software) with their associated hazards is a vital aspect of safety risk minimization for the total system.

When analyzing software that performs, and/or controls, safety-critical functions within a system, a "systems approach" is also required. The success of a software safety program is predicated on it. Today's software is a very critical component of the safety risk potential of systems being developed and fielded. Not only are the internal interfaces of the system important to safety, but so are the external interfaces.

Figure 2-2 depicts specific software internal interfaces within the "system" block (within the ovals) and also external software interfaces to the system. Each identified software interface may possess safety risk potential to the operators, maintainers, environment, or the system itself. The acquisition and development process must consider these interfaces during the design of both the hardware and software systems. To accomplish this, the hardware and software development life cycles must be fully understood and integrated by the design team.

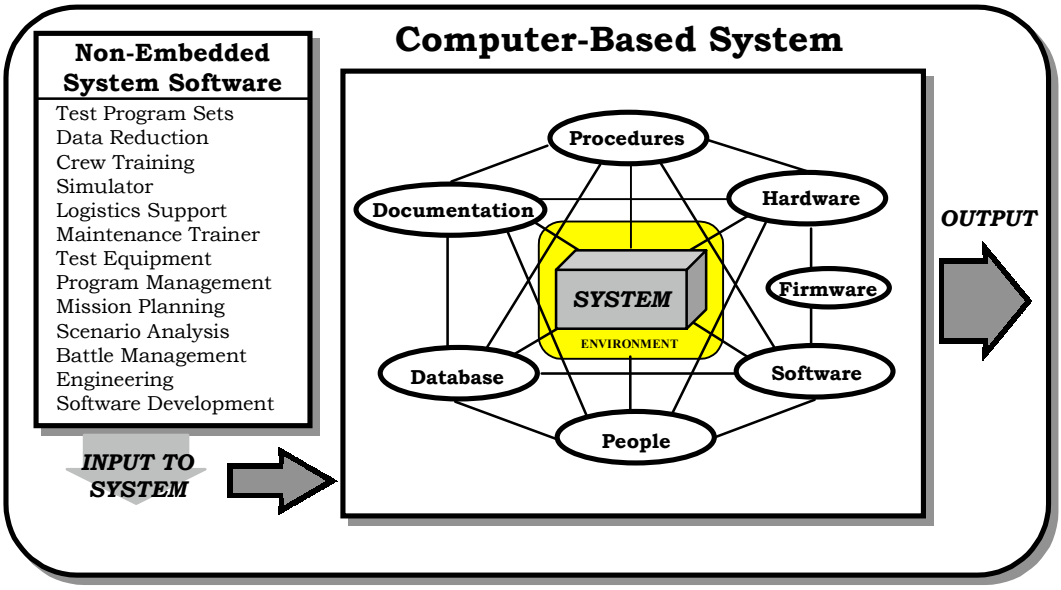


Figure 2-2: Example of Internal System Interfaces

2.6.4.1 The Hardware Development Life Cycle

The typical hardware development life cycle has been in existence for many years. It is a proven acquisition model which has produced, in most instances, the desired engineering results in the design, development, manufacturing, fabrication, and test activities. It consists of five phases. These are identified as the concept exploration and definition, demonstration and validation, engineering and manufacturing development, production and deployment, and operations and support phases. Each phase of the life cycle ends, and the next phase begins, with a milestone

decision point (0, I, II, III, and IV). An assessment of the system design and program status is made at each milestone decision point, and plans are made or reviewed for subsequent phases of the life cycle. Specific activities conducted for each milestone decision are covered in numerous system acquisition management courses and documents. Therefore, they will not be discussed in greater detail in the contents of this Handbook.

The purpose of introducing the system life cycle in this Handbook is to familiarize the reader with a typical life cycle model. The one shown in Figure 2-3 is used in most DOD procurements. It identifies and establishes defined phases for the development life cycle of a system and can be overlaid on a proposed timetable to establish a milestone schedule. Detailed information regarding milestones and phases of a system life cycle can be obtained from Defense Systems Management College (DSMC) documentation, and systems acquisition management course documentation of the individual services.

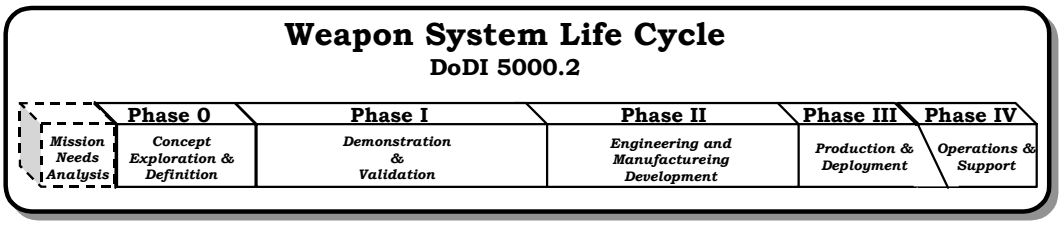


Figure 2-3: Weapon System Life Cycle

2.6.4.2 The Software Development Life Cycle

The system safety team must be fully aware of the software life cycle being used by the development activity. In the past several years, numerous life cycle models have been identified, modified, and used in some capacity on a variety of software development programs. This Handbook will not enter into a discussion as to the merits and limitations of different life cycle process models because the software engineering team must make the decision for or against a model for an individual procurement. The important issue here is for the system safety team to recognize which model is being used, and how they should correlate and integrate safety activities with the chosen software development model to achieve the desired outcomes and safety goals. Several different models will be presented to introduce examples of the various models to the reader.

Figure 2-4 is a graphical representation of the relationship of the software development life cycle to the system/hardware development life cycle. Note that software life cycle graphic shown in Figure 2-4 portrays the DOD-STD-2167A software life cycle, which was replaced with MIL-STD-498, dated December 5, 1994. The minor changes made to the software life cycle by MIL-STD-498 are also shown. Notice also, that the model is representative of the "Waterfall," or "Grand Design" life cycle. While this model is still being used on numerous procurements, other models are more representative of the current software development schemes currently being followed, such as the "Spiral" and "Modified V" software development life cycles.

It is important to recognize that the software development life cycle does not correlate exactly with the hardware (system) development life cycle. It "lags" behind the hardware development

at the beginning but finishes before the hardware development is completed. It is also important to realize that specific design reviews for hardware usually lag behind those required for software. The implications will be discussed in Section 4 of this Handbook.

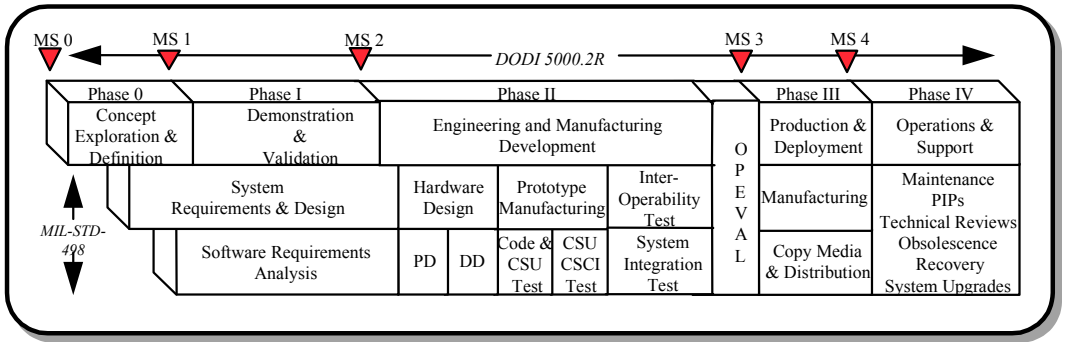


Figure 2-4: Relationship of Software to the Hardware Development Life Cycle

2.6.4.2.1 Grand Design, Waterfall Life Cycle Model[14]

The Waterfall software acquisition and development life cycle model is the oldest in terms of use by software developers. This strategy usually uses DOD-STD-2167A terminology and "...was conceived during the early 1970s as a remedy to the *code-and-fix* method of software development." Grand Design places emphasis on up-front documentation during early development phases, but does not support modern development practices such as prototyping and automatic code generation. "With each activity as a prerequisite for succeeding activities, this strategy is a risky choice for unprecedented systems because it inhibits flexibility." Another limitation to the model is that after a single pass through the model, the system is complete. Therefore, many integration problems are identified much too late in the development process to be corrected without significant cost and schedule impacts. In terms of software safety, interface issues must be identified and rectified as early as possible in the development life cycle to be adequately corrected and verified. Figure 2-5 is a representation of the Grand Design, or Waterfall, life cycle model. The Waterfall model is not recommended for large, software-intensive, systems. This is due to the limitations stated above and the inability to effectively manage program risks, including safety risk during the software development process. The Grand Design does, however, provide a structured and well-disciplined method for software development.

[14] The following descriptions of the software acquisition life cycle models are either quoted or paraphrased from the Guidelines for Successful Acquisition and Management of Software Intensive Systems, Software Technology Support Center (STSC), September 1994, unless otherwise noted.

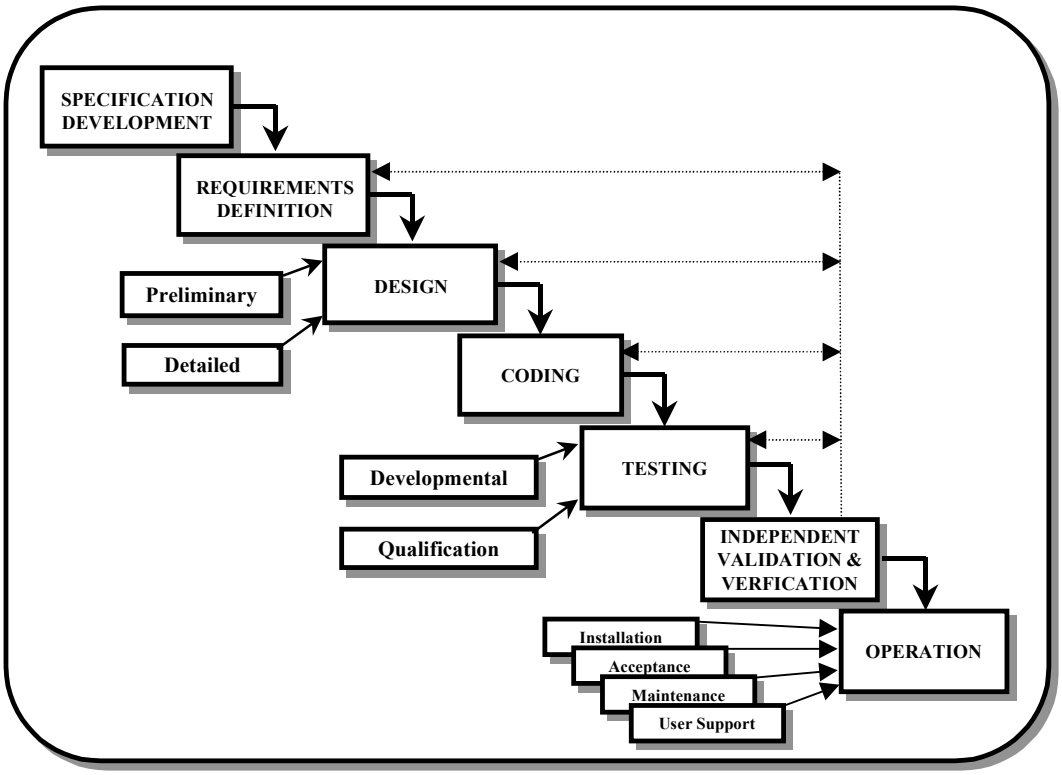


Figure 2-5: Grand Design Waterfall Software Acquisition Life Cycle Model

2.6.4.2.2 Modified V Life Cycle Model

The Modified V software acquisition life cycle model is another example of a defined method for software development. It is depicted in Figure 2-6. This model is heavily weighted in the ability to design, code, prototype, and test in increments of design maturity. "The left side of the figure identifies the specification, design, and coding activities for developing software. It also indicates when the test specification and test design activities can start. For example, the system/acceptance tests can be specified and designed as soon as software requirements are known. The integration tests can be specified and designed as soon as the software design structures are known. And, the unit tests can be specified and designed as soon as the code units are prepared."[15] The right side of the figure identifies when the evaluation activities occur that are involved with the execution and testing of the code at its various stages of evolution.

[15] Software Test Technologies Report, August 1994, STSC, Hill Air Force Base, UT 84056

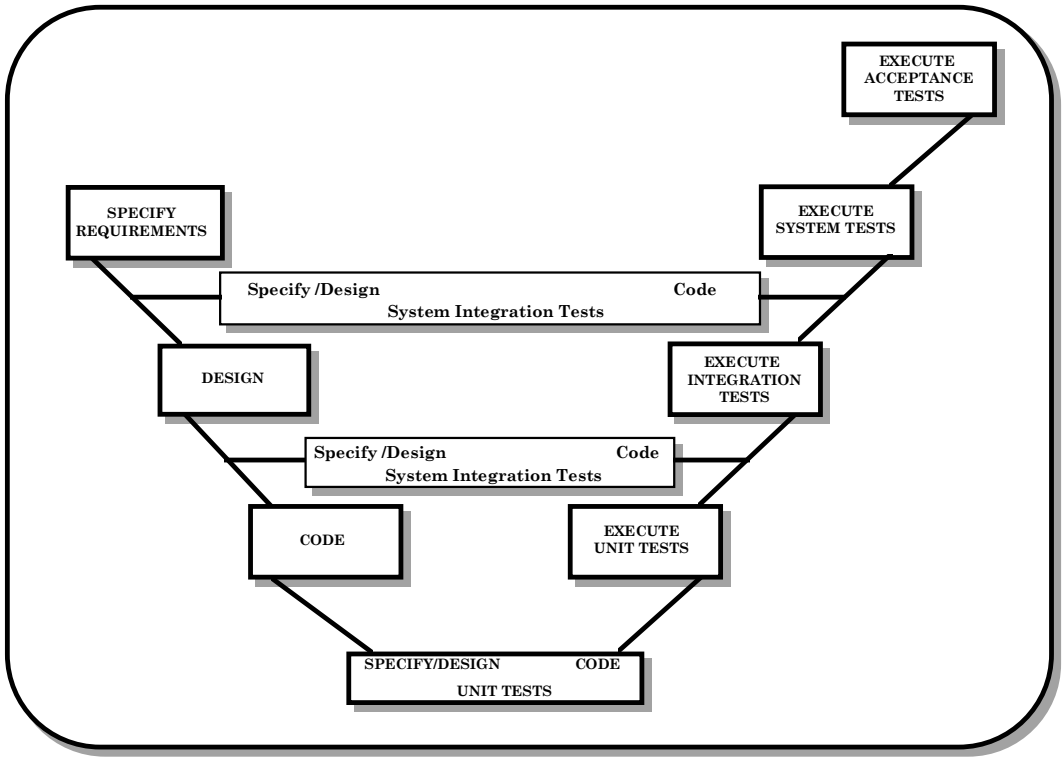


Figure 2-6: Modified V Software Acquisition Life Cycle Model

2.6.4.2.3 Spiral Life cycle Model

The Spiral acquisition life cycle model provides a risk-reduction approach to the software development process. In the Spiral model, Figure 2-7, the radial distance is a measure of effort expended, while the angular distance represents progress made. It combines features of the Waterfall and the incremental prototype approaches to software development. "Spiral development emphasizes evaluation of alternatives and risk assessment. These are addressed more thoroughly than with other strategies. A review at the end of each phase ensures commitment to the next phase or identifies the need to rework a phase if necessary. The advantages of Spiral development are its emphasis on procedures, such as risk analysis, and its adaptability to different development approaches. If Spiral development is employed with demonstrations and Baseline/Configuration Management (CM), you can get continuous user buy-in and a disciplined process."[16]

[16] Guidelines for Successful Acquisition and Management of Software Intensive Systems, STSC, September 1994.

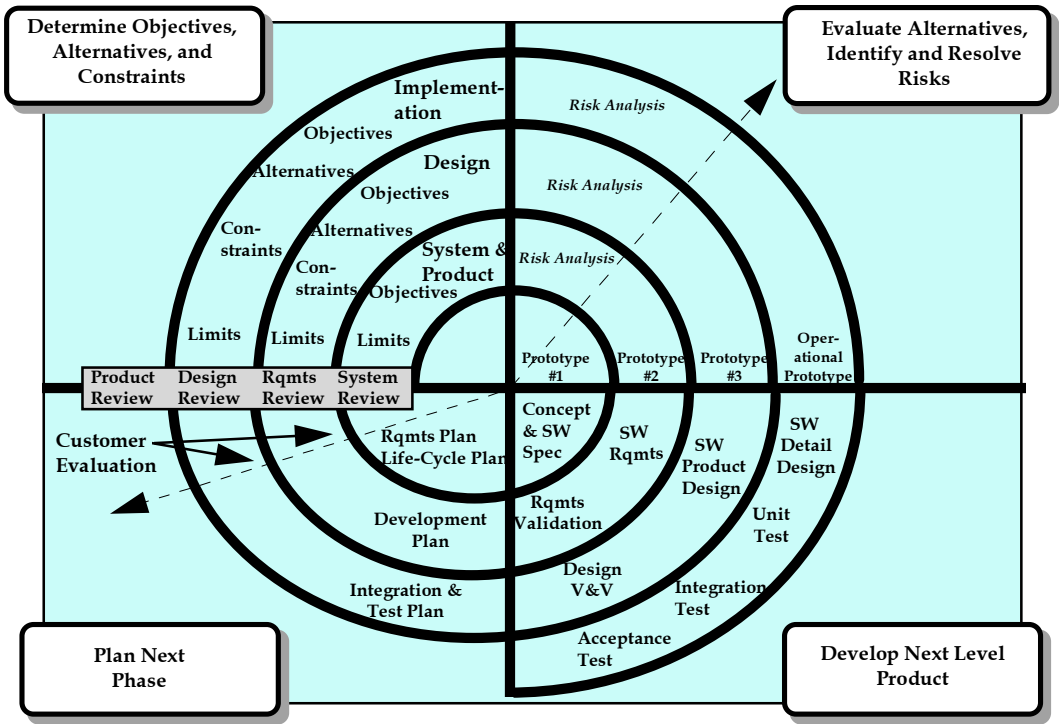


Figure 2-7: Spiral Software Acquisition Life Cycle Model

Within the DOD, an Ada Spiral Model Environment is being considered for some procurements where the Ada language is being used. It provides an environment that combines a model and a tool environment, such that it offers the ability to have continual *touch-and-feel* of the software product (as opposed to paper reports and descriptions). This model represents a "demonstration-based" process that employs a top-down incremental approach that results in an early and continuous design and implementation validation. Advantages of this approach are that it is built from the top down, it supports partial implementation; the structure is automated, real and evolved; and that each level of development can be demonstrated. Each build and subsequent demonstration validates the process and the structure to the previous build.

2.6.4.3 The Integration of Hardware and Software Life Cycles

The life cycle process of system development was instituted so managers would not be forced to make snap decisions. A structured life cycle, complete with controls, audits, reviews, and key decision points, provides a basis for sound decision making based on knowledge, experience, and training. It is a logical flow of events representing an orderly progression from a "user need" to finalize activation, deployment, and support.

The "systems approach" to software safety engineering must support a structured, well-disciplined, and adequately documented system acquisition life cycle model that incorporates both the system development model and the software development model. Program plans (as described in Appendix C, Section C.7) must describe in detail how each engineering discipline will interface and perform within the development life cycle. It is recommended that you refer back to Figure 2-4 and review the integration of the hardware and software development life

cycle models. Graphical representations of the life cycle model of choice for a given development activity must be provided during the planning processes. This activity will aid in the planning and implementation processes of software safety engineering. It will allow for the integration of safety-related requirements and guidelines into the design and code phases of software development. It will also assist in the timely identification of safety-specific test and verification requirements to prove that original design requirements have been implemented as they were intended. It further allows for the incorporation of safety inputs to the prototyping activities in order to demonstrate safety concepts.

2.6.5 A "Team" Solution

The system safety engineer (SSE) cannot reduce the safety risk of systems software by himself. The software safety process must be an integrated team effort between engineering disciplines. Previously depicted in Figure 2-1, software, safety, and systems engineering are the pivotal players of the team. The management block is analogous to a "conductor" that provides the necessary motivation, direction, support, and resources for the team to perform as a well-orchestrated unit.

It is the intent of the authors of this Handbook to demonstrate that neither the software developers, nor safety engineers, can accomplish the necessary tasks to the level of detail required by themselves. This Handbook will focus on the required tasks of the safety engineer, the software engineer, the software safety engineer, the system and design engineers, and the interfaces between each. Regardless of who executes the individual software safety tasks, each engineer must be intimately aware of the duties, responsibilities, and tasks required from each functional discipline. Each must also understand the time (in terms of life cycle schedule), place (in terms of required audits, meetings, reviews, etc.), and functional analysis tasks that must be produced and delivered at any point in the development process. Section 4 will expand on the team approach in detail as the planning, process tasks, products, and risk assessment tasks are presented. Figure 2-8 uses a puzzle analogy to demonstrate that the software safety approach must establish integration between functions and among engineers. Any piece of the puzzle that is missing from the picture will propagate into an unfinished or incomplete software safety work.

The elements contributing to a credible and successful software safety engineering program will include the following:

- A defined and established system safety engineering process,
- A structured and disciplined software development process,
- An established hardware and software systems engineering process,
- An established hardware/software configuration control process, and
- An integrated SSS Team responsible for the identification, implementation, and verification of safety-specific requirements in the design and code of the software.

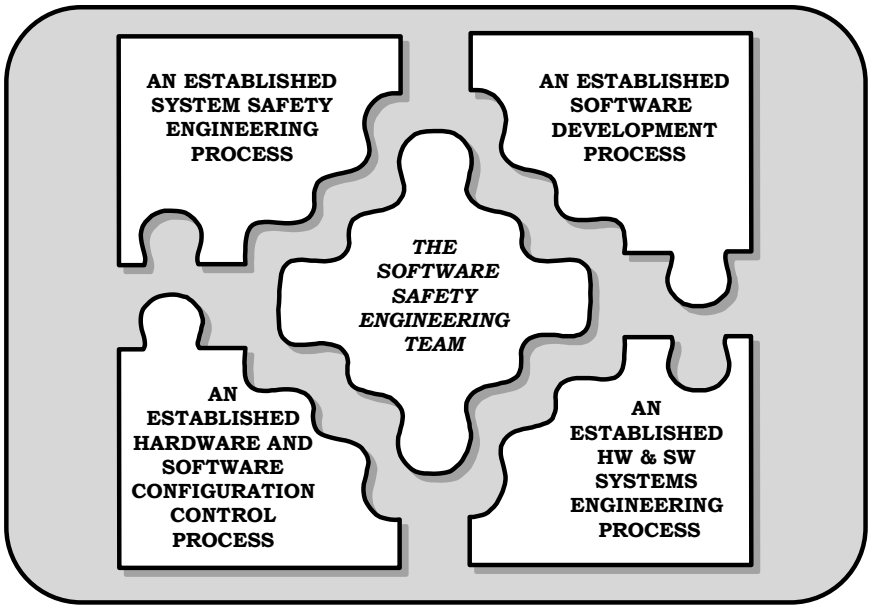


Figure 2-8: Integration of Engineering Personnel and Processes

2.7 Handbook Organization

This Handbook is organized to provide the capability to review or extract subject information important to the reader. For example, the Executive Overview may be the only portion required by the executive officer, program director, or PM to determine whether a software safety program is required for their program. It is to be hoped that the executive section will provide the necessary motivation, authority, and impetus for establishing a software safety program consistent with the nature of their development. Engineering and software managers, on the other hand, will need to read further into the document to obtain the managerial and technical process steps required for a software-intensive, safety-critical system development program. Safety program managers, safety engineers, software engineers, systems engineers, and software safety engineers will need to read even further into the document to gather the information necessary to develop, establish, implement, and manage an effective SwSSP. This includes the "how-to" details for conducting various analyses required to ensure that the system software will function within the system context to an acceptable level of safety risk. Figure 2-9 graphically depicts the layout of the four sections of the Handbook, the appendices, and a brief description of the contents of each.

As shown in Figure 2-9, Section 1 provides an executive overview of the handbook for the purpose of providing executive management a simplified overview of the subject of software safety. It also communicates the requirement and authority for a SSS program; motivation and authority for the requirement; and their roles and responsibilities to the customer, the program, and to the design and development engineering disciplines. Section 2 provides an in-depth description of the purpose and scope of the Handbook, and the authority for the establishment of a SwSSP on DOD procurements and acquisition research and development activities. It also provides a description of the layout of the Handbook as it applies to the acquisition life cycle of a system development. Section 3 provides an introduction to system safety engineering and

management for those readers not familiar with the MIL-STD-882 methods and the approach for the establishment and implementation of a SSP. It also provides an introduction to risk management and how safety risk is an integral part of the risk management function. Section 3 also provides an introduction to, and an overview of, the system acquisition, systems engineering, and software development process and guidance for the effective integration of these efforts in comprehensive systems safety process. Section 4 provides the "how-to" of a baseline software safety program. The authors recognize that not all acquisitions and procurements are similar, nor do they possess the same problems, assumptions, and limitations in terms of technology, resources, development life cycles, and personalities. This section provides the basis for careful planning and forethought required in establishing, tailoring, and implementing a SwSSP guidance for the practitioner, and not as a mindless "checklist" process.

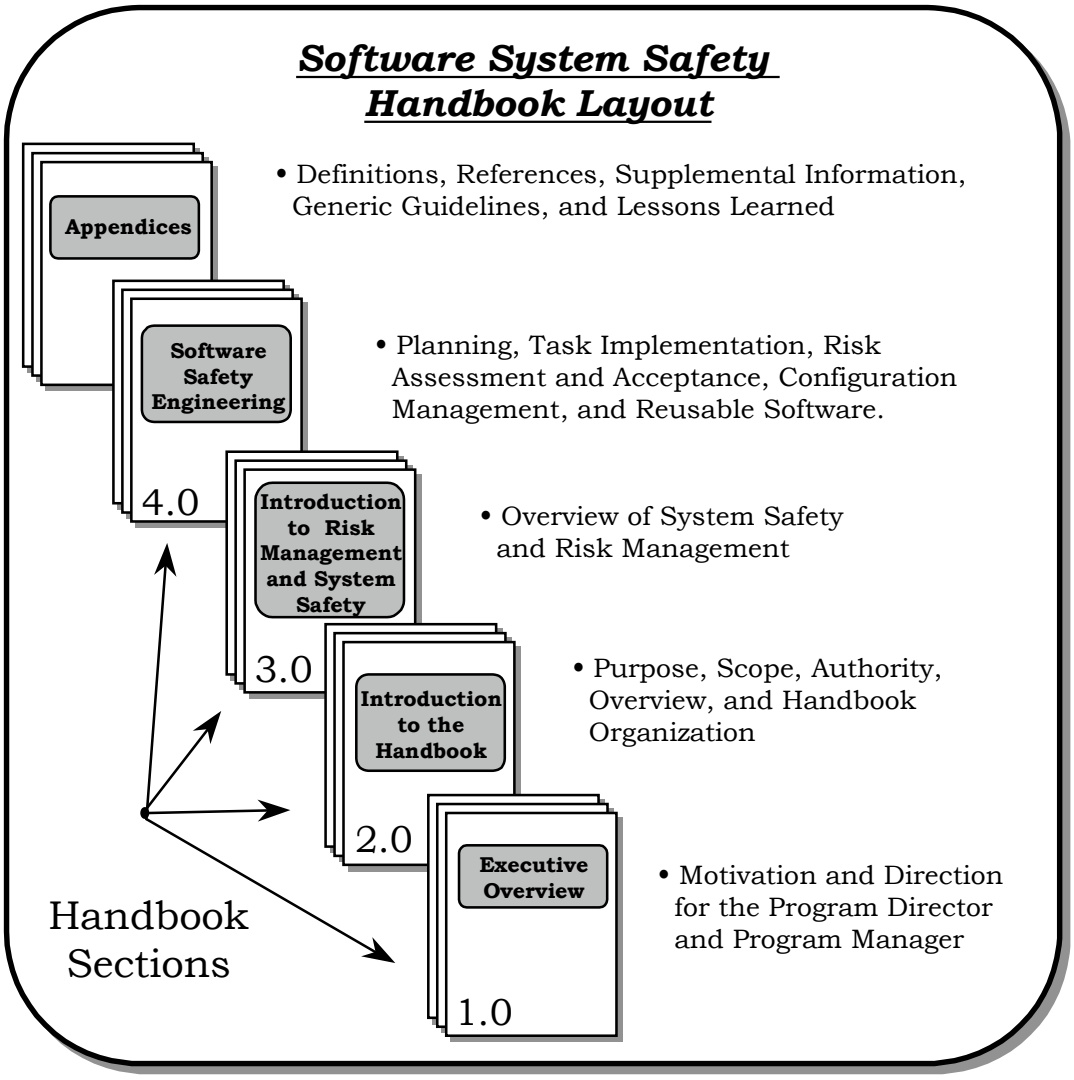


Figure 2-9: Handbook Layout

Section 4, Software Safety Engineering, is formatted logically (see Figure 2-10) to provide the reader with the steps required for planning, task implementation, and risk assessment and

acceptance for a SSS program. Appendix C.9 through C-11 provides information regarding the management of configuration changes and issues pertaining to software reuse and COTS software packages.

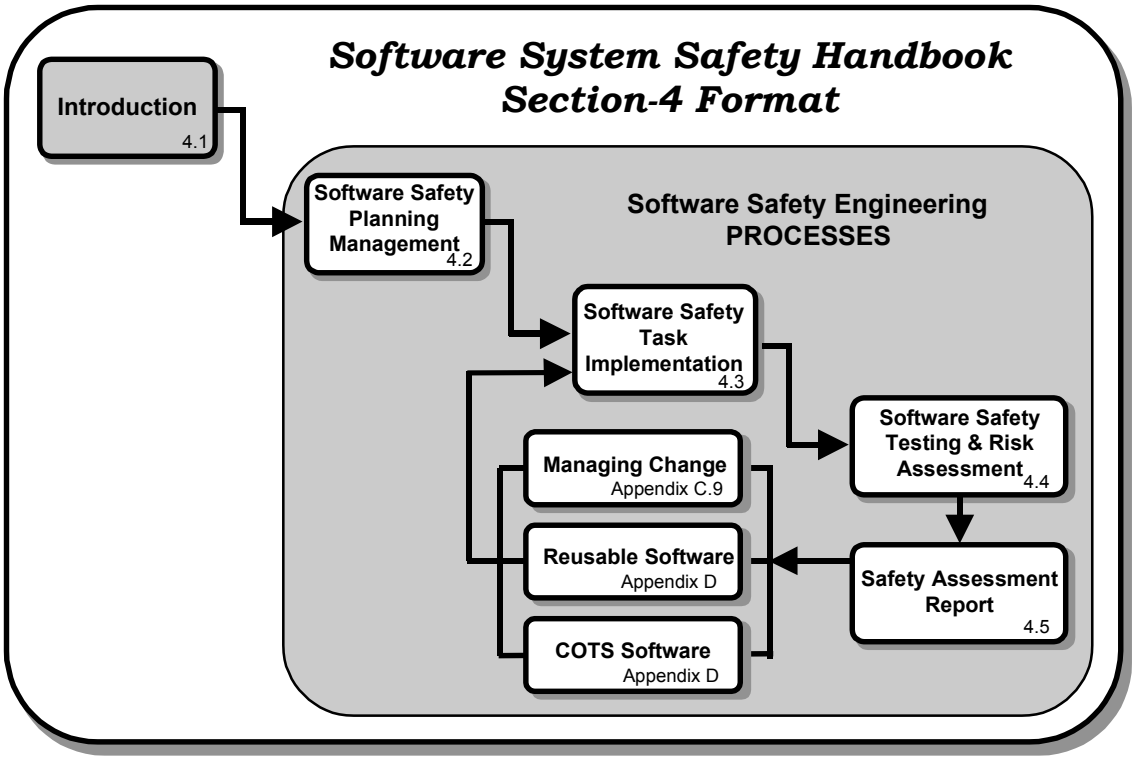


Figure 2-10: Section 4 Format

2.7.1 Planning and Management

Section 4 begins with the planning required to establish a SwSSP. It discusses the program interfaces, contractual interfaces and obligations, safety resources, and program planning and plans. This particular section assists and guides the safety manager and engineer in the required steps of software safety program planning. Although there may be subject areas that are not required for individual product procurements, each area should be addressed and considered in the planning process. It is acceptable to determine that a specific activity or deliverable is not appropriate or necessary for *your* individual program.

2.7.2 Task Implementation

This is the very heart of the handbook as applied to implementing a credible software safety program. It establishes a step-by-step baseline of "best practices" for today's approach in reducing the safety risk of software performing safety-critical and safety-significant functions within a system. A caution at this point is to **not** consider these process steps completely serial in nature. Although they are presented in a near serial format (for ease of reading and understanding), there are many activities that will require parallel processing and effort from the safety manager and engineer. Activities as complicated and as interface-dependent as a software

development within a systems acquisition process will seldom have required tasks line up where one task is complete before the next one begins. This is clearly demonstrated by the development of a SSS program and milestone schedule (see paragraph 4.3.1)

This section of the Handbook describes the tasks associated with contract and deliverable data development (including methods for tailoring), safety-critical function identification, preliminary and detailed hazard analysis, safety-specific requirements identification, implementation, test and verification, and residual risk analysis and acceptance. It also includes the participation in trade studies and design alternatives.

2.7.3 Software Risk Assessment and Acceptance

The risk assessment and acceptance portion of Section 4 focuses on the tasks identifying residual safety risk in the design, test, and operation of the system. It includes the evaluation and categorization of hazards remaining in the system and their impact to operations, maintenance, and support functions. It also includes the graduated levels of programmatic sign-off for hazard and failure mode records of the Subsystem, System, and Operations and Support Hazard Analyses. This section includes the tasks required to identify the hazards remaining in the system, assess their safety risk impact with their severity, probability or software control criticality, and determine the residual safety risk.

2.7.4 Supplementary Appendices

The Handbook's appendices include acronyms, definition of terms, references, supplemental system safety information, generic safety requirements and guidelines, and lessons learned pertaining to the accomplishment of the SSS tasks.

3. Introduction to Risk Management and System Safety

3.1 Introduction

SSS Team members who are not familiar with system safety should read this section. It should also be read by anyone who feels a need to become more familiar with the concept of the HRI and how hazards are rationally assessed, analyzed, correlated, and tracked.

Section 3 will discuss the following:

- Risk and its application to the SwSSP
- Programmatic risks
- Safety risks

3.2 A Discussion of Risk

Everywhere that we turn, we are surrounded by a multitude of risks, some large and some so minimal that they can easily be overlooked, but all demanding to be recognized (i.e., assessed) and dealt with (i.e., managed). We view taking risks as foolhardy, irrational, and to be avoided. Risks imposed on us by others are generally considered to be entirely unacceptable. Everything that we do involves risk. It is an unavoidable part of our everyday lives.

Realistically, some risk of mishap must be accepted. Systems are hardly ever risk free. A totally safe aircraft for instance will never fly, as the potential for a crash is still possible if it becomes airborne. The residual safety risk in the fielded system is the direct result of the accuracy and comprehensiveness of the SSP. How much risk is accepted or not accepted, is the prerogative of management on any give acquisition program. That decision is affected by a great deal of input. As tradeoffs are being considered and the design progresses, it may become evident that some of the safety parameters are forcing higher program risk. From the PM's perspective, a relaxation of one or more of the safety requirements may appear to be advantageous when considering the broader perspective of cost and performance optimization. The PM will frequently make a decision against the recommendation of his system safety manager. The system safety manager must recognize such management prerogatives. However, the prudent PM must make the decision whether to fix the identified problem or formally document acceptance of the added risk. Of course, responsibility of personnel injury or loss of life changes the picture considerably. When the PM decides to accept risk, the decision must be coordinated with all affected organizations and then documented; so that in future years, everyone will know and understand the elements of the decision and why it was made.

Accepting risk is an action of both risk assessment and risk management. Risk acceptance is not as simple a matter as it may first appear. Several points must be kept in mind:

- Risk is a fundamental reality.
- Risk management is a process of tradeoffs.

- Quantifying risk does not ensure safety.
- Risk is a matter of perspective.

Risk Perspectives. When discussing risk, we must distinguish between three different standpoints, which are as follows:

- Standpoint of an INDIVIDUAL exposed to a hazard.
- Standpoint of SOCIETY. Besides being interested in guaranteeing minimum individual risk for each of its members, society is concerned about the total risk to the general public.
- Standpoint of the INSTITUTION RESPONSIBLE FOR THE ACTIVITY. The institution responsible for an activity can be a private company or a government agency. From their point of view, it is essential to keep individual risks to employees or other persons and the collective risk at a minimum. An institution's concern is also to avoid catastrophic accidents.

The system safety effort is an optimizing process that varies in scope and scale over the lifetime of the system. SSP balances are the result of the interplay between system safety and the three very familiar basic program elements: cost, performance, and schedule. Without an acute awareness of the system safety balance on the part of both the PM and the system safety manager, they cannot discuss when, where, and how much they can afford to spend on system safety. We cannot afford mishaps that will prevent the achievement of the primary mission goal, nor can we afford systems that cannot perform because of overstated safety goals.

Safety Management's Risk Review. The SSP examines the interrelationships of all components of a program and its systems with the objective of bringing mishap risk or risk reduction into the management review process for automatic consideration in total program perspective. It involves the preparation and implementation of system safety plans; the performance *of* system safety analyses on both system design and operations, and risk assessments in support of both management and system engineering activities. The system safety activity provides the manager with a means of identifying what the risk of mishap is, where a mishap can be expected to occur, and what alternate designs are appropriate. Most important, it verifies implementation and effectiveness of hazard control. What is generally not recognized in the system safety community is that there are no safety problems in system design. There are only engineering and management problems, which if left unresolved, can result in a mishap. When a mishap occurs, then it is a safety problem. Identification and control of mishap risk is an engineering and management function. This is particularly true of software safety risk.

3.3 Types of Risk

There are various models describing risks that are listed below. The model in Figure 3-1 follows the system safety concept of risk reduction.

Total Risk is the sum of identified and unidentified risks.

Identified Risk is that risk which has been determined through various analytical techniques. The first task of system safety is to make identified risk as large a piece of the overall pie as practical. The time and costs of analytical efforts, the quality of the safety program, and the state of technology impact the amount of risk identified.

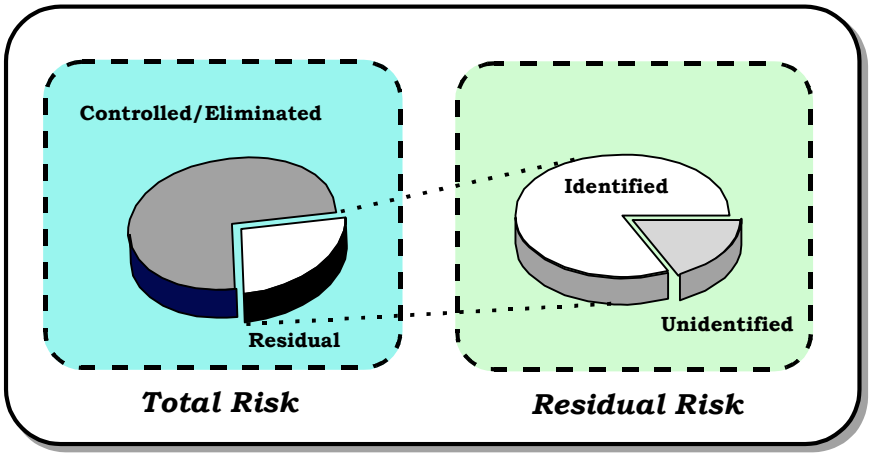


Figure 3-1: Types of Risk

Unacceptable Risk is that risk which cannot be tolerated by the managing activity. It is a subset of identified risk that is either eliminated or controlled.

Residual Risk is the risk left over after system safety efforts have been fully employed. It is sometimes erroneously thought of as being the same as acceptable risk. Residual risk is actually the sum of unacceptable risk (uncontrolled), acceptable risk and unidentified risk. This is the total risk passed on to the user that may contain some unacceptable risk.

Acceptable Risk is the part of identified risk that is allowed to persist without further engineering or management action. It is accepted by the managing activity. However, it is the user who is exposed to this risk.

Unidentified Risk is the risk that has not been determined. It is real. It is important, but it cannot be measured. Some unidentified risk is subsequently determined and measured when a mishap occurs. Some risk is never known.

3.4 Areas of Program Risk

Within the DOD, risk is defined as a potential occurrence that is detrimental to either plans or programs. This risk is measured as the combined effect of the likelihood of the occurrence and a measured or assessed consequence given that occurrence (DSMC 1990). The perceived risk to a program is usually different between program management, systems engineers, users, and safety. Because of this, the responsibility of defining program risk is usually assigned to a small group of individuals from various disciplines that can evaluate the program risks from a broad perspective of the total program concepts and issues to include business, cost, schedule, technical, and programmatic considerations. Although risk management responsibility is assigned to an individual group, the successful management of a program's risk is dependent on contributions

and inputs of all individuals involved in the program management and engineering design functional activities.

In DOD, this risk management group is usually assigned to (or contained within) the systems engineering group. They are responsible for identifying, evaluating, measuring, and resolving risk within the program. This includes recognizing and understanding the warning signals that may indicate that the program, or elements of the program, is off track. This risk management group must also understand the seriousness of the problems identified and then develop and implement plans to reduce the risk. A risk management assessment must be made early in the development life cycle and the risks must continually be reevaluated throughout the development life cycle. The members of the risk management group and the methods of risk identification and control should be documented in the program's Risk Management Plan.

Risk management[17] must consist of three activities:

Risk Planning – This is the process to force organized, purposeful thought to the subject of eliminating, minimizing, or containing the effects of undesirable consequences.

Risk Assessment – This is the process of examining a situation and identifying the areas of potential risk. The methods, techniques, or documentation most often used in risk assessment include the following:

Systems engineering documents

Operational Requirements Document

Operational Concepts Document

Life cycle cost analysis and models

Schedule analysis and models

Baseline cost estimates

Requirements documents

Lessons learned files and databases

Trade studies and analyses

Technical performance measurements and analyses

Work Breakdown Structures (WBS)

Project planning documents

Risk Analysis – This is the process of determining the probability of events and the potential consequences associated with those events relative to the program. The purpose of a risk analysis is to discover the cause, effects, and magnitude of the potential risk, and

[17] Selected descriptions and definitions regarding risk management are paraphrased from the DSMC, Systems Engineering Management Guide, January 1990

to develop and examine alternative actions that could reduce or eliminate these risks. Typical tools or models used in risk analysis include the following:

Schedule Network Model

Life Cycle Cost Model

Quick Reaction Rate/Quantity Cost Impact Model

System Modeling and Optimization

To further the discussion of program risk, short paragraphs are provided to help define schedule, budget, sociopolitical, and technical risk. Although safety, by definition, is a part of technical risk, it can impact all areas of programmatic risk as described in subsequent paragraphs. This is what ties safety risk to technical and programmatic risk.

3.4.1 Schedule Risk

The master systems engineering and software development schedule for a program contains numerous areas of programmatic risk, such as schedules for new technology development, funding allocations, test site availability, critical personnel availability and rotation, etc. Each of these has the potential for delaying the development schedule and can induce unwarranted safety risk to the program. While these examples are by no means the only source of schedule risk, they are common to most programs. The risk manager must identify, analyze, and control risks to the program schedule by incorporating positive measures into the planning, scheduling, and coordinating activities for the purpose of minimizing their impact to the development program.

To help accomplish these tasks, the systems engineering function maintains the Systems Engineering Master Schedule (SEMS) and the Systems Engineering Detailed Schedule (SEDS). Maintaining these schedules helps to guide the interface between the customer and the developer, provides the cornerstone of the technical status and reporting process, and provides a disciplined interface between engineering disciplines and their respective system requirements. An example of the integration, documentation, tracking, and tracing of risk management issues is depicted in Figure 3-2. Note that the SEMS and SEDS schedules, and the risk management effort are supported by a risk issue table and risk management database. These tools assist the risk manager in the identification, tracking, categorization, presentation, and resolution of managerial and technical risk.

Software developers for DOD customers or agencies are said to have maintained a perfect record to date. That is, they have never yet delivered a completed (meets all user/program requirements and specifications) software package on time[18]. While this may be arguable, the inference is worthy of consideration. It implies that schedule risk is an important issue on a software development program. The schedule can become the driving factor forcing the delivery of immature and improperly tested critical software product to the customer. The risk manager, in concert with the safety manager, must ensure that the delivered product does not introduce safety risk to the user, system, maintainer, or the environment that is considered unacceptable. This is

[18] Paraphrased from comments made at the National Workshop on Software Logistics, 1989

accomplished by the implementation of a SwSSP (and safety requirements) early in the software design process. The end result should produce a schedule risk reduction by decreasing the potential for re-design and re-code of software possessing safety deficiencies.

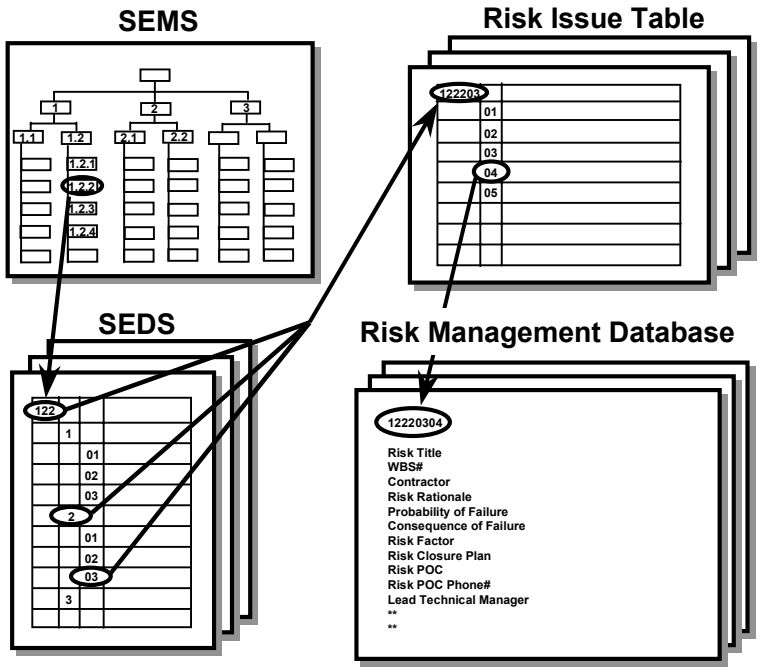


Figure 3-2: Systems Engineering, Risk Management Documentation

3.4.2 Budget Risk

Almost hand-in-hand with schedule risk comes budget risk. Although they can be mutually exclusive, that is seldom the case. The lack of monetary resources is always a potential risk in a development program. Within the DOD research, acquisition, and development agencies, the potential for budget cuts or congressionally mandated program reductions always seems to be lurking around the next corner. Considering this potential, budgetary planning, cost scheduling, and program funding coordination become paramount to the risk management team. They must ensure those budgetary plans for current- and out-years are accurate and reasonable, and those potential limitations or contingencies to funding are identified, analyzed, and incorporated into the program plans.

In system safety terms, the development of safety-critical software requires significant program resources, highly skilled engineers, increased training requirements, software development tools, modeling and simulation, and facilities and testing resources. To ensure that this software meets functionality, safety, and reliability goals, these activities become “drivers” for both the budget and schedule of a program. Therefore, the safety manager must ensure that all safety-specific software development and test functions are prioritized in terms of safety risk potential to the program and to the operation of software after implementation. The prioritization of safety hazards and failure modes, requirements, specifications, and test activities attributed to software help to facilitate and support the tasks performed by the risk management team. It will help them

understand, prioritize, and incorporate the activities necessary to minimize the safety risk potential for the program.

3.4.3 Sociopolitical Risk

This is a difficult subject to grasp from a risk management perspective. It is predicated more on public and political perceptions than basic truth and fact. Examples of this type of risk are often seen during the design, development, test, and fielding of a nuclear weapon system in a geographical area that has a strong public, and possibly political, resistance. With an example like this in mind, several programmatic areas become important for discussion. First, program design, development, and test results have to be predicated on complete, technical fact. This will preclude any public perception of attempts to hide technical shortfalls or safety risk. Second, social and political perceptions can generate programmatic risk that must be considered by the risk managers. This includes the potential for budget cuts, schedule extensions, or program delays due to funding cuts as a result of public outcry and protest and its influence on politicians.

Safety plays a significant role in influencing the sensitivities of public or political personages toward a particular program. It must be a primary consideration in assessing risk management alternatives. If an accident (even a minor accident without injury) should occur during a test, it could result in program cancellation.

The sociopolitical risk may also change during the life cycle of a system. For example, explosive handling facilities once located in isolated locations often find residential areas encroaching. Protective measures adequate for an isolated facility may not be adequate as residents, especially those not associated with the facility, move closer. While the PM cannot control the later growth in population, he/she must consider this and other factors during the system development process.

3.4.4 Technical Risk

Technical risk is where safety risk is most evident in system development and procurement. It is the risk associated with the implementation of new technologies or new applications of existing technologies into the system being developed. These include the hardware, software, human factors interface, and environmental safety issues associated with the design, manufacture, fabrication, test, and deployment of the system. Technical risk results from poor identification and incorporation of system performance requirements to meet the intent of the user and system specifications. The inability to incorporate defined requirements into the design (lack of a technology base, lack of funds, lack of experience, etc.) increases the technical risk potential.

Systems engineers are usually tasked with activities associated with risk management. This is due to their assigned responsibility for technical risk within the design engineering function. The systems engineering function performs specification development, functional analysis, requirements allocation, trade studies, design optimization and effectiveness analysis, technical interface compatibility, logistic support analysis, program risk analysis, engineering integration and control, technical performance measurement, and documentation control.

The primary objective of risk management in terms of software safety is to understand that safety risk is a part of the technical risk of a program. All program risks must be identified, analyzed, and either eliminated or controlled. This includes safety risk, and thus, software (safety) risk.

3.5 System Safety Engineering

To understand the concept of system safety as it applies to software development, the user needs a basic introduction and description of system safety since software is a subset of the SSP activities. "System safety as we know it today began as a grass roots movement that was introduced in the 40s, gained momentum during the 50s, became established in the 60s, and formalized its place in the acquisition process in the 70s. The system safety concept was not the brain child of one person, but rather a call from the engineering and safety community to design and build safer equipment by applying lessons learned from our accident investigations."[19] It grew out of "conditions arising after WW II when its "parent" disciplines - systems engineering and systems analysis were developed to cope with new and complex engineering problems."[20] System safety evolved alongside and in conjunction with, systems engineering and systems analysis. Systems engineering considers "the overall process of creating a complex human - machine system and systems analysis providing (1) the data for the decision - making aspects of that process and (2) an organized way to select among the latest alternative designs."[21] In the 1950s intense political pressure focused on safety following several catastrophic mishaps. These included incidents where Atlas and Titan intercontinental ballistic missiles exploded in their silos during operational testing. Investigation of the cause of these accidents revealed that a large percentage of the causal factors could be traced to deficiencies in design, operation, and management that could have, and should have, been detected and corrected prior to placing the system in service. This recognition led to the development of system safety approaches to identify and control hazards in the design of the system in order to minimize the likelihood and/or severity of first-time accidents.

As system safety analytical techniques and managerial methods evolved, they have been documented in various government and commercial standards. The first system safety specification was a document created by the Air Force in 1966, MIL-S-38130A. In June 1969, MIL-STD-882 replaced this standard and a SSP became mandatory for all DOD-procured products and systems. Many of the later system safety requirements and standards in industry and other government agencies were developed based on MIL-STD-882, and remain so today. As both DOD and NASA began to use computers/software increasingly to perform critical system functions, concern about the safety aspects of these components began to emerge. The DOD initiated efforts to integrate software into SSPs in the 1980s with the development of an extensive set of software safety tasks (300 series tasks) for incorporation into MIL-STD-882B (Notice 1).

[19] Air Force System Safety Handbook, August 1992
[20] Leveson, Nancy G., Safeware, System Safety and Computers, 1995, Addison Wesley, page 129
[21] Ibid, page 143

The identification of separate software safety tasks in MIL-STD-882B focused engineering attention on the hazard risks associated with the software components of a system and its critical effect on safety. However, the engineering community perceived them as "segregated" tasks to the overall system safety process as system safety engineers tried to push the responsibility for performing these tasks onto software engineers. Since software engineers had little understanding of the "system safety" process and of the overall system safety functional requirements, this was an unworkable process for dealing with SSRs. Therefore, the separate software safety tasks were not included in MIL-STD-882C as separate tasks, but were integrated into the overall system-related safety tasks. In addition, software engineers were given a clear responsibility and a defined role in the SSS process in MIL-STD-498.

MIL-STD-882 defines system safety as:

"*The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle*."

SSP objectives can be further defined as follows:

a. Safety, consistent with mission requirements, is designed into the system in a timely, cost-effective manner.

b. Hazards associated with systems, subsystems, or equipment are identified, tracked, evaluated, and eliminated; or their associated risk is reduced to a level acceptable to the Managing Authority (MA) by evidence analysis throughout the entire life cycle of a system.

c. Historical safety data, including lessons learned from other systems, are considered and used.

d. Minimum risk consistent with user needs is sought in accepting and using new design technology, materials, production, tests, and techniques. Operational procedures must also be considered.

e. Actions taken to eliminate hazards or reduce risk to a level acceptable to the MA are documented.

f. Retrofit actions required to improve safety are minimized through the timely inclusion of safety design features during research, technology development for, and acquisition of a system.

g. Changes in design, configuration, or mission requirements are accomplished in a manner that maintains a risk level acceptable to the MA.

h. Consideration is given early in the life cycle to safety and ease of disposal [including Explosive Ordnance Disposal (EOD)], and demilitarization of any hazardous materials associated with the system. Actions should be taken to minimize the use of hazardous materials and, therefore, minimize the risks and life cycle costs associated with their use.

i. Significant safety data are documented as "lessons learned" and are submitted to data banks or as proposed changes to applicable design handbooks and specifications.

j. Safety is maintained and assured after the incorporation and verification of Engineering Change Proposals (ECP) and other system-related changes.

With these definitions and objectives in mind, the system safety manager/engineer is the primary individual(s) responsible for the identification, tracking, elimination, and/or control of hazards or failure modes that exist in the design, development, test, and production of both hardware and software. This includes their interfaces with the user, maintainer, and the operational environment. System safety engineering is a proven and credible function supporting the design and systems engineering process. The steps in the process for managing, planning, analyzing, and coordinating system safety requirements are well established and when implemented, successfully meet the above stated objectives.

These general SSP requirements are as follows:

a. Eliminate identified hazards or reduce associated risk through design, including material selection or substitution.

b. Isolate hazardous substances, components, and operations from other activities, areas, personnel, and incompatible materials.

c. Locate equipment so that access during operations, servicing, maintenance, repair, or adjustment minimizes personnel exposure to hazards.

d. Minimize risk resulting from excessive environmental conditions (e.g., temperature, pressure, noise, toxicity, acceleration, and vibration).

e. Design to minimize risk created by human error in the operation and support of the system.

f. Consider alternate approaches to minimize risk from hazards that cannot be eliminated. Such approaches include interlocks; redundancy; fail-safe design; fire suppression; and protective clothing, equipment, devices, and procedures.

g. Protect power sources, controls, and critical components of redundant subsystems by separation or shielding.

h. Ensure personnel and equipment protection (when alternate design approaches cannot eliminate the hazard) provide warning and caution notes in assembly, operations, maintenance, and repair instructions as well as distinctive markings on hazardous components and materials, equipment, and facilities. These shall be standardized in accordance with MA requirements.

i. Minimize severity of personnel injury or damage to equipment in the event of a mishap.

j. Design software-controlled or monitored functions to minimize initiation of hazardous events or mishaps.

k. Review design criteria for inadequate or overly restrictive requirements regarding safety. Recommend a new design criterion supported by study, analyses, or test data.

A good example of the need for, and the credibility of, a system safety engineering program is the Air Force aircraft mishap rate improvement since the establishment of the SSP in the design, test, operations, support, and training processes. In the mid-1950s, the aircraft mishap rates were over 10 per 100,000 flight hours. Today, this rate has been reduced to less than 1.25 per 100,000 flight hours.

Further information regarding the management and implementation of system safety engineering (and the analyses performed to support the goals and objectives of a SSP) is available through numerous technical resources. It is not the intent of this Handbook to become another technical source book for the subject of system safety, but to address the implementation of SSS within the discipline of system safety engineering. If specific system safety methods, techniques, or concepts remain unclear, please refer to the list of references in Appendix B for supplemental resources relating to the subject matter.

With the above information regarding system safety engineering (as a discipline) firmly in tow, a brief discussion must be presented as it applies to hazards and failure mode identification, categorization of safety risk in terms of probability and severity, and the methods of resolution. This concept must be firmly understood as the discussion evolves to the accomplishment of software safety tasks within the system safety engineering discipline.

3.6 Safety Risk Management

The process of system safety management and engineering is "deceptively simple"[22] although it entails a great deal of work. It is aimed at identifying system hazards and failure modes, determining their causes, assessing hazard severity and probability of occurrence, determining hazard control requirements, verifying their implementation, and identifying and quantifying any residual risk remaining prior to system deployment. Within an introduction of safety risk management, the concepts of hazard identification, hazard categorization, and hazard risk reduction must be presented. Safety risk management focuses on the safety aspects of technical risk as it pertains to the conceptual system proposed for development. It attempts to identify and prioritize hazards that are most severe, and/or have the greatest probability of occurrence. The safety engineering process then identifies and implements safety risk elimination or reduction requirements for the design, development, test, and system activation phases of the development life cycle.

As the concept of safety risk management is further defined, keep in mind that the defined process is relatively simple and that the effort to control safety risk on a program will most likely be reduced if the process is followed.

[22] System Safety Analysis Handbook, A Resource Book For Safety Practitioners

3.6.1 Initial Safety Risk Assessment

The efforts of the SSE are launched by the performance of the initial safety risk assessment of the system. In the case of most DOD procurements, this takes place with the accomplishment of the Preliminary Hazard List (PHL) and the PHA. These analyses are discussed in detail later in this Handbook. The primary focus here is the assessment and analysis of hazards and failure modes that are evident in the proposed system. This section of the Handbook will focus on the basic principles of system safety and hazard resolution. Specific discussions regarding how software influences or is related to hazards will be discussed in detail in Section 4.

3.6.1.1 Hazard and Failure Mode Identification

A hazard is defined as follows: *A condition that is prerequisite to a mishap.* The SSE identifies these conditions, or hazards. The initial hazard analysis, and the Failure Modes and Effects Analysis (FMEA) accomplished by the reliability engineer, provides the safety information required to perform the initial safety risk assessment of identified hazards. Without identified hazards and failure modes, very little can be accomplished to improve the overall safety of a system (remember this fact as software safety is introduced). Identified hazards and failure modes become the basis for the identification and implementation of safety requirements within the design of the system. Once the hazards are identified, they must be categorized in terms of safety risk.

3.6.1.2 Hazard Severity

The first step in classifying safety risk requires the establishment of hazard severity within the context of the system and user environments. This is typically done in two steps, first using the severity of damage and then applying the number of times that the damage might occur. Table 3-1 provides an example of how severity can be qualified.

Table 3-1: Hazard Severity

For Example Purposes Only

DESCRIPTION	*CATEGORY*	*SEVERITY OF HAZARD EFFECT*
Catastrophic	I	Death Or System Loss
Critical	II	Severe Injury, Occupational Illness, Major System Or Environmental Damage
Marginal	III	Minor Injury, Occupational Illness, Minor System Or Environmental Damage
Negligible	IV	Less Than Minor Injury, Illness, System Damage Or Environmental Damage

Note that this example is typical to the MIL-STD-882-specified format. As you can see, the "severity of hazard effect" is qualitative and can be modified to meet the special needs of a program. There is an important aspect of the graphic to remember for any procurement. In order to assess safety severity, a benchmark to measure against is essential. The benchmark allows for the establishment of a qualitative baseline that can be communicated across programmatic and technical interfaces. It must be in a format language that makes sense among individuals and between program interfaces.

3.6.1.3 Hazard Probability

The second half of the equation for the determination of safety risk is the identification of the probability of occurrence. The probability that a hazard or failure mode may occur, given that it is not controlled, can be determined by numerous statistical techniques. These statistical probabilities are usually obtained from reliability analysis pertaining to hardware component failures acquired through qualification programs. Component failure rates from reliability engineering are not always obtainable. This is especially true on advanced technology programs where component qualification programs do not exist and "one-of-a-kind" items are procured. Thus, the quantification of probability to a desired confidence level is not always possible for a specific hazard. When this occurs, alternative techniques of analysis are required for the qualification or quantification of hazard probability of hardware nodes. Examples of credible alternatives include Sensitivity Analysis, Event Tree Diagrams, and FTAs. An example of the categorization of probability is provided in Table 3-2 and is similar to the format recommended by MIL-STD-882.

Table 3-2: Hazard Probability

For Example Purposes Only

DESCRIPTION	*LEVEL*	*PROGRAM DESCRIPTION*	*PROBABILITY*
Frequent	**A**	**Will Occur**	**1 in 100**
Probable	**B**	**Likely To Occur**	**1 in 1000**
Occasional	**C**	**Unlikely To Occur, But Possible**	**1 in 10,000**
Remote	**D**	**Very Unlikely To Occur**	**1 in 100,000**
Improbable	**E**	**Assume It Will Not Occur**	**1 in 1,000,000**

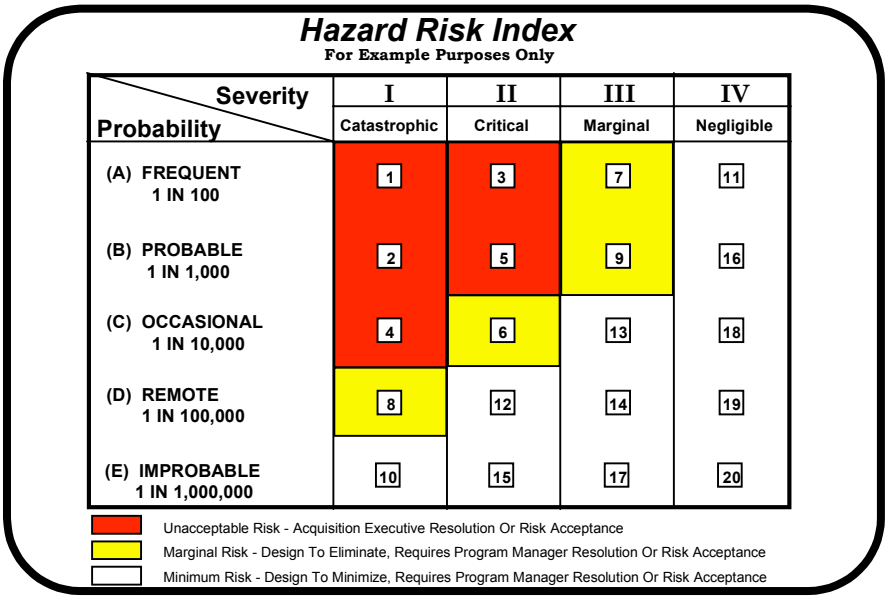
As with the example provided for hazard severity, Table 3-2 can be modified to meet the specification requirements of the user and/or developer. A systems engineering team (to include system safety engineering) may choose to shift the probability numbers an order of magnitude in

either direction or to include, or reduce, the number of categories. All of the options are acceptable if the entire team is in agreement. This agreement must definitely include the customer's opinions and specification requirements. Also of importance when considering probability categories is the inclusion of individual units, entire populations, and time intervals (periods) which are appropriate for the system being analyzed.

3.6.1.4 HRI Matrix

Hazard severity and hazard probability when integrated into a table format produces the HRI matrix, and the initial HRI risk categorization for hazards prior to control requirements is implemented. An example of a HRI matrix is provided as Table 3-3. This example was utilized on a research and development activity where little component failure data was available. This matrix is divided into three levels of risk as indicated by the grayscale legend beneath the matrix. HRIs of 1-5 (the darkest gray) are considered unacceptable risk. These risks are considered high and require resolution or acceptance from the Acquisition Executive (AE). HRIs of 6, 7, 8, and 9 (the medium gray) are considered to be marginal risk; while HRIs of 10 through 16 are minimum risk. Those hazards deemed marginal should be redesigned for elimination and require PM approval or risk acceptance. Those hazards deemed minimum should be redesigned to further minimize their risk and require PM approval or risk acceptance. HRIs 10-16 were considered lower risk hazards and were put under the cognizance of the lead design engineer and the safety manager.

Table 3-3: HRI Matrix



The true benefit of the HRI matrix is the ability and flexibility to prioritize hazards in terms of severity and probability. This prioritization of hazards allows the PM, safety manager, and engineering manager the ability to also prioritize the expenditure and allocation of critical resources. Although it seems simplistic, a hazard with an HRI of 11 should have fewer resources expended in its analysis, design, test, and verification than a hazard of 4. Without the availability of the HRI matrix, the allocation of resources becomes more arbitrary.

Another benefit of the HRI matrix, is the accountability and responsibility of program and technical management to the system safety effort. The SwSSP identifies and assigns specific levels of management authority with the appropriate levels of safety hazard severity and probability. The HRI methodology holds program management and technical engineering accountable for the safety risk of the system during design, test and operation, and the residual risk upon delivery to the customer.

From the perspective of the safety analyst, the HRI is a tool that is used during the entire system safety effort throughout the product life cycle. Note, however, that the HRI as a tool is more complex when applied to the evaluation of system hazards and failure modes influenced by software inputs or software information. Alternatives to the classical HRI are discussed in detail in Section 4.

3.6.2 Safety Order of Precedence

The ability to adequately eliminate or control safety risk is predicated on the ability to accomplish the necessary tasks early in the design phases of the acquisition life cycle. For example, it is more cost effective and technologically efficient to eliminate a known hazard by changing the design (on paper), than retrofitting a fleet in operational use. Because of this, the system safety engineering methodology employs a safety order of precedence for hazard elimination or control. When incorporated, the design order of precedence further eliminates or reduces the severity and probability of hazard/failure mode initiation and propagation throughout the system. The following is extracted from MIL-STD-882C, subsection 4.4.

a. Design for Minimum Risk - From the first, design to eliminate hazards. If an identified hazard cannot be eliminated, reduce the associated risk to an acceptable level, as defined by the MA, through design selection.

b. Incorporate Safety Devices - If identified hazards cannot be eliminated or their associated risk adequately reduced through design selection, that risk shall be reduced to a level acceptable to the MA through the use of fixed, automatic, or other protective safety design features or devices. Provisions shall be made for periodic functional checks of safety devices when applicable.

c. Provide Warning Devices - When neither design nor safety devices can effectively eliminate identified hazards or adequately reduce associated risk, devices shall be used to detect the condition and to produce an adequate warning signal to alert personnel of the hazard. Warning signals and their application shall be designed to minimize the probability of incorrect personnel reaction to the signals and shall be standardized within like types of systems.

d. Develop Procedures and Training - Where it is impractical to eliminate hazards through design selection or adequately reduce the associated risk with safety and warning devices, procedures and training shall be used. However, without a specific waiver from the MA, no warning, caution, or other form of written advisory shall be used as the only risk reduction method for Category I or II hazards. Procedures may include the use of personal protective equipment. Precautionary notations shall be standardized

as specified by the MA. Tasks and activities judged to be safety-critical by the MA may require certification of personnel proficiency.

3.6.3 Elimination or Risk Reduction

The process of hazard and failure mode elimination or risk reduction is based on the design order of precedence. Once hazards and failure modes are identified by evidence analysis and categorized, specific (or functionally derived) safety requirements must be identified for incorporation into the design for the elimination or control of safety risk. Defined requirements can be applicable for any of the four categories of the defined order of safety precedence. For example, a specific hazard may have several design requirements identified for incorporation into the system design. However, to further minimize the safety risk of the hazard, supplemental requirements may be appropriate for safety devices, warning devices, and operator/maintainer procedures and training. In fact, most hazards have more than one design or risk reduction requirement unless the hazard is completely eliminated through the first (and only) design requirement. Figure 3-3 shows the process required to eliminate or control safety risk via the order of precedence described in Paragraph 3.6.2.

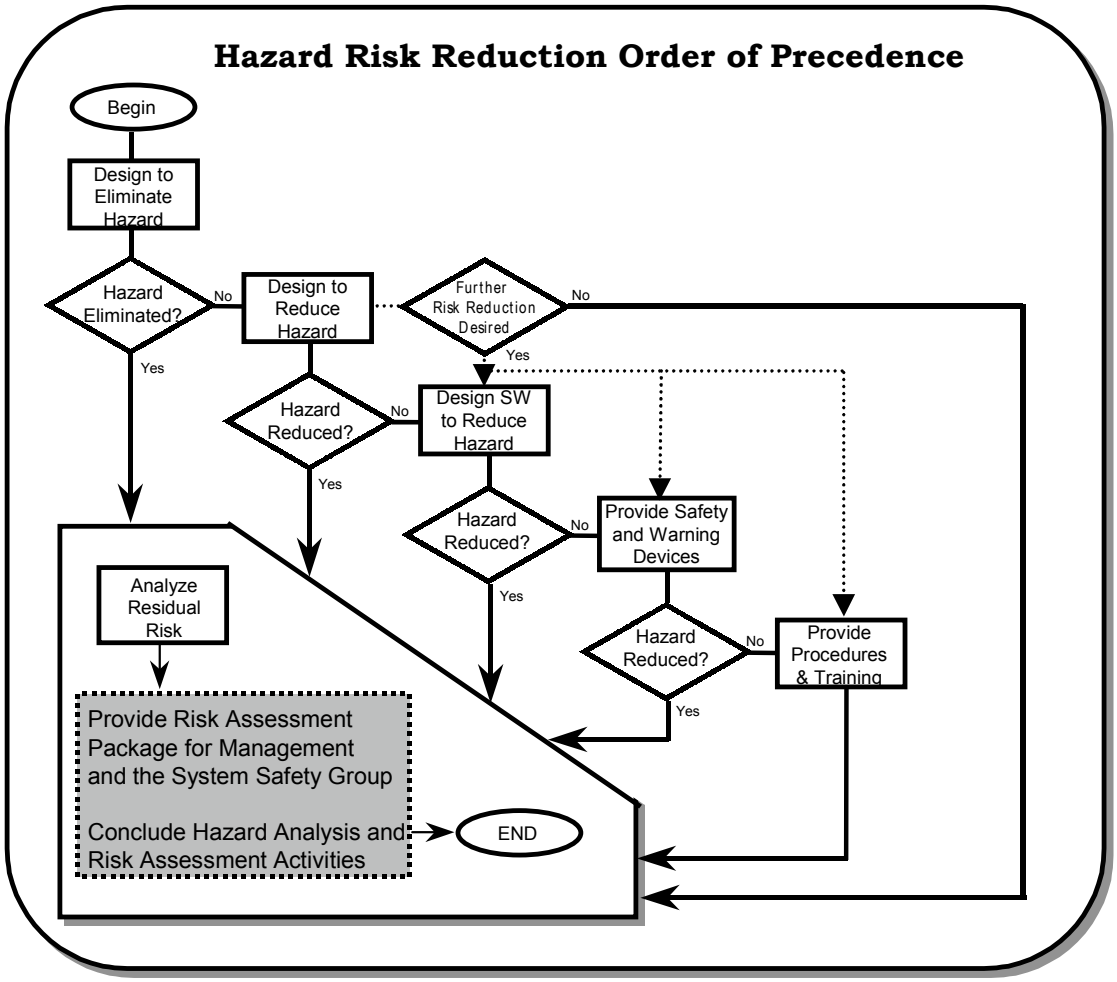


Figure 3-3: Hazard Reduction Order of Precedence

Identification of safety-specific requirements to the design and implementation portions of the system does not complete the safety task. The safety engineer must verify that the derived requirements have indeed been implemented as intended. Once hazard elimination and control requirements are identified and communicated to the appropriate design engineers, testing requirements must be identified for hazards which have been categorized as safety-critical. The categorization of safety risk in accordance with severity and probability must play a significant role in the depth of testing and requirements verification methods employed. Very low risk hazards do not require the same rigor of safety testing to verify the incorporation of requirements as compared to those associated with safety-critical hazards. In addition, testing cannot always be accomplished whereas verification methods may be appropriate (i.e., designer sign-off on hazard record, as-built drawing review, inspection of manufactured components, etc.)

3.6.4 Quantification of Residual Safety Risk

After the requirements are implemented (to the extent possible), and appropriately verified in the design, the safety engineer must analyze each identified and documented hazard record to assess and analyze the residual risk that remains within the system during its operations and support activities. This is the same risk assessment process that was performed in the initial analysis described in Paragraph 3.6.1. The difference in the analysis is the amount of design and test data to support the risk reduction activities. After the incorporation of safety hazard elimination or reduction requirements, the hazard is once again assessed for severity, probability of occurrence, and an HRI determined. A hazard with an initial HRI of 4 may have been reduced in safety risk to an HRI of 8. However, since in this example, the hazard was not completely eliminated and only reduced; there remains a residual safety risk. Granted, it is not as severe or as probable as the original; but the hazard does exist. Remember that the initial HRI of a hazard is determined during the PHA development prior to the incorporation or implementation of requirements to control or reduce the safety risk and is often an initial engineering judgment. The final HRI categorizes the hazard after the requirements have been implemented and verified by the developer. If hazards are not reduced sufficiently to meet the safety objectives and goals of the program, they must be reintroduced to safety engineering for further analyses and safety risk reduction. It should be noted that risk is generally reduced within a probability category. Risk reduction across severity levels generally requires a hardware design change.

In conjunction with the safety analysis, and the available engineering data and information available, residual safety risk of the system, subsystems, user, maintainer, and tester interfaces must be quantified. Hazard records with remaining residual risk must be correlated within subsystems, interfaces, and the total system for the purpose of calculating the remaining risk. This risk must be communicated in detail [via the System Safety Working Group (SSWG) and the detailed hazard record system], to the PM, the lead design engineers, the test manager, and the user and fully documented in the hazard database record. If residual risk in terms of safety is unacceptable to the PM, further direction and resources must be provided to the engineering effort.

3.6.5 Managing and Assuming Residual Safety Risk

Managing safety risk is another one of those "simple processes" which usually takes a great deal of time, effort, and resources to accomplish. Referring back to Table 3-3, HRI Matrix, specific categories must be established in the matrix to identify the level of management accountability, responsibility, and risk acceptance. Using Table 3-3 as an example, hazards with an HRI of between 1 through 5 are considered unacceptable[23]. These hazards, if not reduced to a lower level below an HRI of 5, cannot be officially closed without the acquisition executive's signature. This forces the accountability of assuming this particular risk to the appropriate level of management (the top manager). However, the PM can officially close hazards from HRI 6 through 20. This is to say that the PM would be at the appropriate level of management to assume the safety risk to reduce the HRI to a lower category.

Recognize that Tables 3-1 through 3-3 are **for purposes of example only**. They provide a graphical representation and depiction of how a program may be set up with three levels of program and technical management. It is ideal to have the PM as the official sign-off for all residual safety risk to maintain safety accountability with that individual. Remember that the PM is responsible for the safety of a product or system at the time of test and deployment. The safety manager must establish an accountability system for the assumption of residual safety risk based upon user inputs, contractual obligations, and negotiations with the PM.

[23] Remember that this is for example purpose only. Within DOD programs HRI 1 through 9 would require the AE's acceptance of risk.

4. Software Safety Engineering

4.1 Introduction

This section of the Handbook will introduce the managerial process and the technical methods and techniques inherent in the performance of software safety tasks within the context of a systems safety engineering and software development program. It will include detailed tasks and techniques for the performance of safety analysis, and for the traceability of SSRs from design to test. It will also provide the current "best practices" which may apply as one considers the necessary steps in establishing a credible and cost-effective SSS program (Figure 4-1).

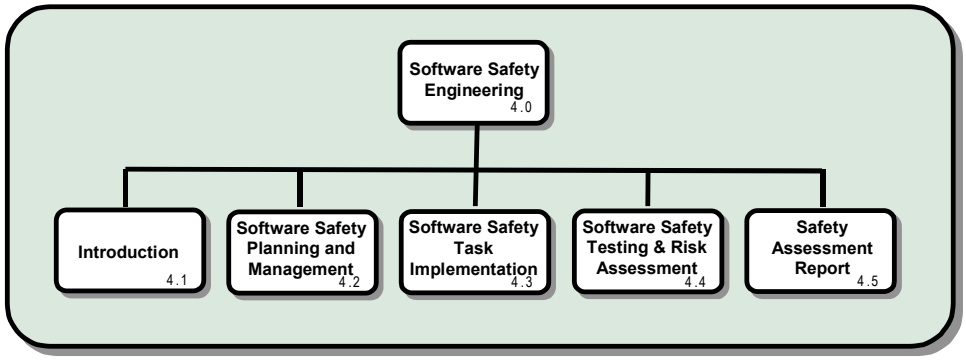


Figure 4-1: Section 4 Contents

Section 4 is applicable to all managerial and technical disciplines. It describes the processes, tools, and techniques to reduce the safety risk of software operating in safety-critical systems.

Its primary purposes are as follows:

- Define a recommended software safety engineering process.
- Describe essential tasks to be accomplished by each professional discipline assigned to the SSS Team.
- Identify interface relationships between professional disciplines and the individual tasks assigned to the SSS Team.
- Identify "best practices" to complete the software safety process and describe each of its individual tasks.
- Recommend "tailoring" actions to the software safety process to identify specific user requirements.

Section 4 should be reviewed and understood by all systems engineers, system safety engineers, software safety engineers, and software development engineers. It is also appropriate for review by PMs interested in the technical aspects of the SSS processes and the possible process improvement initiatives implemented by their systems engineers, software developers, design engineers, and programmers. This section not only describes the essential tasks required by the

system safety engineers, but also the required tasks that must be accomplished by the software safety engineers, systems engineers, and the software developers. This includes the critical communication interfaces between each functional discipline. It also includes the identification, communication, and implementation of initial SSRs and guidelines.

The accomplishment of a software safety management and engineering program requires careful forethought, adequate support from various other disciplines, and timely application of expertise across the entire software development life cycle. Strict attention to planning is required in order to integrate the developer's resources, expertise, and experience with tasks to support contractual obligations established by the customer. This section focuses on the establishment of a software safety program within the system safety engineering and the software development process. It establishes a baseline program that, when properly implemented, will ensure that both initial SSRs and requirements specifically derived from functional hazards analysis are identified, prioritized, categorized, and traced through design, code, test, and acceptance.

A goal of this section of the Handbook is to formally identify the software safety duties and responsibilities assigned to the safety engineer, the software safety engineer, the software engineer, and the managerial and technical interfaces of each through sound systems engineering methods (Figure 4-2). This section of the Handbook will identify and focus on the logical and practical relationships between the safety, design, and software disciplines. It will also provide the reader with the information necessary for the assignment of software safety responsibilities, and the identification of tasks attributed to system safety, software development, as well as hardware and digital systems engineering.

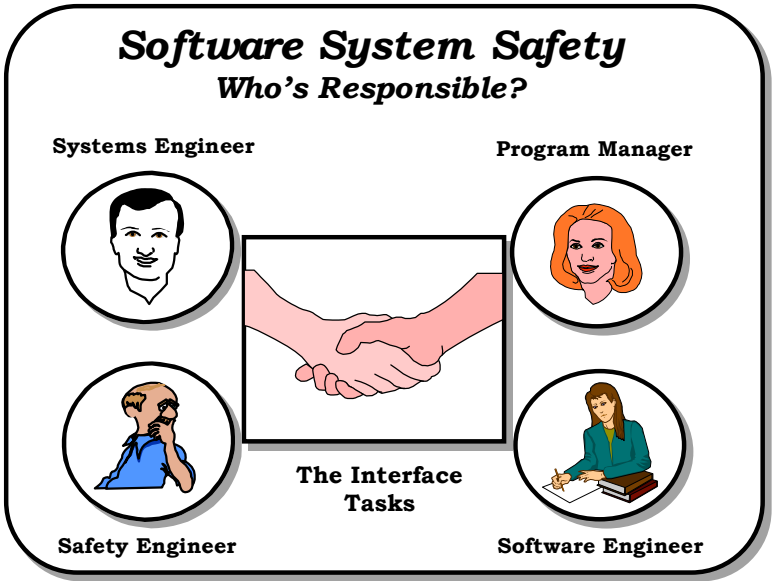


Figure 4-2: Who is Responsible for SSS?

This Handbook assumes a novice's understanding of software safety engineering within the context of system safety and software engineering. Note that many topics of discussion within this section are considered constructs within basic system safety engineering. This is due to the fact that it is impossible to discuss software safety outside of the domain of system safety

engineering and management, systems engineering, software development, and program management.

4.1.1 Section 4 Format

This section is formatted specifically to present both graphical and textual descriptions of the managerial and technical tasks that must be performed for a successful software safety-engineering program. Each managerial process task and technical task, method, or technique will be formatted to provide the following:

- Graphical representation of the process step or technical method
- Introductory and supporting text
- Prerequisite (input) requirements for task initiation
- Activities required to perform the task (including interdisciplinary interfaces)
- Associated subordinate tasks
- Critical task interfaces
- A description of required task output(s) and/or product(s)

This particular format helps to explain the inputs, activities, and outputs for the successful accomplishment of activities to meet the goals and objectives of the software safety program. For those that desire additional information, Appendices A-G are provided to supplement the information in the main sections of this Handbook. The appendices are intended to provide practitioners with supplemental information and credible examples for guidance purposes. The titles of the appendices are as follows:

Appendix A - Definition of Terms

Appendix B - References

Appendix C - Handbook Supplemental Information

Appendix D - Generic Software Safety Requirements and Guidelines

Appendix E - Lessons Learned

Appendix F - Process Chart Worksheets

Appendix G - Examples of Contractual Language [RFP, SOW/Statement of Objectives (SOO)]

4.1.2 Process Charts

Each software safety-engineering task possesses a supporting process chart. Each chart was developed for the purpose of providing the engineer with a detailed and complete "road map" for performing software safety engineering within the context of software design, code, and test

activities. Figure 4-3 provides an example of the depth of information considered for each process task. The depth of information presented in this figure includes processes, methods, documents, and deliverables associated with system safety engineering and management activities. However, for the purposes of this Handbook, these process charts were "trimmed" to contain the information deemed essential for the effective management and implementation of the software safety program under the parent SSP. The in-depth process chart worksheets are provided in Appendix G for those interested in this detailed information.

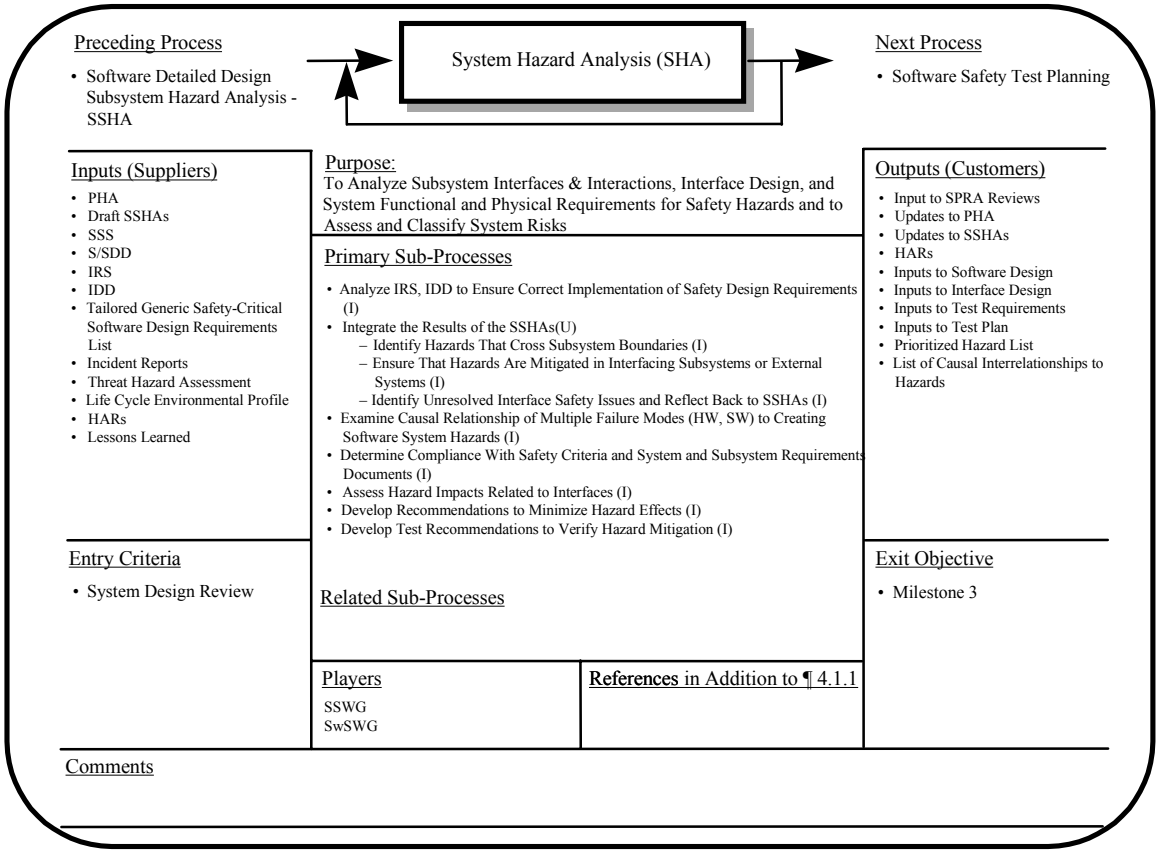


Figure 4-3: Example of Initial Process Chart

Each process chart presented in this handbook will contain the following:

- Primary task description
- Task inputs
- Task outputs
- Primary sub-process tasks
- Critical interfaces

4.1.3 Software Safety Engineering Products

The specific products to be produced by the accomplishment of the software safety engineering tasks are difficult to segregate from those developed within the context of the SSP. It is likely, within individual programs, that supplemental software safety documents and products will be produced to support the system safety effort. These may include supplemental analysis, Data Flow Diagrams (DFD), functional flow analysis, and software requirements specifications (SRS) and the development of Software Analysis Folders (SAF). This Handbook will identify and describe the documents and products that the software safety tasks will either influence or generate. Specific documents include the following:

a. System Safety Program Plan (SSPP)

b. Software Safety Program Plan (SwSPP)

c. Generic Software Safety Requirements List (GSSRL)

d. Safety-Critical Functions List (SCFL)

e. PHL

f. PHA

g. Subsystem Hazard Analysis (SSHA)

h. Safety Requirements Criteria Analysis (SRCA)

i. System Hazard Analysis (SHA)

j. Safety Assessment Report (SAR)

4.2 Software Safety Planning Management

The software safety program must be integrated with and parallel to both the SSP and the software development program milestones. The software safety analyses must provide the necessary input to software development milestones such that safety design requirements, implementation recommendations, or design changes can be incorporated into the software with minimal impact. Program planning precedes all other phases of the SSS program and is perhaps the single most important step in the overall safety program. Inadequately specified safety requirements in the contractual documents generally lead to program schedule and cost impacts later when safety issues arise and the necessary system safety engineering has not been accomplished. The software aspects of system safety tend to be more problematic in this area since the risk associated with the software is often ignored or not well understood until late in the system design. Establishing the safety program and/or performing the necessary safety analyses later in the program results in delays, cost increases, and a less effective safety program.

The history of software-related safety issues, as derived from lessons learned, establishes the need for a practical, logical, and disciplined approach to reducing the safety risk of software performing safety-critical functions within a system. This managerial and engineering discipline must be established “up front” and must be included in the planning activities that both describe

and document the breadth and depth of the program. Detailed planning ensures the identification of critical program interfaces and support and establishes formal lines of communication between disciplines and among engineering functions. Depicted in Figure 4-4, he potential for program success increases through sound planning activities that identify and formalize the managerial and technical interfaces of the program. To minimize the depth of the material presented, supporting and supplemental text is provided in Appendix C.

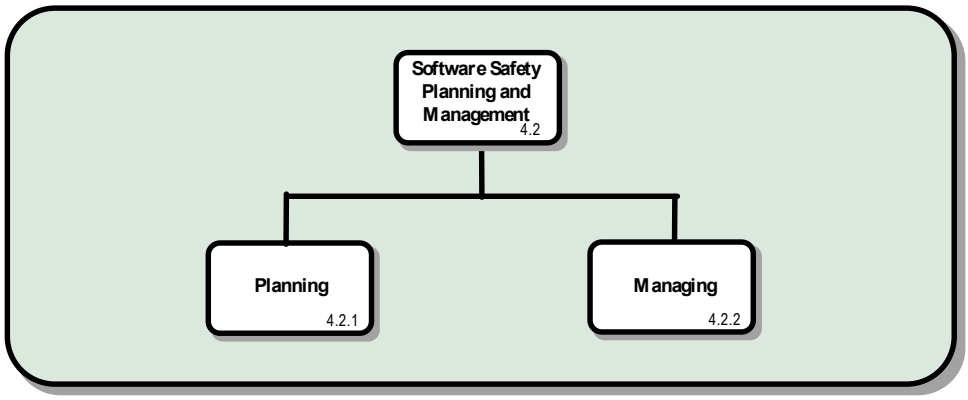


Figure 4-4: Software Safety Planning

This section is applicable to all members of the SSS Team. It assumes a minimal understanding and experience with safety engineering programs. The topics include the following:

a. Identification of managerial and technical program interfaces required by the SSS Team.

b. Definition of user and developer contractual relationships to ensure that the SSS Team implements the tasks, and produces the products, required to meet contractual requirements.

c. Identification of programmatic and technical meetings and reviews normally supported by the SSS Team.

d. Identification and allocation of critical resources to establish a SSS Team and conduct a software safety program.

e. Definition of planning requirements for the execution of an effective program.

The planning for an effective SSP and software safety program requires extensive forethought from both the supplier and the customer. Although they both envision a perfect SSP, there are subtle differences associated with the identification, preparation, and execution of a successful safety program from these two perspectives. The contents of Figures 4-5 and 4-6 represent the primary differences between agencies that both must understand before considering the software safety planning and coordinating activities.

4.2.1 Planning

Comprehensive planning for the software safety program requires an initial assessment of the degree of software involvement in the system design and the associated hazards. Unfortunately,

this is difficult since little is usually known about the system other than operational requirements during the early planning stages. Therefore, the safety SOW must encompass all possible designs. This generally results in a fairly generic SOW that will require later tailoring of a SSS program to the system design and implementation.

Figure 4-5 represents the basic inputs, outputs, tasks, and critical interfaces associated with the planning requirements associated with the PA. Frustrations experienced by safety engineers executing the system safety tasks can usually be traced back to the lack of adequate planning by the customer. The direct result is normally an under-budget, under-staffed safety effort that does not focus on the most critical aspects of the system under development. This usually can be traced back to the customer not assuring that the Request for Proposal (RFP), SOW, and the contract contain the correct language, terminology, and/or tasks to implement a safety program and the required or necessary resources. Therefore, the ultimate success of any safety program strongly depends on the planning function by the customer.

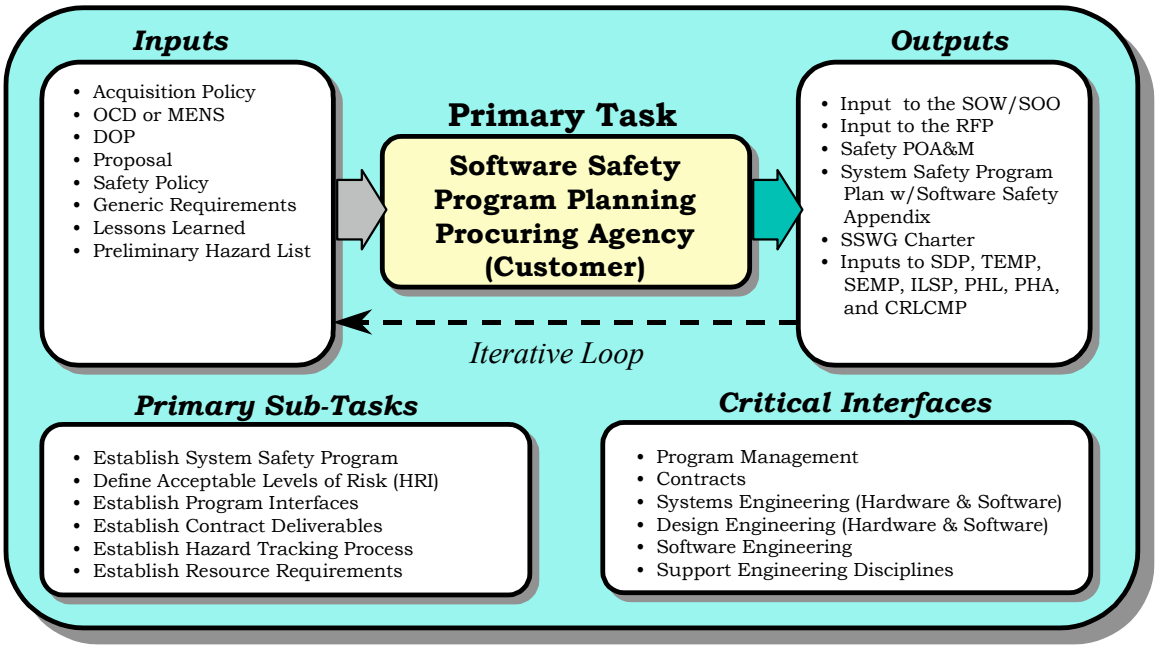


Figure 4-5: Software Safety Planning by the Procuring Authority

For the PA, software safety program planning begins as soon as the need for the system is identified. The PA must identify points of contact within the organization and define the interfaces between various engineering disciplines, administrative support organizations, program management, contracting group, and Integrated Product Teams (IPT) within the PA to develop the necessary requirements and specifications documents. In the context of acquisition reform, invoking military standards and specifications for DOD procurements is not permitted or is significantly reduced. Therefore, the PA must incorporate the necessary language into any contractual documents to ensure that the system under development will meet the safety goals and objectives.

PA safety program planning continues through contract award and may require periodic updating during initial system development and as the development proceeds through various phases.

However, management of the overall SSP continues through system delivery and acceptance and throughout the system's life cycle. After deployment, the PA must continue to track system hazards and risks and monitor the system in the field for safety concerns identified by the user. The PA must also make provisions for safety program planning and management for any upgrades, product improvements, maintenance, technology refreshment, and other follow-on efforts to the system.

The major milestones affecting the PA's safety and software safety program planning include release of contract requests for proposals or quotes, proposal evaluation, major program milestones, system acceptance testing and evaluation, production contract award, initial operational capability (release to the users), and system upgrades or product improvements.

Although the Developing Agency's (DA) software safety program planning begins after receipt of a contract RFP, or quotes, the DA can significantly enhance his/her ability to establish an effective program through prior planning (see Figure 4-6). Prior planning includes establishing effective systems engineering and software engineering processes that fully integrate system and software systems safety. Corporate engineering standards and practices documents that incorporate the tenets of system safety provide a strong baseline from which to build a successful SSP even though the contract may not contain specific language regarding the safety effort.

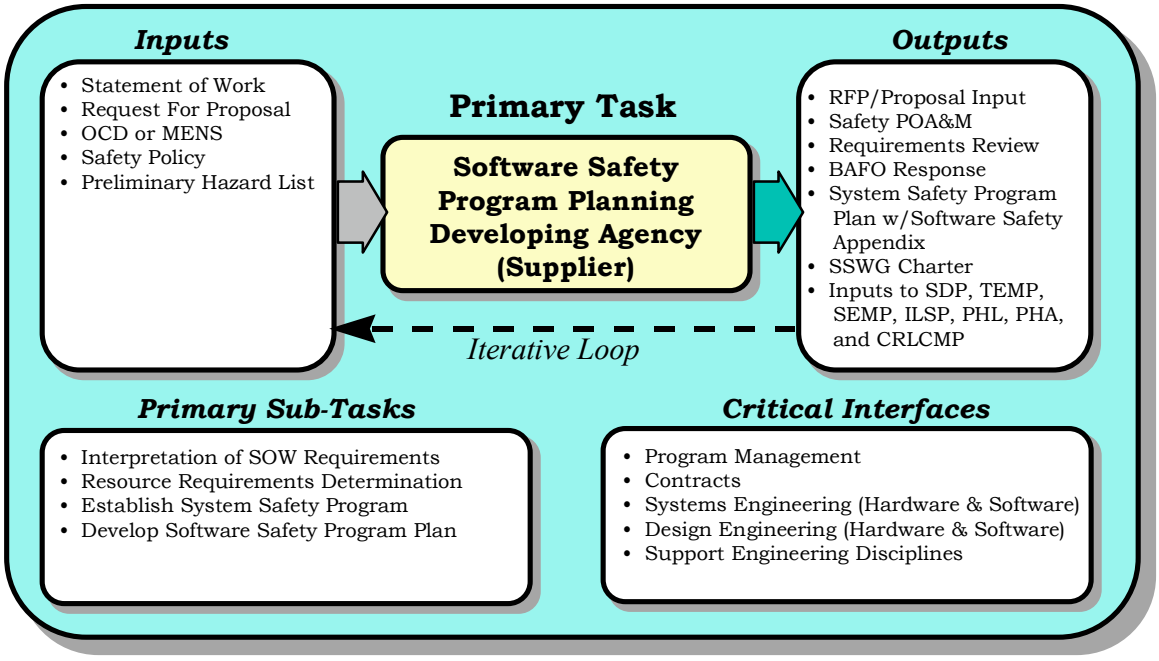


Figure 4-6: Software Safety Planning by the Developing Agency

Acquisition reform recommends that the Government take a more interactive approach to system development without interfering with that development. The interactive aspect is to participate as a member of the DA's IPTs as an advisor without hindering development. This requires a careful balance on the part of the government participants. From the system safety and SSS perspective, that includes active participation in the appropriate IPTs by providing the government perspective on recommendations and decisions made in those forums. This also

requires the government representative to alert the developer to hazards known to the government but not to the developer.

Acquisition reform also requires the DA to warrant the system thus making the DA liable for any mishaps that occur, even after system acceptance by the PA. Although the courts have yet to fully test that liability, the DA can significantly reduce his/her liability through this interactive process. Having government representatives present when making safety-related decisions provides an inherent "buy-in" by the Government to the residual risks in the system. This has the effect of significantly reducing the DA's liability.[24] MIL-STD 882D also implies this reduction in liability.

Where is this discussion leading? Often, contract language is non-specific and does not provide detailed requirements, especially with respect to safety requirements for the system. Therefore, it is the DA's responsibility to define a comprehensive SSP that will ensure that the delivered system provides an acceptably low level of safety risk to the customer, not only for the customer's benefit, but for the DA's benefit as well. At the same time, the DA must remain competitive and reduce safety program costs to the lowest practical level consistent with ensuring the delivery of a system with the lowest risk practical. Although the preceding discussion focused on the interaction between the Government and the DA, the same tenets apply to any contractual relationship, especially between prime and subcontractors.

The DA software safety planning continues after contract award and requires periodic updates as the system proceeds through various phases of development. These updates should be in concert with the PA's software safety plans. However, management of the overall system and SSS programs continues from contract award through system delivery and acceptance and may extend throughout the system life cycle, depending on the type of contract. If the contract is a Total System Responsibility contract or requires the DA perform routine maintenance, technology refreshments, or system upgrade, the software safety program management and engineering must continue throughout the system's life cycle. Thus, the DA must make provisions for safety program planning and management for these phases and other follow-on efforts on the system.

The major milestones affecting the DA's safety and software safety program planning include the receipt of contract requests for proposals or quotes, contract award, major program milestones, system acceptance testing and evaluation, production contract award, release to the customer, system upgrades, and product improvements.

While the software safety planning objectives of the PA and DA may be similar, the planning and coordination required to meet these objectives may come from different angles (in terms of specific tasks and their implementation), but they must be in concert (Figure 4-7). Regardless, both agencies must work together to meet the safety objectives of the program. In terms of planning, this includes the following:

- Establishment of a SSP
- Definition of acceptable levels of safety risk

[24] This represents a consensus opinion of lawyers and legal experts in the practice of defending government contractors in liability cases.

- Definition of critical program, management, and engineering interfaces
- Definition of contract deliverables
- Development of a Software Hazard Criticality Matrix (SHCM) (see Paragraph 4.2.1.5)

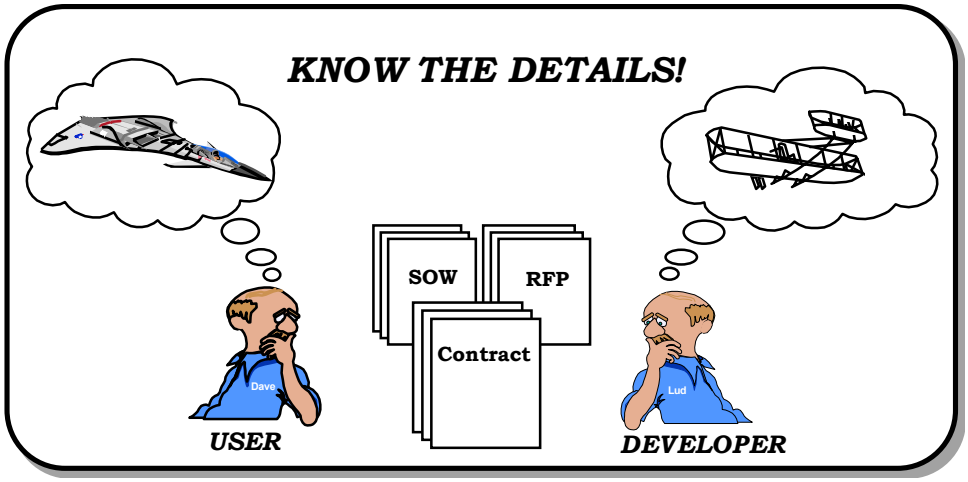


Figure 4-7: Planning the Safety Criteria Is Important

4.2.1.1 Establish the System Safety Program

The PA must establish the safety program as early as practical in the development of the system. The PM should identify a Principal for Safety (PFS – Navy term) or other safety manager early in the program to serve as the single point of contact for all safety-related matters on the system. This individual will interface with safety review authorities, the DA safety team, PA and DA program management, the safety engineering team, and other groups as required to ensure that the safety program is effective and efficient. The PFS may also establish and chair a Software Systems Safety Working Group (SwSWG) or SSS Team. For large system developments where software is likely to be a major portion of the development, a safety engineer for software may also be identified who reports directly to the overall system PFS. The size of the safety organization will depend on the complexity of the system under development, and the inherent safety risks. Another factor influencing the size of the PM's safety team is the degree of interaction with the customer and supplier and the other engineering and program disciplines. If the development approach is a team effort with a high degree of interaction between the organizations, the safety organization may require additional personnel to provide adequate support.

The PA should prepare a System Safety Management Plan (SSMP) describing the overall safety effort within the PA organization and the interface between the PA safety organization and the DA's system safety organization. The SSMP is similar to the SSPP in that it describes the roles and responsibilities of the program office individuals with respect to the overall safety effort. The PFS or safety manager should coordinate the SSMP with the DA's SSPP to ensure that the tasks and responsibilities are complete and will provide the desired risk assessment. The SSMP differs from the SSPP in that it does not describe the details of the safety program, such as

analysis tasks, contained in the SSPP. A special note with regard to programs initiated under MIL-STD-882D. MIL-STD-882D does not require or contain a Contract Deliverable Requirements List (CDRL) listing for a SSPP. However, Section 4.1 requires that the PM and the developer document the "agreed upon" system safety process. This is virtually identical to the role of the SSPP. Therefore, the PFS or safety manager coordinates the SSMP with this documented safety process.

The PA must specify the software safety program for programs where software performs or influences safety-critical functions of the system. The PA must establish the team in accordance with contractual requirements, managerial and technical interfaces and agreements, and the results of all planning activities discussed in previous sections of this Handbook. Proper and detailed planning will increase the probability of program success. The tasks and activities associated with the establishment of the SSP are applicable to both the supplier and the customer. Unfortunately, the degree of influence of the software on safety-critical functions in the system is often not known until the design progresses to the point of functional allocation of requirements at the system level.

The PM must predicate the software safety program on the goals and objectives of the system safety and the software development disciplines of the proposed program. The safety program must focus on the identification and tracking (from design, code, and test) of both initial SSRs and guidelines, and those requirements derived from system-specific, functional hazards analyses. Common deficiencies in software safety programs are usually the lack of a team approach in addressing both the initial and the functional SSRs of a system. The software development community has a tendency to focus on only the initial SSRs while the system safety community may focus primarily on the functional SSRs derived through hazard analyses. A sound SSS program traces both sets of requirements through test and requirements verification activities. The ability to identify (in total) all applicable SSRs is essential for *any* given program and must be adequately addressed.

4.2.1.2 Defining Acceptable Levels of Risk

One of the key elements in safety program planning is the identification of the acceptable level of risk for the system. This process requires both the identification of a HRI and a statement of the goal of the safety program for the system. The former establishes a standardized means with which to group hazards by risk (e.g., unacceptable, undesirable, etc.) while the latter provides a statement of the expected safety quality of the system. The ability to categorize specific hazards into the HRI matrix is based upon the ability of the safety engineer to assess hazard severity and likelihood of occurrence. The PA, in concert with the user, must develop a definition of the acceptable risk and provide that to the DA. The PA must also provide the developer with guidance on risk acceptance authorities and reporting requirements. DOD 5000.2R requires that high-risk hazards (Unacceptable hazard per MIL-STD-882) obtain component CAE signature for acceptance. Serious risk hazards (Undesirable) require acceptance at the PEO level. The DA must provide the PM supporting documentation for the risk acceptance authority.

4.2.1.3 Program Interfaces

System safety engineering is responsible for the coordination, initiation, and implementation of the software safety engineering program. While this responsibility cannot be delegated to any other engineering discipline within the development team, software safety must assign specific tasks to the engineers with the appropriate expertise. Historically, system safety engineering performs the engineering necessary to identify, assess, and eliminate or reduce the safety risk of hazards associated with complex systems. Now, as software becomes a major aspect of the system, software safety engineering must establish and perform the required tasks and establish the technical interfaces required to fulfill the goals and objectives of the system safety (and software safety) program. However, the SSS Team cannot accomplish this independently without the inter-communication and support from other managerial and technical functions. Within the DOD acquisition and product development agencies, IPTs have been established to ensure the success of the design, manufacture, fabrication, test, and deployment of weapon systems. These IPTs formally establish the accountability and responsibility between functions and among team members. This accountability and responsibility is both from the top down (management-to-engineer) and from the bottom up (engineer-to-management).

The establishment of a credible SSS activity within an organization requires this same rigor in the identification of team members, the definition of program interfaces, and the establishment of lines of communication. Establishing formal and defined interfaces allows program and engineering managers to assign required expertise for the performance of the identified tasks of the software safety engineering process. Figure 4-8 shows the common interfaces necessary to adequately support a SwSSP. It includes management interfaces, technical interfaces, and contractual interfaces.

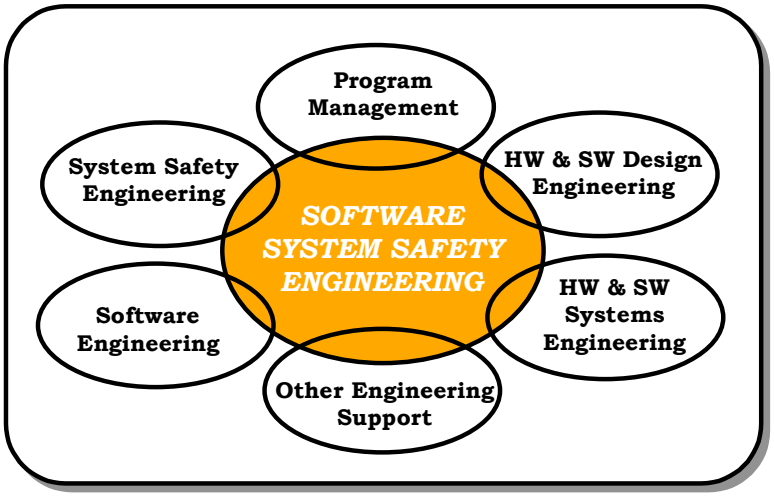


Figure 4-8: Software Safety Program Interfaces

4.2.1.3.1 Management Interfaces

The PM, under the authority of the AE or the PEO:

- Coordinates the activities of each professional discipline for the entire program,

- Allocates program resources,
- Approves the programs' planning documents, including the SSPP, and
- Reviews safety analyses; accepts impact on system for Critical and higher category hazards (based upon acceptable levels of risk); and submits finding to PEO for acceptance of unmitigated, unacceptable hazards.

It is the PM's responsibility to ensure that processes are in place within a program that meet, not only the programmatic, technical, and safety objectives, but also the functional and system specifications and requirements of the customer. The PM must allocate critical resources within the program to reduce the sociopolitical, managerial, financial, technical, and safety risk of the product. Therefore, management support is essential to the success of the SSS program.

The PM ensures that the safety team develops a practical process and implements the necessary tasks required to:

- Identify system hazards,
- Categorize hazards in terms of severity and likelihood,
- Perform causal factor analysis,
- Derive hardware and software design requirements to eliminate and/or control the hazards,
- Provide evidence for the implementation of hardware and software safety design requirements,
- Analyze and assess the residual safety risk of any hazards that remain in the design at the time of system deployment and operation, and
- Report the residual safety risk and hazards associated with the fielded system to the appropriate acceptance authority.

The safety manager and the software engineering manager depend on program management for the allocation of necessary resources (time, tools, training, money, and personnel) for the successful completion of the required SSS engineering tasks.

Within the DOD framework, the AE (Figure 4-9) is ultimately responsible for the acceptance of the residual safety risk at the time of test, initial systems operation, and deployment. The AE must certify at the Test Readiness Review (TRR), and the Safety Program Review Authority (SPRA) [sometimes referred to as a Safety Review Board (SRB)], that all hazards and failure modes have been eliminated or the risk mitigated or controlled to a level As-Low-As-Reasonably-Possible (ALARP). At this critical time, an accurate assessment on the residual safety risk of a system facilitates informed management and engineering decisions. Under the old acquisition process, without the safety risk assessment provided by a credible system safety process, the AE assumed the personal, professional, programmatic, and political liabilities in the decision making process. If the PM failed to implement effective system and SSS programs,

he/she may assume the liability due to failure to follow DOD directives. The developer now assumes much of that liability under acquisition reform. The ability of the PFS or safety manager to provide an accurate assessment of safety risk depends on the support provided by program management throughout the design and development of the system. Under acquisition reform, the government purchases systems as if they are off-the-shelf products. The developer warrants the system for performance and safety characteristics thus making the developer liable for any mishaps that occur. However, the AE is ultimately responsible for the safety of the system and the assessment and acceptance of the residual risk. The developer's safety team, in coordination with the PA's safety team must provide the AE with the accurate assessment of the residual risk such that he/she can make informed decisions. Again, this is also implied by MIL-STD 882D.

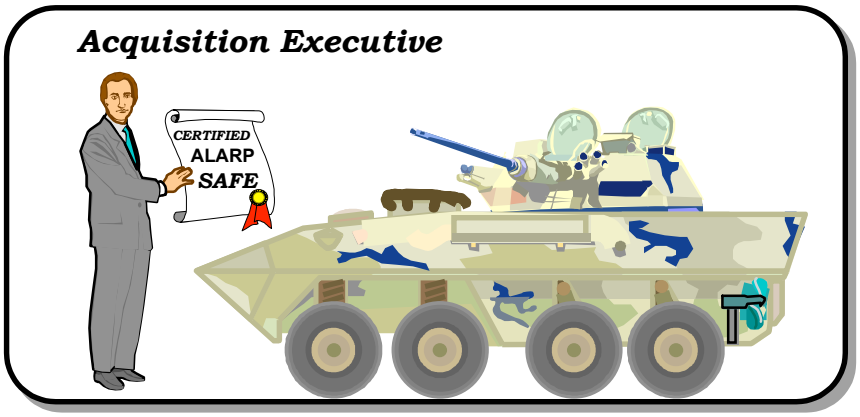


Figure 4-9: Ultimate Safety Responsibility

4.2.1.3.2 Technical Interfaces

The engineering disciplines associated with system development must also provide technical support to the SSS Team (Figure 4-10). Engineering management, design engineers, systems engineers, software development engineers, Integrated Logistics Support (ILS), and other domain engineers supply this essential engineering support. Other domain engineers include reliability, human factors, quality assurance (QA), test and evaluation, verification and validation, maintainability, survivability, and supportability. Each member of the engineering team must provide timely support to the defined processes of the SSS Team to accomplish the safety analyses and for specific design influence activities which eliminate, reduce, or control hazard risk. This includes the traceability of SSRs from design-to-test (and test results) with its associated and documented evidence of implementation.

A sure way for the software safety activity to fail is to not secure software engineering acceptance and support of the software safety process, functions, and implementation tasks. One must recognize that most formal education and training for software engineers and developers does not present, teach, or rationalize system safety. The system safety process relating to the derivation of functional SSR through hazard analyses is foreign to most software developers. In fact, the concept that software can be a causal factor to a hazard is foreign to many software engineers.

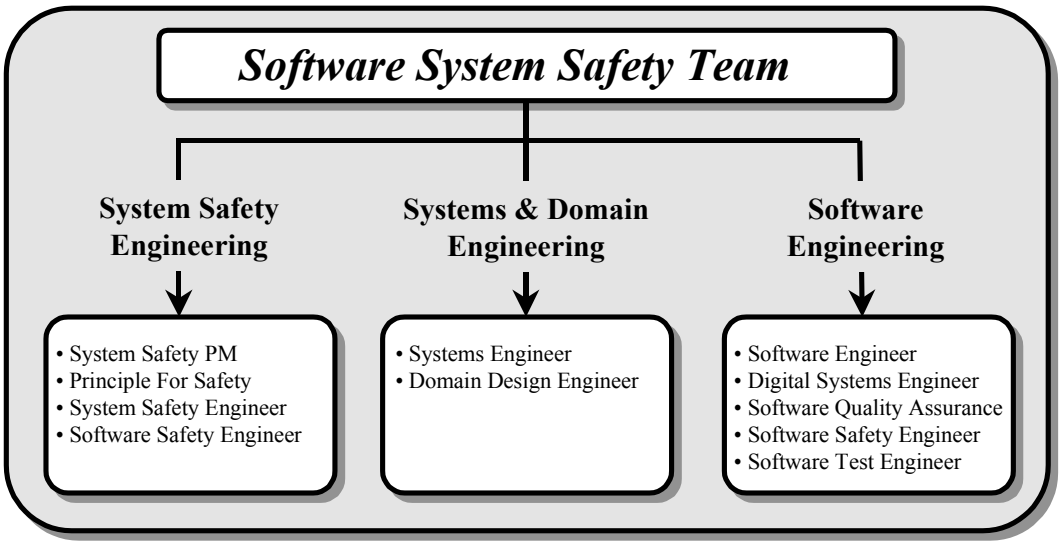


Figure 4-10: Proposed SSS Team Membership

Without the historical experience of cultivating technical interfaces between software development and system safety engineering, several issues may need resolution. They include:

Software engineers may feel threatened that system safety has the responsibility for activities considered part of the software engineering realm

Software developers are confident enough in their own methods of error detection, error correction, and error removal, that they ignore the system safety inputs to the design process. This is normally in support of initial SSRs

There is insufficient communication and resource allocation between software development and system safety engineering to identify, analyze, categorize, prioritize, and implement both generic and derived SSRs

A successful SSS effort requires the establishment of a technical SSS Team approach. The SSP Manager, in concert with the systems engineer and software engineering team leaders must define the individual tasks and specific team expertise required and assigns responsibility and accountability for the accomplishment of these tasks. The SwSPP must include the identification and definition of the required expertise and tasks in the software safety portion or appendix. The team must identify both the generic SSRs and guidelines and the functional safety design requirements derived from system hazards and failure modes that have specific software input or influence. Once these hazards and failure modes are identified, the team can identify specific safety design requirements through an integrated effort. All SSRs must be traceable to test and be correct, complete, and testable where possible. The Requirements Traceability Matrix (RTM) within the SRCA documents this traceability. The implemented requirements must eliminate, control, or reduce the safety risk as low as reasonably possible while meeting the user requirements within operational constraints. Appendix C.3 contains supplemental information pertaining to the technical interfaces.

4.2.1.3.3 Contractual Interfaces

Management planning for the SSS function includes the identification of contractual interfaces and obligations. Each program has the potential to present unique challenges to the system safety and software development managers. These may include a RFP that does not specifically address the safety of the system, to contract deliverables that are extremely costly to develop. Regardless of the challenges, the tasks needed to accomplish a SSS program must be planned to meet both the system and user specifications and requirements and the safety goals of the program. The following are those contractual obligations that are deemed to be most essential for any given contract:

- RFP
- SOW
- Contract
- CDRL

Example templates of a RFP and SOW/SOO are contained in Appendix G.

4.2.1.4 Contract Deliverables

The SOW defines the deliverable documents and products (e.g., CDRLs) desired by the customer. Each CDRL deliverable should be addressed in the SSPP to include the necessary activities and process steps required for its production. Completion of contract deliverables is normally tied to the acquisition life cycle of the system being produced and the program milestones identified in the Systems Engineering Management Plan (SEMP). The planning required by the system safety manager ensure that the system safety and software safety processes provide the necessary data and output for the successful accomplishment of the plans and analysis. The system safety schedule should track closely to the SEMP and be proactive and responsive to both the customer and the design team. Contract deliverables should be addressed individually on the safety master schedule and within the SSPP whether these documents are contractual deliverables or internal documents required to support the development effort.

As future procurements under acquisition reform will generally not have specific military and DOD standards and few if any deliverables, the PA must ensure that sufficient deliverables are identified and contractually required to meet programmatic and technical objectives.

This activity must also specify the content and format of each deliverable item. As existing government standards transition to commercial standards and guidance, the safety manager must ensure that sufficient planning is accomplished to specify the breadth, depth, and timeline of each deliverable [which is normally defined by Data Item Descriptions (DID)]. The breadth and depth of the deliverable items must provide the necessary audit trail to ensure that safety levels of risk is achieved (and are visible) during development, test, support transition, and maintenance in the out-years. The deliverables must also provide the necessary evidence or audit trail for validation and verification of SSRs. The primary method of maintaining a sufficient audit trail is the utilization of a developer's safety data library (SDL). This library would be the repository for all

safety documentation. Appendix C, Section C.1 describes the contractual deliverables that should be contained in the SDL.

4.2.1.5 Develop Software Hazard Criticality Matrix

Criteria described in MIL-STD-882 provides the basis for the HRI (described in Paragraph 3.6.1.4). This example may be used for guidance, or an alternate HRI may be proposed. The given HRI methodology used by a program must possess the capability to graphically delineate the boundaries between acceptable, allowable, undesirable (i.e., serious), and unacceptable (i.e., high) risk. Figure 4-11 provides a graphical representation of a risk acceptance matrix. In this example, the hazard record database contains 10 hazards, which currently remain in the unacceptable categories (categories IA, IB, IC, IIA, IIB, and IIIA), of safety risk. This example explicitly states that the hazards represented in the unacceptable range must be resolved.

FREQUENCY OF OCCURRENCE	HAZARD CATEGORIES			
	I CATASTROPHIC	II CRITICAL	III MARGINAL	IV NEGLIGIBLE
A - FREQUENT	0	0	0	0
B - PROBABLE	4	1	0	0
C - OCCASIONAL	5	16	0	0
D - REMOTE	24	25	3	0
E - IMPROBABLE	1	1	1	0

Legend:	
IA, IB, IC, IIA, IIB, IIIA	**UNACCEPTABLE**, condition must be resolved. Design action is required to eliminate or control hazard.
ID, IIC, IID, IIIB, IIIC	**UNDESIRABLE**, Program Manager decision is required. Hazard must be controlled or hazard probability reduced.
IE, IIE, IIID, IIIE, IVA, IVB	**ALLOWABLE**, with Program Manager review. Hazard control desirable if cost effective.
IVC, IVD, IVE	**ACCEPTABLE** without review. Normally not cost effective to control. Hazard is either negligible or can be assumed will not occur.

Figure 4-11: Example of Risk Acceptance Matrix

The ability to categorize specific hazards into the above matrix is based upon the ability of the safety engineer to assess their severity and likelihood of occurrence. Historically, the traditional HRI matrix did not include the influence of the software on the hazard occurrence. The rationale for this is twofold: When the HRI matrix was developed, software was generally not used in safety-critical roles. Second, applying failure probabilities to software is impractical. The traditional HRI uses the hazard severity and probability of occurrence to assign the HRI with probabilities defined in terms of mean time between failure, probability of failure per operation, or probability of failure during the life cycle, depending on the nature of the system. This relies heavily on the ability to obtain component reliability information from engineering sources. However, applying probabilities of this nature to software, except in purely qualitative terms, is impractical. Therefore, software requires an alternative methodology. Software does not fail in the same manner as hardware. It does not wear out, break, or have increasing tolerances that

result in failures. Software errors are generally errors in the requirements (failure to anticipate a set of conditions that lead to a hazard, or influence of an external component failure on the software) or implementation errors (coding errors, incorrect interpretation of design requirements). If the conditions occur that cause the software to not perform as expected, a failure occurs. Therefore, reliability predictions become a prediction of when the specific conditions will occur that cause it to fail. Without the ability to accurately predict a software error occurrence, alternate methods of hazard categorization must be available when the hazard possesses software causal factors.

During the early phases of the safety program, the prioritization and categorization of hazards is essential for the allocation of resources to the functional area possessing the highest risk potential. This section of the Handbook presents a method of categorizing hazards having software causal factors strictly for purposes of allocation of resources to the SSS program. This methodology does not provide an assessment of the residual risk associated with the software at the completion of development. However, the execution of the safety program, the development and analysis of SSRs and the verification of their implementation in the final software provide the basis for a qualitative assessment of the residual risk in traditional terms.

4.2.1.5.1 Hazard Severity

Regardless of the hazard causal factors (hardware, software, human error, or environment) the severity of the hazard remains constant. The consequence of a hazard's occurrence remains the same regardless of what actually caused the hazard unless the design of the system somehow changes the possible consequence. As the hazard severity is the same, the severity table presented in Paragraph 3.6.1.2 (Table 3-1, Hazard Severity), remains an applicable criteria for the determination of hazard criticality for those hazards having software causal factors.

4.2.1.5.2 Hazard Probability

The difficulty of assigning useful probabilities to faults or errors in software requires a supplemental method of determining hazard risk where software causal factors exist. Figure 4-12 demonstrates that in order to determine a hazard probability, the analyst must assess the software causal factors in conjunction with the causal factors from hardware, human error, and other factors. The determination of hardware and human error causal factor probabilities remains constant (although there is significant disagreement regarding assigning probabilities to human error) in terms of historical "best" practices. Regardless, the risk assessment process must address the contribution of the software to the hazard's cumulative risk.

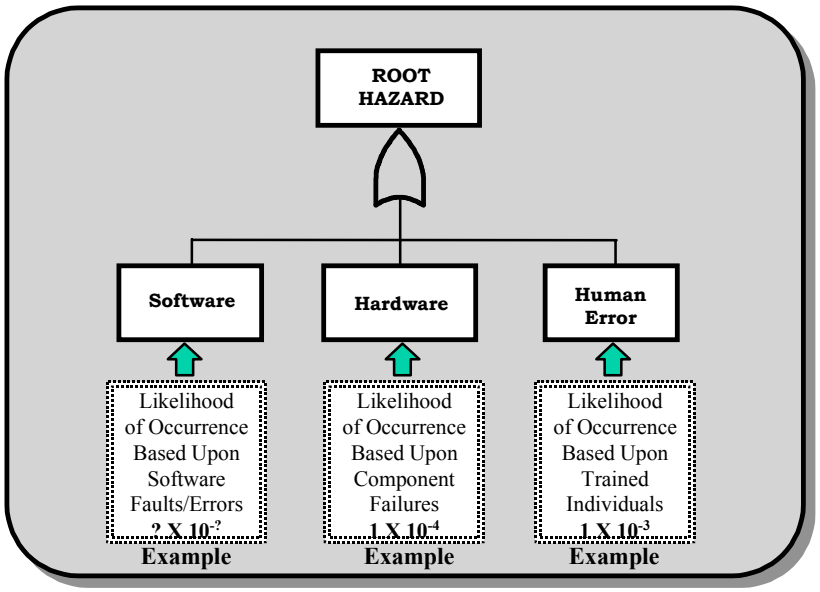


Figure 4-12: Likelihood of Occurrence Example

There have been numerous methods of determining the software's influence on system-level hazards. Two of the most popular are presented in MIL-STD-882C and RTCA DO-178B (see Figure 4-13). These do not specifically determine software-caused hazard probabilities, but instead assess the software's "control capability" within the context of the software causal factors. In doing so, each software causal factor can be labeled with a software control category for the purpose of helping to determine the degree of autonomy that the software has on the hazardous event. The SSS Team must review these lists and tailor them to meet the objectives of the SSP and software development program.

MIL-STD-882C	***RTCA-DO-178B***
(I) Software exercises autonomous control over potentially hazardous hardware systems, subsystems or components without the possibility of intervention to preclude the occurrence of a hazard. Failure of the software or a failure to prevent an event leads directly to a hazards occurrence.	**(A)** Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft.
(IIa) Software exercises control over potentially hazardous hardware systems, subsystems, or components allowing time for intervention by independent safety systems to mitigate the hazard. However, these systems by themselves are not considered adequate.	**(B)** Software whose anomalous behavior, as shown by the System Safety assessment process, would cause or contribute to a failure of system function resulting in a hazardous/severe-major failure condition of the aircraft.
(IIb) Software item displays information requiring immediate operator action to mitigate a hazard. Software failure will allow or fail to prevent the hazard's occurrence.	**(C)** Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a major failure condition for the the aircraft.
(IIIa) Software items issues commands over potentially hazardous hardware systems, subsystem, or components requiring human action to complete the control function. There are several, redundant, independent safety measures for each hazardous event.	**(D)** Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft.
(IIIb) Software generates information of a safety critical nature used to make safety critical decisions. There are several, redundant, independent safety measures for each hazardous event.	**(E)** Software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of function with no effect on aircraft operational capability or pilot workload. Once software has been confirmed as level E by the certification authority, no further guidelines of this document apply.
(IV) Software does not control safety critical hardware systems, subsystems, or components and does not provide safety critical information.	

Figure 4-13: Examples of Software Control Capabilities

Once again, the concept of labeling software causal factors with control capabilities is foreign to most software developers and programmers. They must be convinced that this activity has utility in the identification and prioritization of software entities that possess safety implications. In most instances, the software development community desires the list to be as simplistic and short as possible. The most important aspect of the activity must not be lost; that is, the ability to categorize software causal factors in determining the hazard likelihood and the design, code, and test activities required to mitigate the potential software cause. Autonomous software with functional links to catastrophic hazards demands more coverage than software that influences low severity hazards.

4.2.1.5.3 Software Hazard Criticality Matrix

The SHCM, shown in Figure 4-14, assists PMs, SSS Team, and the subsystem and system designers in allocating resources to the software safety effort.

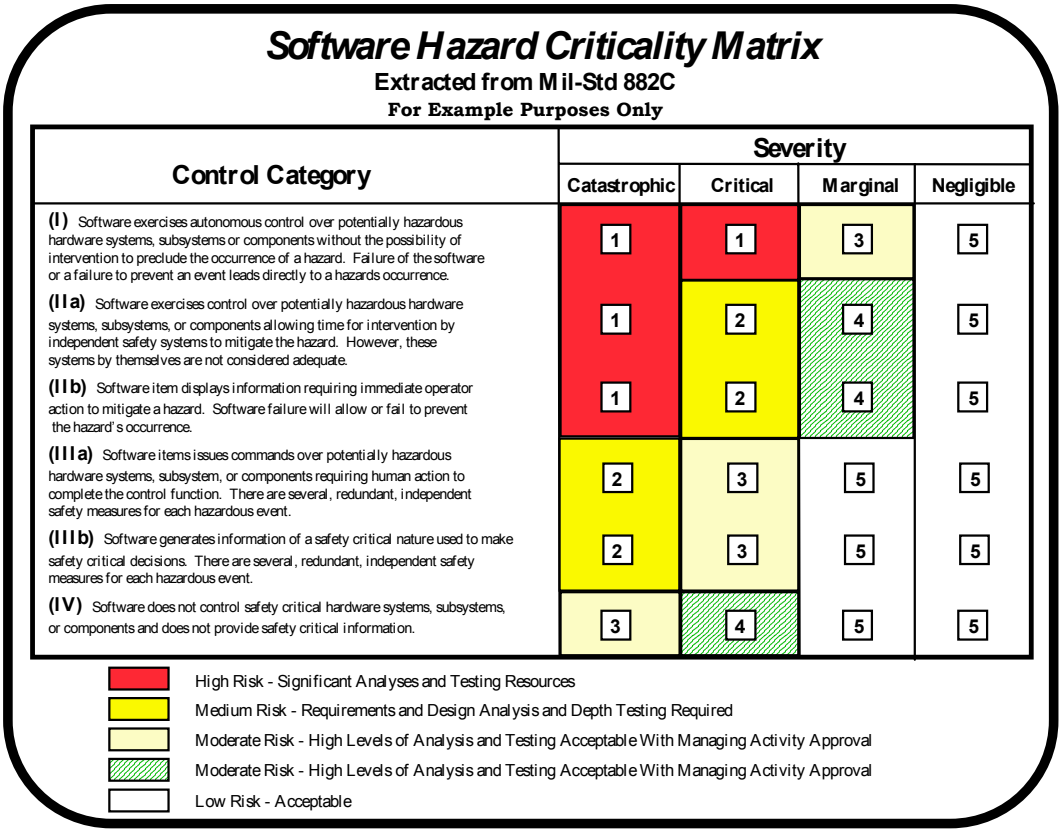


Figure 4-14: Software Hazard Criticality Matrix, MIL-STD-882C

It is not an HRI matrix for software. The higher the Software Hazard Risk Index (SHRI) number, the fewer resources required to ensure that the software will execute safely in the system context. The software control measure of the SHCM also assists in the prioritization of software design and programming tasks. However, the SHRI's greatest value may be during the functional allocation phase. Using the SHRI, software safety can influence the design to:

- Reduce the autonomy of the software control of safety-critical aspects of the system,
- Minimize the number of safety-critical functions in the software, and
- Use the software to reduce the risk of other hazards in the system design.

If conceptual design (architecture) shows a high degree of autonomy over safety-critical functions, the software safety effort requires significantly more resources. Therefore, the systems engineering team can consider this factor in the early design phases. By reducing the number of software modules containing safety-critical functions, the developer reduces the portion of the software requiring safety assessment and thus the resources required for that assessment. The systems engineering team must balance these issues with the required and desired capabilities of the system. Too often, developers rush to use software to control functionality when non-software alternatives will provide the same capabilities. While the safety risk associated with the non-software alternatives must still be assessed, the process is likely to be less costly and resource intensive.

4.2.2 Management

SSS program management (Figure 4-15), like SSP management, begins as soon as the SSP is established, and continues throughout the system development. Management of the effort requires a variety of tasks or processes, from establishing the SwSWG to preparing the SAR. Even after a system is placed in service, management of the SSS effort continues to address modifications and enhancements to the software and the system. Often, changes in the use or application of a system necessitate a re-assessment of the safety of the software in the new application.

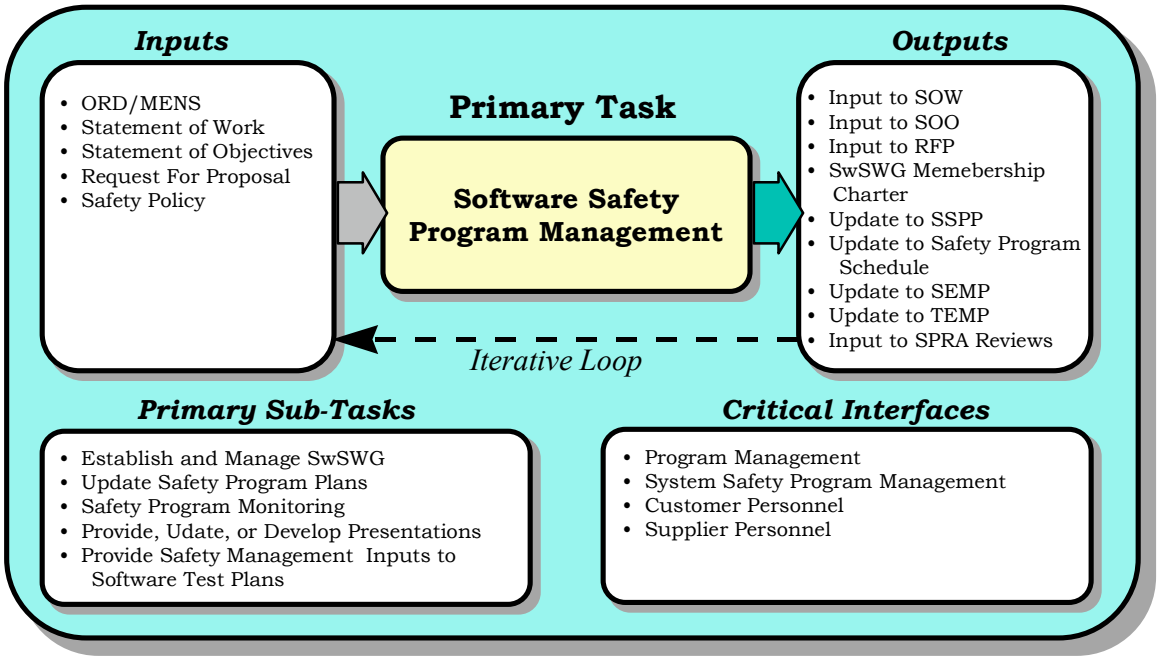


Figure 4-15: Software Safety Program Management

Effective management of the safety program is essential to the effective and efficient reduction of system risk. This section discusses the managerial aspects of the software safety tasks and provides guidance in establishing and managing an effective software safety program. Initiation of the SSP is all that is required to begin the activities pertaining to software safety tasks. Initial management efforts parallel portions of the planning process since many of the required efforts (such as establishing a hazard tracking system or researching lessons learned) need to begin very early in the safety program. Safety management pertaining to software generally ends with the completion of the program and its associated testing; whether it is a single phase of the development process (e.g., concept exploration) or continues through the development, production, deployment, and maintenance phases. In the context of acquisition reform, this means that management of the efforts must continue throughout the system life cycle. From a practical standpoint, management efforts end when the last safety deliverable is completed and is accepted by the customer. Management efforts then may revert to a "caretaker" status in which the PFS or safety manager monitors the use of the system in the field and identifies potential safety deficiencies based on user reports and accident/incident reports. Even if the developer has no responsibility for the system after deployment, the safety program manager can develop a valuable database of lessons learned for future systems by identifying these safety deficiencies.

Establishing a software safety program includes establishing a Software Safety Working Group (SwSWG). This is normally a sub-group of the SSWG and chaired by the PFS or safety manager. The SwSWG has overall responsibility for the following:

- Monitoring and control of the software safety program,
- Identifying and resolving hazards with software causal factors,
- Interfacing with the other IPTs, and
- Performing final safety assessment of the system design.

A detailed discussion of a SwSWG is found in the supplemental information of Appendix C, paragraph C.5.2.

It is in this phase of the program that the Software Safety Plan of Action and Milestones (POA&M) is developed based on the overall software development program POA&M in coordination with the system safety POA&M. Milestones from the software development POA&M, particularly design reviews and transition points (e.g., from unit code and test to integration) determine the milestones required of the software safety program. The SwSWG must ensure that the necessary analyses are complete in time to provide the necessary input to various development efforts to ensure effective integration of software safety into the overall software development process. The overall Phases, Milestones and Processes Chart, discussed in Paragraph 4.3 below, identifies the major program milestones from MIL-STD-498 and -499 with the associated software safety program events.

One of the most difficult aspects of software safety program management is the identification and allocation of resources required to adequately assess the safety of the software. In the early planning phases, the configuration of the system and the degree of interaction of the software with the potential hazards in the system are largely unknown. The higher the degree of software

involvement, the greater the resources required to perform the assessment. To a large extent, the software safety program manager can use the early analyses of the design, participation in the functional allocation, and high-level software design process to ensure that the amount of safety-critical software is minimized. If safety-critical functions are distributed throughout the system and its related software, then the software safety program must encompass a much larger portion of the software. However, if the safety-critical functions are associated with as few software modules as practical, the level of effort may be significantly reduced.

Effective planning and integration of the software safety efforts into the other IPTs will significantly reduce the software safety-related tasks that must be performed by the SSS Team. Incorporating the generic SSRs into the plans developed by the other IPTs allows them to assume responsibility for their assessment, performance, and/or evaluation. For example, if the SSS Team provides the quality assurance generic SSRs to the Software Quality Assurance (SQA) IPT, they will perform compliance assessments with requirements, not just for safety, but for all aspects of the software engineering process. In addition, if the SQA IPT "buys-into" the software safety program and its processes, it significantly supplements the efforts of the software safety engineering team, reduces their workload, and avoids duplication of effort. The same is true of the other IPTs such as CM and Software Test and Evaluation. In identifying and allocating resources to the software safety program, the software safety program manager can perform advance planning, establish necessary interfaces with the other IPTs, and identify individuals to act as software safety representatives on those IPTs.

Identifying the number of analyses and the level of detail required to adequately assess the software involves a number of processes. Experience with prior programs of a similar nature is the most valuable resource that the software safety program manager has for this task. However, every program development is different and involves different teams of people, PA requirements, and design implementations. The process begins with the identification of the system-level hazards in the PHL. This provides an initial idea of the concerns that must be assessed in the overall safety program. From the system specification review process, the functional allocation of requirements results in a high-level distribution of safety-critical functions and system-level safety requirements to the design architecture. The safety-critical functions and requirements are thus known in general terms. Software functions that have a high safety-criticality (e.g., warhead arming and firing) will require a significant analysis effort that may include code-level analysis. Safety's early involvement in the design process can reduce the amount of software that requires analysis; however, the software safety manager must still identify and allocate resources to perform these tasks. Those safety requirements that conflict with others (e.g., reliability) require trade-off studies to achieve a balance between desirable attributes.

The software control categories discussed in Paragraph 4.2.1.5 provide a useful tool for identifying software that requires high levels of analysis and testing. Obviously, the more critical the software, the higher the level of effort necessary to analyze, test, and assess the risk associated with the software. In the planning activities, the SwSWG identifies the analyses necessary to assess the safety of specific modules of code. The best teacher for determining the level of effort required is experience. These essential analyses do not need to be performed by the software engineering group and may be assigned to another group or person with the specialized expertise necessary. The SwSWG will have to provide the necessary safety-related

guidance and training to the individuals performing the analysis, but only to the extent necessary for them to accomplish the task.

One of the most important aspects of software safety program management is monitoring the activities of the safety program throughout system development to ensure that tasks are on schedule and within cost, and to identify potential problem areas that could impact the safety or software development activities. The software safety manager must:

- Monitor the status and progress of the software and system development effort to ensure that program schedule changes are reflected in the software safety program POA&M.
- Monitor the progress of the various IPTs and ensure that the safety interface to each is working effectively. When problems are detected, either through feedback from the software safety representative or other sources, the software safety manager must take the necessary action to mitigate the problem.
- Monitor and receive updates regarding the status of analyses, open Hazard Action Report (HAR), and other safety activities on a weekly basis. Significant HARs should be discussed at each SwSWG meeting and the status updated as required. A key factor that the software safety program manager must keep in mind is the tendency for many software development efforts to begin compressing the test schedule as slippage occurs in the software development schedule. He or she must ensure that the safety test program is not compromised as a result of the slippage.

SPRA requirements vary with the PA and are often governed by PA directives. The contract will generally identify the review requirements; however, it is the responsibility of the DA to ensure that the software safety program incorporates the appropriate reviews into the SwSPP. The system safety manager must identify the appropriate SPRA and review the schedule during the development process. SPRA reviews generally involve significant effort outside of the other software safety tasks. The DA must determine the level of effort required for each review and the support that will be required during the review, and incorporate these into the SwSPP. For complex systems, multiple reviews are generally required to update the SPRA and ensure that all of the PA requirements are achieved.

Although SPRA requirements may vary from each PA, some require a technical data package and briefing to a review board. The technical data package may be a SAR or may be considerably more complex. The DA must determine whether they are to provide the technical data package and briefing, or whether that activity is to be performed independently. In either event, safety program personnel may be required to participate or attend the reviews to answer specific technical questions that may arise. Normally, the presenters require several weeks of preparation for the SPRA reviews. Preparation of the technical data package and supporting documentation requires time and resources even though the data package is a draft or final version of the SAR.

4.3 Software Safety Task Implementation

This section of the Handbook describes the primary task implementation steps required for a baseline SSS engineering program. It presents the necessary tasks required for the integration of software safety activities into the functional areas of system and software development. Remember, software systems safety (or software safety) is a **subset** of both the system safety engineering process and the software engineering and development process.

As the Handbook introduces the software safety engineering process, it will identify the inputs to the described tasks and the products that the specific process step produces. Each program and engineering interface tied to software safety engineering must agree with the processes, tasks, and products of the software safety program and must agree with the timing and scope of effort to verify that it is in concert with the objectives and requirements of each interface. If other program disciplines are not in agreement, or do not see the functional utility of the effort, they will usually default to a "non-support" mode.

Figure 4-16 provides a graphical depiction of the software safety activities required for the implementation of a credible SSS program. Remember that the process steps identified in this Handbook represent a baseline program that has a historical lessons learned base and includes the best practices from successful programs. As each procurement, software acquisition, or development has the potential and probability to be uniquely diverse, the safety manager *must* use this section as a guide only. Each of the following steps should be analyzed and assessed to identify where minor changes are required or warranted for the software development program proposed. If these tasks, with the implementation of minor changes, are incorporated in the system acquisition life cycle, the SSS effort has a very high likelihood of success.

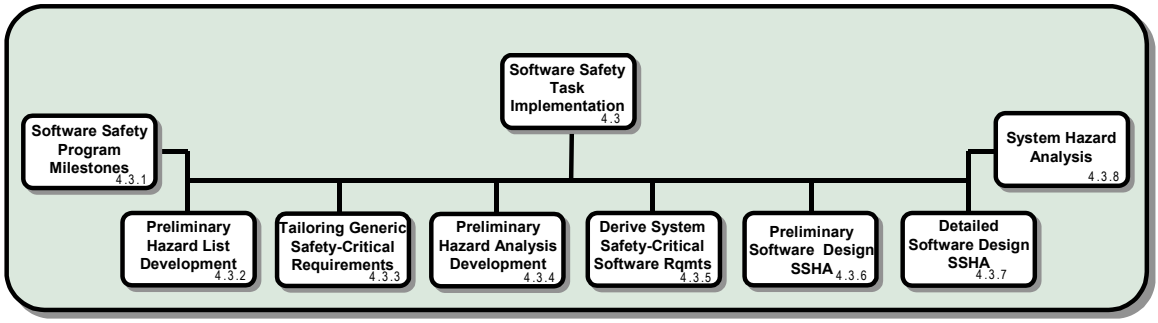


Figure 4-16: Software Safety Task Implementation

The credibility of software safety engineering activities within the hardware and software development project depends on the credibility of the individual(s) performing the managerial and technical safety tasks. It also depends on the identification of a logical, practical, and cost effective process that produces the safety products to meet the safety objectives of the program. The primary safety products include hazard analyses, initial safety design requirements, functionally derived safety design requirements (based on hazard causes), test requirements to produce evidence for the elimination and/or control of the safety hazards, and the identification of safety requirements pertaining to operations and support of the product. The managerial and technical interfaces must agree that the software safety tasks defined in this section will provide

the documented evidence for the resolution of identified hazards and failure modes in design, manufacture (code in software), fabrication, test, deployment, and support activities. It must also thoroughly define and communicate residual safety risk to program management at any point in time during each phase of the development life cycle.

4.3.1 Software Safety Program Milestones

The planning and management of a successful software safety program is supplemented by the safety engineering and management program schedule. The schedule should include near-term and long-term events, milestones, and contractual deliverables. The schedule should also reflect the system safety management and engineering tasks that are required for each life cycle phase of the program and that are required to support DOD milestone decisions. Specific safety data to support special safety boards or safety studies for compliance and certification purposes is also crucial. Examples include FAA certification, US Navy Weapon System Explosives Safety Review Board approval, Defense Nuclear Agency Nuclear Certification, and the U.S. Air Force Non-Nuclear Munitions Safety Board approval. The PM must track each event, deliverable, and/or milestone to ensure that safety analysis activities are timely in the development process to help facilitate cost-effective and technically feasible design solutions. These activities ensure that the SSS program will meet the desired safety specifications of program and system development activities.

Planning for the SSP must include the allocation of resources to support the travel of safety management and engineers. The contractual obligations of the SOW, in concert with the processes stated in the program plans and the required support of program meetings, dictate the scope of safety involvement. With the limited funds and resources of today's programs, the system safety manager must determine and prioritize the level of support allocated to program meetings and reviews. Planning for the budgeted travel allocations for the safety function must assess the number of meetings requiring support, the number of safety personnel required to attend, and the physical location of the meetings. Likewise, budgets must include adequate funds for support tools, such as database programs for hazard tracking and analysis tools. The resource allocation activity becomes complicated if priorities are not established "up-front" with the determination of meetings to support, tools required, and other programmatic activities. Once priorities are established, safety management can alert program management to meetings that cannot be supported due to budget constraints for the purpose of concurrence or the reallocation of resources. Figure 4-17 provides an example milestone schedule for a software safety program. It graphically depicts the relationship of safety-specific activities to the acquisition life cycles of both system and software development.

Remember that each procurement is unique and will have subtle differences associated with managerial and technical interfaces, timelines, processes and milestones. This schedule is an example with specific activities and time relationship-based "typical" programs. Program planning must integrate program-specific differences into the schedule and support the practical assumptions and limitations of the program.

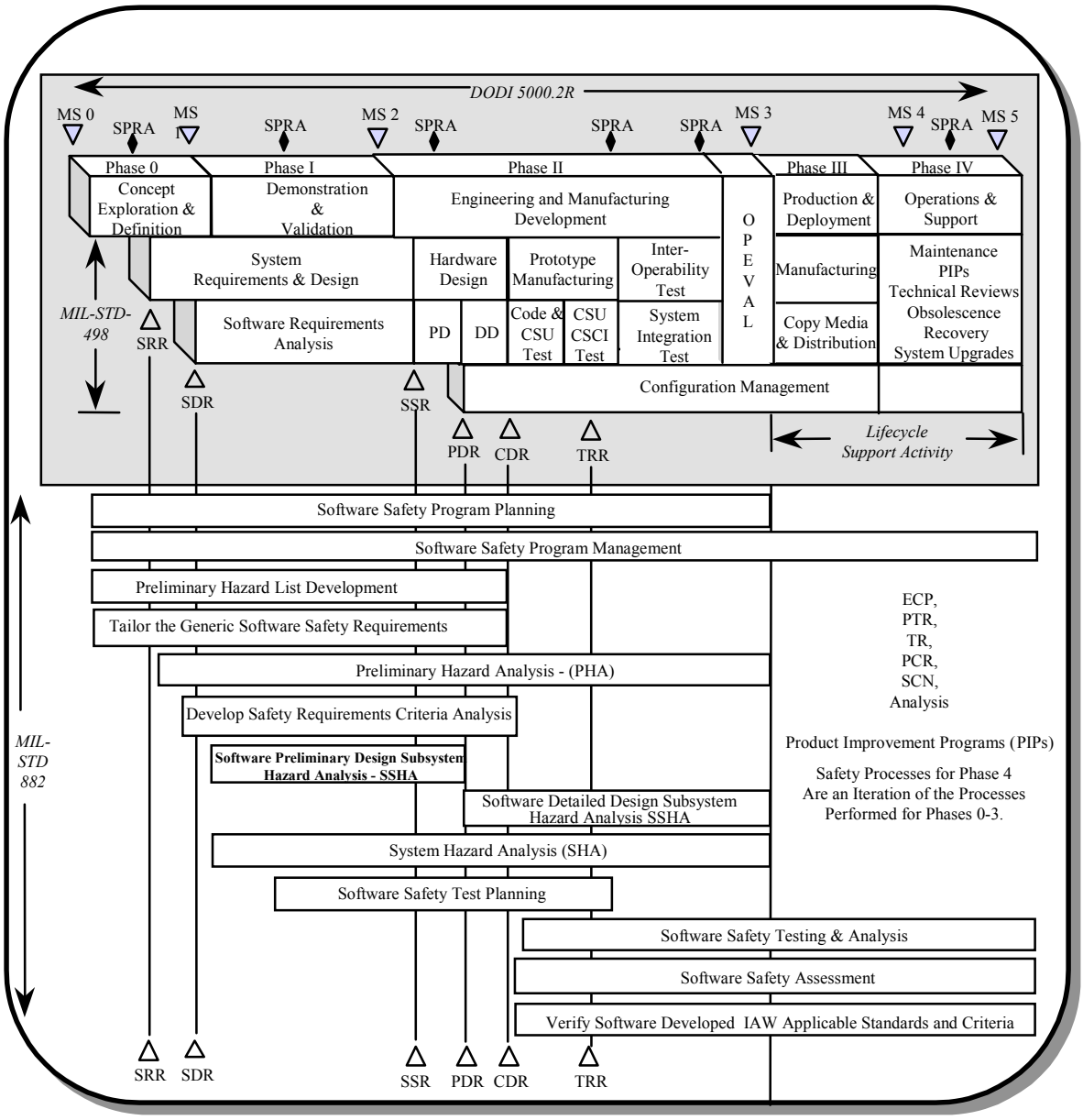


Figure 4-17: Example POA&M Schedule

As described in Paragraph 4.2.2, the POA&M will also include the various safety reviews, PA reviews, internal reviews, and the SwSWG meetings. The software safety assessment milestones are generally geared to the SPRA reviews, since the technical data package required is in fact either the draft or final software-related SAR. Hazard analysis schedules must reflect program milestones where hazard analysis input is required. For example, SSRs resulting from generic requirements tailoring (documented in the SRCA) must be available as early as practical in the design process for integration into design, programmatic, and system safety documents. Specific safety requirements from the PHA and an initial set of safety design requirements must be available prior to the PDR for integration into the design documents. System safety and software safety must participate in the system specification review and provide recommendations during

the functional allocation of system requirements to hardware, software, operation, and maintenance. After functional allocation is complete, the Software Engineering IPT, with the help of the software safety representative, will develop the SRS. At this point, SSS should have the preliminary software safety assessment complete with hazards identified and initial software-related HRIs. The SwSWG updates the analyses as the system development progresses however, the safety design requirements (hardware, software, and human interfaces) must be complete prior to the CDR. Requirements added after the CDR can have a major impact on program schedule and cost.

During the development of the SRS, the SSS Team initiates the SSHA and its evaluation of the preliminary software design. This preliminary design analysis assesses the system and software architecture, and provides design recommendations to reduce the associated risk. This analysis provides the basis for input to the design of the Computer Software Configuration Items (CSCIs), and the individual software modules. At this point the software safety engineer (SwSE) must establish a SAF for each CSCI or Computer Software Unit (CSU), depending on the complexity of the design to document the analysis results generated. As the design progresses and detailed specifications are available, the SSS Team initiates a SSHA that assesses the detailed software design. The team analyzes the design of each module containing safety-critical functions and the software architecture in the context of hazard failure pathways and documents the results in the SAF. For highly safety-critical software, the analysis will extend to the source code to ensure that the intent of the SSRs is properly implemented.

The development of safety test requirements begins with the identification of SSRs. SSRs can be either safety design requirements, generic or functional (derived) requirements, or requirements generated from the implementation of hazard controls that will be discussed in Paragraph 4.3.5. SSRs incorporated into software documentation automatically becomes a part of the software test program. However, throughout the development, the software safety organization must ensure that the test plans and procedures will provide the desired validation of SSRs demonstrating that they meet the intent of the requirement. Section 4.4 provides additional guidance on the development of the safety test program. Detailed inputs regarding specific safety tests are derived from the hazard analyses, causal factor analysis, and the definition of software hazard mitigation requirements. Safety-specific test requirements are provided to the test organization for development of specific test procedures to validate the SSRs. The analysis associated with this phase begins as soon as test data from the safety tests is available.

The SHA begins as soon as functional allocation of requirements occurs and continues through the completion of system design. Specific milestones for the SHA include providing safety test requirements for integration testing to the test organization and detailed test requirements for interface testing. The latter will be required before testing of the software with other system components begins.

4.3.1 Preliminary Hazard List Development

The PHL is a contractual deliverable on many programs and is described in Appendix C, paragraph C.1.3. This list is the initial set of hazards associated with the system under development. Development of the PHL requires knowledge of the physical and functional requirements of the system and some foreknowledge of the conceptual system design. The

documentation of the PHL helps to initiate the analyses that must be performed on the system, subsystems, and their interfaces. The PHL is based upon the review of analyses of similar systems, lessons learned, potential kinetic energies associated with the design, design handbooks, and user and systems specifications. The generated list also aids in the development of initial (or preliminary) requirements for the system designers and the identification of programmatic (technical or managerial) risks to the program. Figure 4-18 illustrates the PHL development process.

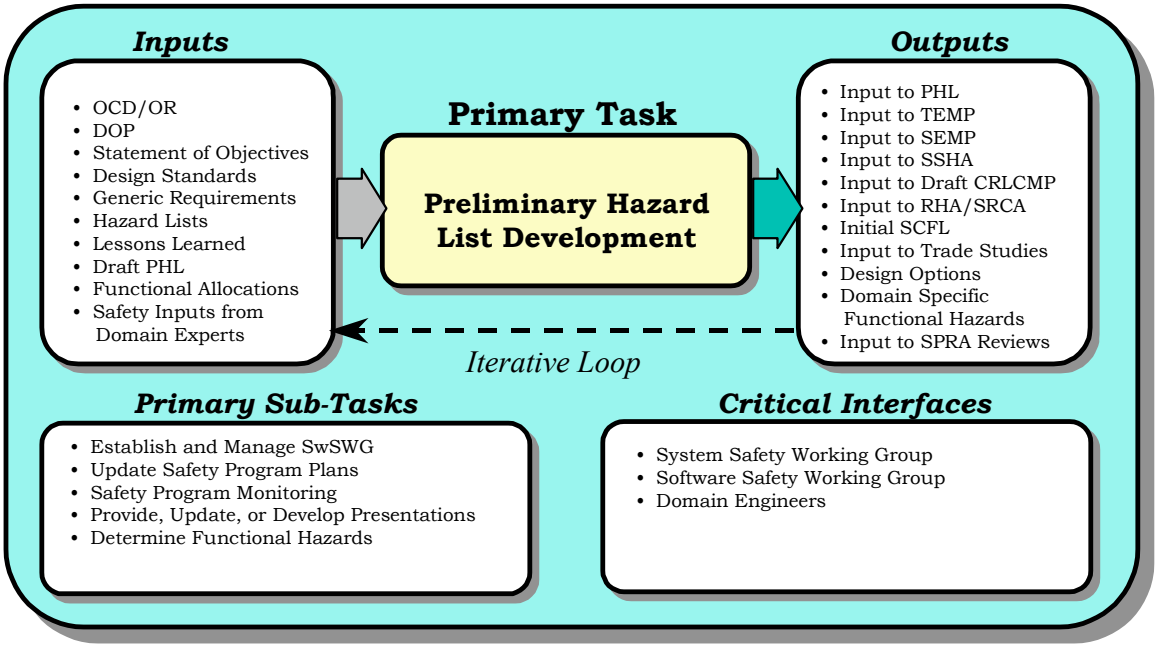


Figure 4-18: Preliminary Hazard List Development

The development of the PHL is an integrated engineering task that requires cooperation and communication between functional disciplines and among systems, safety, and design engineers. The assessment and analysis of all preliminary and current data pertaining to the proposed system accomplish this task. From a documentation perspective, the following should be available for review:

- Preliminary system specification
- Preliminary product specification
- User requirements document
- Lessons learned
- Analysis of similar systems
- Prior safety analyses (if available)
- Design criteria and standards

From the preceding list of documentation and functional specifications, system safety develops a preliminary list of system hazards for further analysis. Although the identified hazards may appear to be general or immature at this time, this is normal for the early phase of system development. As the hazards are analyzed against system physical and functional requirements, they will mature to become the hazards fully documented in the PHA, SSHA, SHA, and the Operating and Support Hazard Analysis (O&SHA). A preliminary risk assessment of the PHL hazards will help determine whether trade studies or design options must be considered to reduce the potential for unacceptable or unnecessary safety risk in the design.

In addition to the information assessed from preliminary documents and databases, technical discussions with systems engineering to help determine the ultimate functions associated with the system that are safety-critical. Functions that should be assessed include manufacturing, fabrication, operations, maintenance, and test. Other technical considerations include transportation and handling, software/hardware interfaces, software/human interfaces, hardware/human interfaces, environmental health and safety, explosive and other energetic components, product loss prevention, as well as nuclear safety considerations.

This effort begins with the safety engineer analyzing the functionality of each segment of the conceptual design. From the gross list of system functions, the analyst must determine the safety ramifications of loss of function, interrupted function, incomplete function, function occurring out of sequence, or function occurring inadvertently. This activity also provides for the initial identification of safety-critical functions. The rationale for the identification of safety-critical functions (list) of the system is addressed in the identification of safety deliverables (Appendix C, Paragraph C.1.4). It should be reiterated at this point, that this is an activity that must be performed as a part of the defined software safety process. This process step ensures that the project manager, systems and design engineers, in addition to the software developers and engineers are aware of each function of the design considered safety-critical or to have a safety impact. It also ensures that each individual module of code that performs these functions is officially labeled as “safety-critical” and that defined levels of design and code analysis and test activity are mandated. An example of the possible safety-critical functions of a tactical aircraft is provided in Figure 4-19.

There are two benefits to identifying the safety-critical functions of a system. First, the identification assists the SSS Team in the categorization and prioritization of safety requirements for the software architecture early in the design life cycle. If the software performs or influences the safety-critical function(s), that module of code becomes safety-critical. This eliminates emotional discussions on whether individual modules of code are designed and tested to specific and extensive criteria. Second, it reduces the level of activity and resource allocations to software code not identified as safety-critical. This benefit is cost avoidance.

At this phase of the program, specific ties from the PHL to the software design are quite premature. Specific ties to the software are normally through hazard causal factors, which have yet to be defined at this point in the development. However, there may be identified hazards which have preliminary ties to safety-critical functions which in turn are functionally linked to the preliminary software design architecture. If this is the case, this functional link should be adequately documented in the safety analysis for further development and analysis. At the same time, there are likely to be specific “generic” SSRs applicable to the system (see Appendix E).

These requirements are available from multiple sources and must be specifically tailored to the program as they apply to the system design architecture.

Figure 4-19: An Example of Safety-Critical Functions

4.3.2 Tailoring Generic Safety-Critical Requirements

Figure 4-20 depicts the software engineering process for tailoring the generic safety-related software requirement list. Generic SSRs are those design features, design constraints, development processes, "best practices," coding standards and techniques, and other general requirements that are levied on a system containing safety-critical software, regardless of the functionality of the application. The requirements themselves are not safety specific (i.e., not tied to a specific system hazard). In fact, they may just as easily be identified as reliability requirements, good coding practices, and the like. They are, however, based on lessons learned from previous systems where failures or errors occurred that either resulted in a mishap or a potential mishap. The PHL, as described above, may help determine the disposition or applicability of many individual generic requirements. The software safety analysis must identify the applicable generic SSRs necessary to support the development of the SRS as well as programmatic documents (e.g., SDP). A tailored list of these requirements should be provided to the software developer for inclusion into the SRS and other documents.

Several individuals, agencies, and/or institutions have published lists of generic safety requirements for consideration. To date, the most thorough is included in Appendix E, Generic Requirements and Guidelines, which includes the STANAG 4404, NATO Standardization Agreement, *Safety Design Requirements and Guidelines for Munitions Related Safety-Critical Computing Systems*, the Mitre (Ada) list, and other language-specific requirements. These

requirements should be assessed and prioritized according to the applicability to the development effort. Whatever list is used, the analyst must assess each item individually for compliance, non-compliance, or non-applicability. On a particular program, the agreed upon generic SSRs should be included in the SRCA and appropriate high-level system specifications.

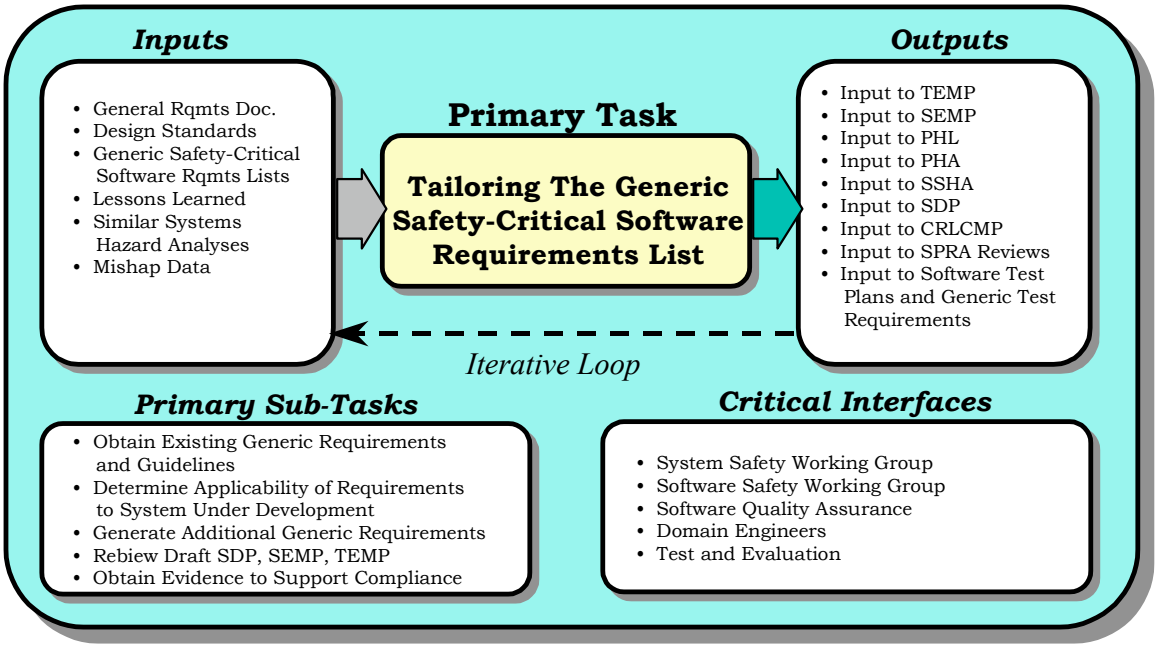


Figure 4-20: Tailoring the Generic Safety Requirements

Figure 4-21 is an example of a worksheet form that may be used to track generic SSR implementation. Whether the program is complying with the requirement, the physical location of the requirement and the physical location of the evidence of implementation must be cited in the EVIDENCE block. If the program is not complying with the requirement (e.g., too late in the development to impose a safety kernel) or the requirement is not-applicable (e.g., an Ada requirement when developing in assembly language), a statement of explanation must be included in the RATIONALE block. An alternative mitigation of the source risk that the requirement addresses should be described if applicable, possibly pointing to another generic requirement on the list.

A caution regarding the "blanket" approach of establishing the entire list of guidelines or requirements for a program: Each requirement will cost the program critical resources; people to assess and implement; budget for the design, code, and testing activities; and program schedule. Unnecessary requirements will impact these factors and result in a more costly product with little or no benefit. Thus, these requirements should be assessed and prioritized according to the applicability to the development effort. Inappropriate requirements, which have not been adequately assessed, are unacceptable. The analyst must assess each requirement individually and introduce only those that may apply to the development program.

Some requirements only necessitate a sampling of evidence to provide implementation (e.g., no conditional GO-TO statements). The lead software developer will often be the appropriate

individual to gather the implementation evidence of the generic SSRs from those who can provide the evidence. The lead software developer may assign SQA, CM, V&V, human factors, software designers, or systems designers to fill out individual worksheets. The entire tailored list of completed forms should be approved by the SSE and submitted to the SDL and referred to by the SAR. This provides evidence of generic SSR implementation

GENERIC SOFTWARE SAFETY REQUIREMENTS IMPLEMENTATION	INTENDED COMPLIANCE		
	YES	NO	N/A
Item: Coding Requirements Issues Has an analysis (scaling, frequency response, time response, discontinuity, initialization, etc.) of the macros been performed?		X	
Rationale: (If NO or N/A, describe the rationale for the decision and resulting risk.) There are no macros in the design (discussed at checklist review 1/11/96)			
Evidence: (If YES, describe the kind of evidence that will be provided. Note: Specify sampling percentage per SwSPP, if applicable.)			
Action: (State the Functional area with responsibility.) Software Development POC:			

Figure 4-21: Example of a Generic Software Safety Requirements Tracking Worksheet

4.3.3 Preliminary Hazard Analysis Development

The PHA is a safety engineering and software safety engineering analysis performed to identify and prioritize hazards and their casual factors in the system under development. Figure 4-22 depicts the safety engineering process for the PHA. There is nothing unique about the software aspects other than the identification of the software causal factors. Many safety engineering texts provide guidance for developing the PHA. This Handbook will not describe the processes for brevity. Many techniques provide an effective means of identifying system hazards and the determination of their causal factors.

A note of caution is that each methodology focuses on a process that will identify a substantial portion of the hazards, however, none of the methodologies are complete. For example, the Energy-Barrier trace analysis is an effective process, however, it may lead the analyst to neglect certain energy control functions. In addition, in applying this technique, the analyst must consider not only the obvious forms of energy (chemical, electrical, mechanical, etc.) but also such energy forms as biological. Many analysts use the life cycle profile of a system as the basis

for the hazard identification and analysis. Unless the analyst is particularly astute, he/she may miss subtle system hazards and, more importantly, causal factors. Appendix B provides a list of references that includes many texts describing the PHA.

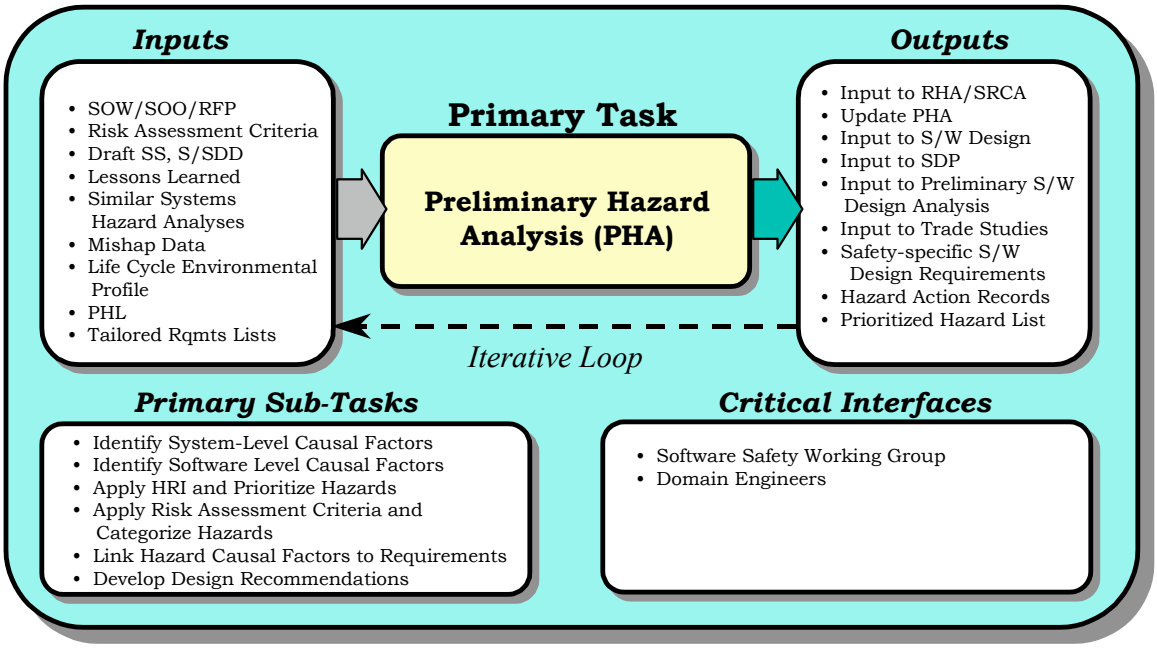


Figure 4-22: Preliminary Hazard Analysis

The PHA becomes the springboard documentation to launch the SSHA and SHA analyses as the design matures and progresses through the development life cycle. Preliminary hazards can be eliminated (or officially closed through the SSWG) if they are deemed to be inappropriate for the design. Remember that this analysis is preliminary and is used to provide early design considerations that may or may not be derived or matured into design requirements.

Throughout this analysis, the PHA provides input to trade-off studies. Trade-off analyses performed in the acquisition process are listed in Table 4-1 (DSMC, 1990). These analyses offer alternative considerations for performance, producibility, testability, survivability, compatibility, supportability, reliability, and system safety during each phase of the development life cycle. System safety inputs to trade studies include the identification of potential or real safety concerns, and the recommendations of credible alternatives that may meet all (or most) of the requirements while reducing overall safety risk.

The entire unabridged list of potential hazards developed in the PHL is the entry point of PHA. The list should be "scrubbed" for applicability and reasonability as the system design progresses. The first step is to eliminate from the PHL any hazards not applicable to the system [e.g., if the system uses a titanium penetrator vice a Depleted Uranium (DU) penetrator, eliminate the DU related hazards]. The next step is to categorize and prioritize the remaining hazards according to the (System) HRI. The categorization provides an initial assessment of system hazard severity and probability of occurrence and, thus, the risk. The probability assessment at this point is usually subjective and qualitative. After developing the prioritized list of preliminary hazards,

the analysis proceeds with determining the hardware, software, and human interface causal factors to the individual hazard as shown in Figure 4-23.

Table 4-1: Acquisition Process Trade-off Analyses

Acquisition Phase	Trade-Off Analysis Function
Mission Area Analysis	Prioritize Identified User Needs
Concept Exploration	Compare New Technologies With Proven Concepts Select Concepts Best Meeting Mission Needs Select Alternative System Configuration
Demonstration Validation	Select Technology Reduce Alternative Configurations to a Testable Number
Full Scale Development	Select Component/Part Designs Select Test Methods Select Operational Test & Evaluation Quantities
Production	Examine Effectiveness of all Proposed Design Changes Perform Make-Or-Buy, Process, Rate, and Location Decisions

After the prioritized list of preliminary hazards is determined, the analysis proceeds with determining the hardware, software, and human interface causal factors to the individual hazard as shown in Figure 4-23.

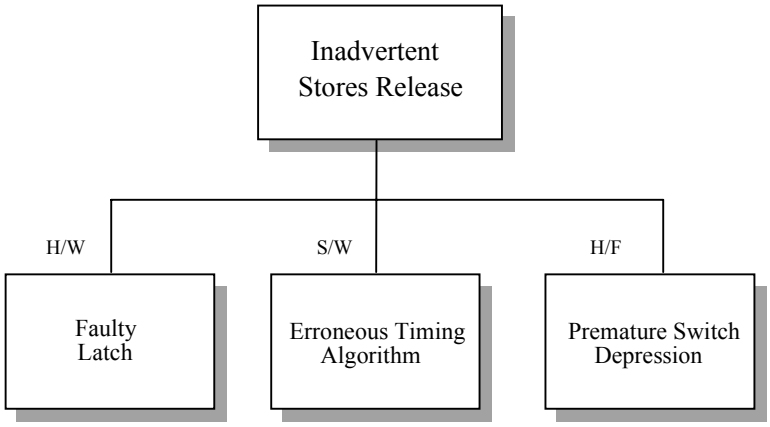


Figure 4-23: Hazard Analysis Segment

This differentiation of causes assists in the separation and derivation of specific design requirements for implementation in software. For example, as the analysis progresses, the analyst may determine that software or hardware could subsequently contribute to a hardware casual factor. A hardware component failure may cause the software to react in an undesired manner leading to a hardware-influenced software causal factor. The analyst must consider all paths to ensure coverage of the software safety analysis.

Although this tree diagram can represent the entire system, software safety is particularly concerned with the software causal factors linked to individual hazards in addition to ensuring that the mitigation of each causal factor is traced from requirements to design and code, and

subsequently tested. These preliminary analyses and subsequent system and software safety analyses identify when software is a potential cause, or contributor to a hazard, or will be used to support the control of a hazard.

At this point, tradeoffs evolve. It should become apparent at this time whether hardware, software, or human training best mitigates the first-level causal factors of the PHL item (the root event that is undesirable). This causal factor analysis provides insight into the best functional allocation within software design architecture. It should be noted that requirements designed to mitigate the hazard causal factors do not have to be one-to-one, i.e., one software causal factor does not yield one software control requirement. *Safety requirements can be one-to-one, one-to-many, or many-to-one in terms of controlling hazard causal factors to acceptable levels of safety risk.* In many instances, designers can use software to compensate for hardware design deficiencies or where hardware alternatives are impractical. As software is considered to be cheaper to change than hardware, software design requirements may be designed to control specific hardware causal factors. In other instances, one design requirement (hardware or software) may eliminate or control numerous hazard causal factors (e.g., some generic requirements). *This is extremely important to understand as it illuminates the importance of not accomplishing hardware safety analysis and software safety analysis separately.* A system-level, or subsystem-level hazard can be caused by a single causal factor or a combination of many causal factors. The safety analyst must consider all aspects of what causes the hazard and what will be required to eliminate or control the hazard. Hardware, software, and human factors can usually not be segregated from the hazard and cannot be analyzed separately. The analysis performed at this level is integrated into the trade-off studies to allow programmatic and technical risks associated with various system architectures to be determined.

Both software-initiated causes and human error causes influenced by software input must be adequately communicated to the digital systems engineers and software engineers to identify software design requirements that preclude the initiation of the root hazard identified in the analysis. The software development team may have already been introduced to the applicable generic SSRs. These requirements must address how the system will react safely to operator errors, component failures, functional software faults, hardware/software interface failures, and data transfer errors. As detailed design progresses, however, functionally derived software requirements will be defined and matured to specifically address causal factors and failure pathways to a hazardous condition or event. Communication with the software design team is paramount to ensure adequate coverage in preliminary design, detailed design, and testing.

If a PHL is executed on a system that has progressed past the requirements phase, a list or a tree of identified software safety-critical functions becomes helpful to flesh out the fault tree, or the tool used to represent the hazards and their causal factors. In fact, the fault tree method is one of the most useful tools in the identification of specific causal factors in both the hardware and software domains.

During the PHA activities, the link from the software casual factors to the system-level requirements must be established. If there are causal factors that, when inverted descriptively, cannot be linked to a requirement, they must be reported back to the SSWG for additional consideration as well as development and incorporation of additional requirements or implementations into the system-level specifications.

The hazards are formally documented in a hazard tracking database record system. They include information regarding the description of the hazard, casual factors, the effects of the hazard (possible mishaps) and the preliminary design considerations for hazard control. Controlling causal factors reduces the probability of occurrence of the hazard. Performing the analysis includes assessing hazardous components, safety-related interfaces between subsystems, environmental constraints, operation, test and support activities, emergency procedures, test and support facilities, and safety-related equipment and safeguards. A suggested PHA format (Figure 4-24) is defined by the CDRL and can be included in the Hazard Tracking database. This is only a summary of the analytical evidence that needs to be progressively included in the SDL to support the final safety and residual risk assessment in the SAR.

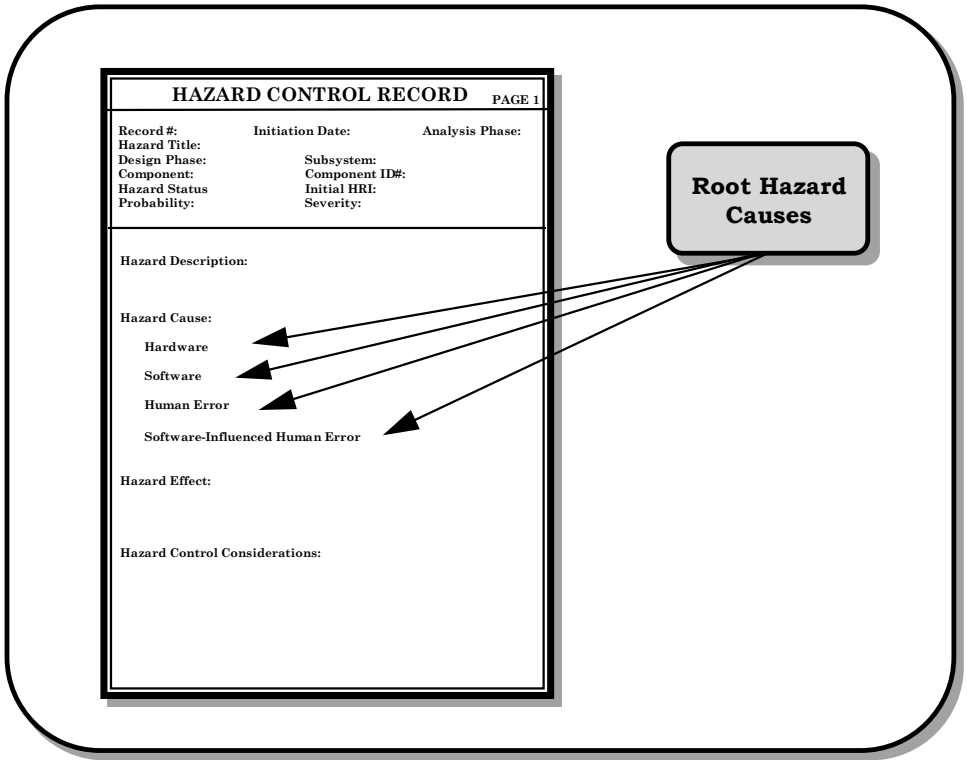


Figure 4-24: Example of a Preliminary Hazard Analysis

The PHA becomes the input document and information source for all other hazard analyses performed on the system including the SSHA, SHA, and the O&SHA.

4.3.4 Derive System Safety-Critical Software Requirements

Safety-critical SSRs are derived from known safety-critical functions, tailored generic SSRs, and hazard causal factors determined from previous activities. Figure 4-25 identifies the software safety engineering process for developing the SRCA.

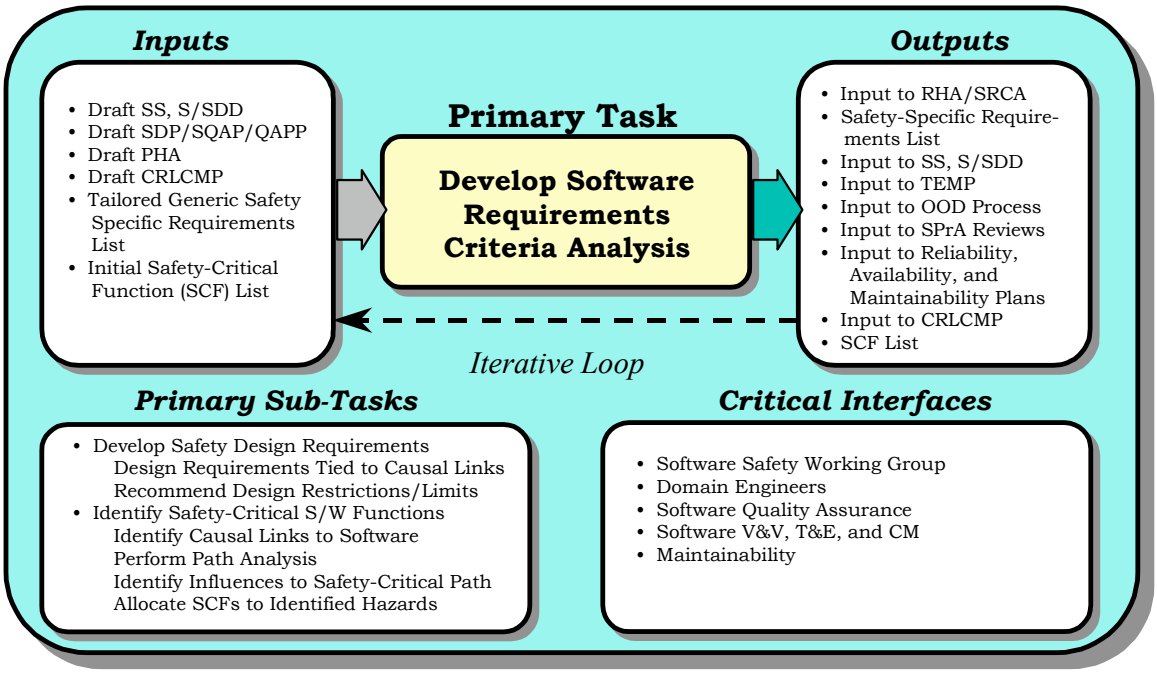


Figure 4-25: Derive Safety-Specific Software Requirements

Safety requirement specifications identify the specifics and the decisions made, based upon the level of safety risk, desired level of safety assurance, and the visibility of software safety within the developer organization. Methods for doing so are dependent upon the quality, breadth, and depth of initial hazard and failure mode analyses and on lessons learned and/or derived from similar systems. As stated previously, the generic list of requirements and guidelines establishes the starting point, which initiates the system-specific SSR identification process. Identification of system-specific software requirements is the direct result of a complete hazard analysis methodology (see Figure 4-26).

SSRs are derived from four sources: generic lists, analysis of the system functionality (safety design requirements), from the causal factor analysis, and from implementation of hazard controls. The analysis of system functionality identifies those functions in the system that, if not properly executed, can result in an identified system hazard. Therefore, the correct operation of the function related to the safety design requirements is critical to the safety of the system making them safety-critical as well. The software causal factor analysis identifies lower-level design requirements that, based on their relationship to safety-critical functions, or the context of the failure pathway of the hazard make them safety-related as well. Finally, design requirements developed to mitigate other system-level hazards (e.g., monitors on safety-critical functions in the hardware) are also SSRs.

The SwSE must present the SSRs to the customer via the SwSWG for concurrence with the assessment as to whether they eliminate or resolve the hazardous condition to acceptable levels of safety risk prior to their implementation. For most SSRs, there must be a direct link between the requirement and a system-level hazard. The following paragraphs provide additional guidance on developing SSRs other than the generics.

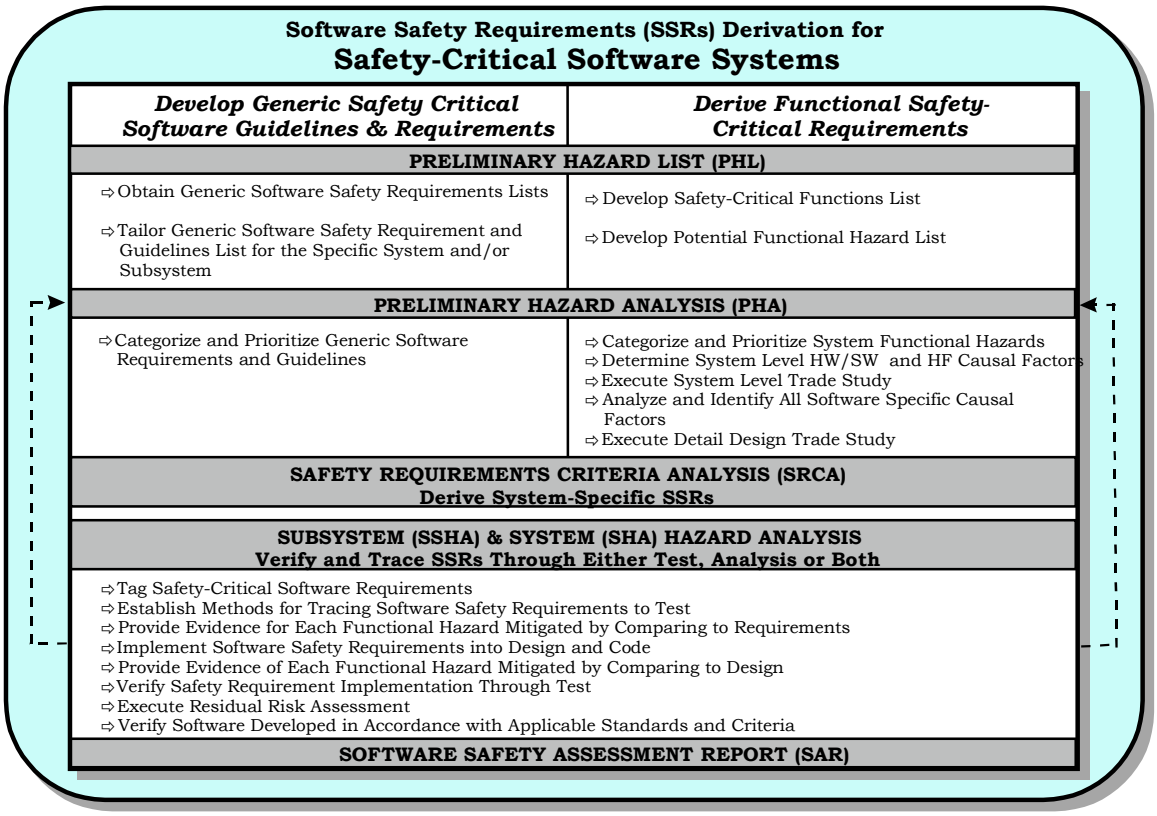


Figure 4-26: Software Safety Requirements Derivation

4.3.4.1 Preliminary Software Safety Requirements

The initial attempt to identify system-specific SSRs evolves from the PHA performed in the early phase of the development program. As previously discussed, the PHL/PHA hazards are a product of the information reviewed pertaining to systems specifications, lessons learned, analyses from similar systems, common sense, and preliminary design activities. The analyst ties the identified hazards to functions in the system (e.g., inadvertent rocket motor ignition to the ARM and FIRE functions in the system software). The analyst flags these functions and their associated design requirements as safety-critical and enters them into the RTM within the SRCA. The analyst should develop or ensure that the system documentation contains appropriate safety requirements for these safety-critical functions (e.g., ensure that all safety interlocks are satisfied prior to issuing the ARM command or the FIRE command). Lower levels of the specification will include specific safety interlock requirements satisfying these preliminary SSRs. These types of requirements are safety design requirements.

The safety engineer also analyzes the hazards identified in the PHA to determine the potential contribution by the software. For example, a system design requires the operator to actually commit a missile to launch, however, the software provides the operator a recommendation to fire the missile. This software is also safety-critical and must be designated as such and included in the RTM. Other safety-critical interactions may not be as obvious and will require more in-depth analysis of the system design. The analyst must also analyze the hazards identified in the

PHA and develop preliminary design considerations to mitigate other hazards in the system. Many of these design considerations will include software thus making that software safety-critical as well. During the early design phases, the safety analyst identifies these requirements to design engineering for consideration and inclusion. This is the beginning of the identification of the functionally derived SSRs.

These design considerations, along with the generic SSRs, represent the preliminary SSRs of the system, subsystems, and their interfaces (if known). These preliminary SSRs must be accurately defined in the hazard tracking database for extraction when reporting the requirements to the design engineering team.

4.3.4.2 Matured Software Safety Requirements

As the system and subsystem designs mature, the requirements unique to each subsystem also mature via the SSHA. The safety engineer, during this phase of the program, attends design reviews and meetings with the subsystem designers to accurately define the subsystem hazards. The safety engineer documents the identified hazards in the hazard tracking database and identifies and analyzes the hazard "causes." When using fault trees as the functional hazard analysis methodology, the causal factors leading to the root hazard determine the derived safety-critical functional requirements. It is at this point in the design that preliminary design considerations are either formalized and defined into specific requirements, or eliminated if they no longer apply with the current design concepts. The SSRs mature through analysis of the design architecture to connect the root hazard to the causal factors. The analyst continues the causal factors' analysis to the lowest level necessary for ease of mitigation (Figure 4-27).

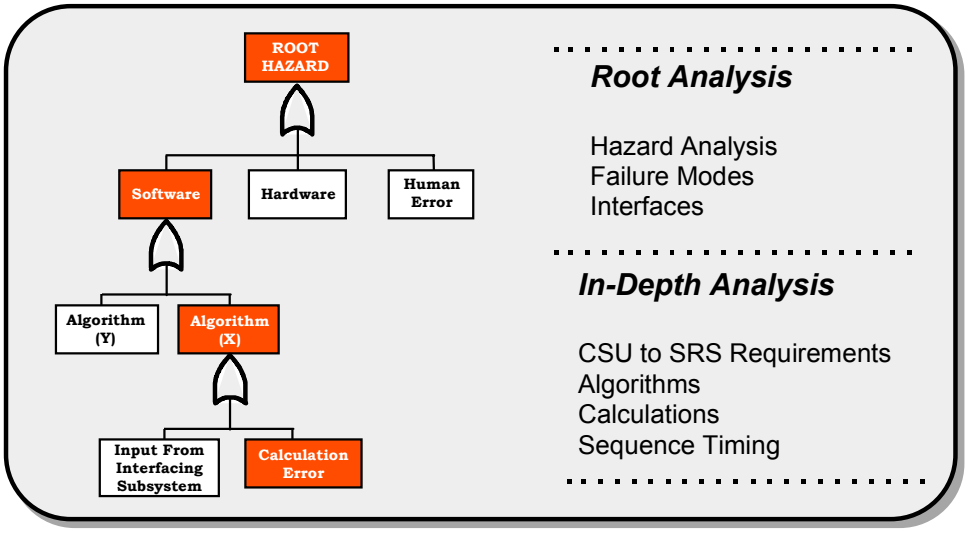


Figure 4-27: In-Depth Hazard Cause Analysis

This helps mature the functional analysis started during preliminary SSR identification. The deeper into the design that the analysis progresses, the more simplistic (usually) and cost effective the mitigation requirements tend to become. Additional SSRs may also be derived from the implementation of hazard controls (i.e., monitor functions, alerts to hazardous conditions

outside of software, unsafe system states, etc.). The PHA phase of the program should define causes to at least the CSCI level, whereas the SSHA and SHA should analyze the causes to the algorithm level for areas designated as safety-critical.

4.3.4.3 The subsystem analysis begins during concept exploration and continues through the detailed design and CDR. The safety analyst must ensure that the safety analyses keep pace with the design. As the design team makes design decisions and defines implementations, the safety analyst must reevaluate and update the affected hazard records.Documenting Software Safety Requirements

The SRCA should document all identified SSRs. The objective of the SRCA is to ensure that the intent of the SSRs in the system software is met and that the SSRs eliminate, mitigate, and/or control the identified causal factors. Mitigating and/or controlling the causal factors reduces the probability of occurrence of the hazards identified in the PHA. The SRCA also provides the means for the safety engineer to trace each SSR from the system level specification, to the design specifications, to individual test procedures and test results' analysis. The safety engineer uses this traceability, known as a RTM, to verify that all SSRs can be traced from system level specifications to design to test. The safety engineer should also identify all safety-critical SSRs to distinguish them from safety-significant SSRs in the RTM. Safety-critical SSRs are those that directly influence a safety-critical function (software control categories 1 and 2), while safety-significant SSRs are those that indirectly influence safety-critical functions. In terms of the Nuclear Safety Analysis process, these are first- and second-level interfaces, respectively. The RTM provides a useful tool to the software development group. They will be immediately aware of the safety-critical and safety-significant functions and requirements in the system. This will also alert them when making modifications to safety-critical CSCIs and CSUs that may impact SSRs. The SRCA is a "living" document that the analyst constantly updates throughout the system development.

4.3.4.4 Software Analysis Folders

At this stage of the analysis process, it is also a good practice to start the development of SAFs. The purpose of a SAF is to serve as a repository for all of the analysis data generated on a particular CSCI by the safety engineer. SAFs should be developed on a CSCI basis and should be made available to the entire SSS Team during the software analysis process. Items to be included within the SAFs include, but are not limited to:

- Purpose and functionality of the CSCI, source code listings annotated by the safety engineer,
- Safety-Critical Functions (SCF) and SSRs pertaining to the CSCI under analysis, SSR traceability results,
- Test procedures and test results pertaining to the CSCI,
- Record and disposition of all Program Trouble Reports (PTR)/Software Trouble Reports (STR) generated against the particular CSCI, and

- A record of any and all changes made to the CSCI. SAFs need to be continually updated during the preliminary and detailed design SSHA phases.

4.3.5 Preliminary Software Design, Subsystem Hazard Analysis

The identification of subsystem and system hazards and failure modes inherent in the system under development (Figure 4-28) is essential to the success of a credible software safety program. Today, the primary method of reducing the safety risk associated with software performing safety-critical or significant functions is to first identify the system hazards and failure modes, and then determine which hazards and failure modes are *caused* or *influenced by* software or lack of software. This determination includes scenarios where information produced by software could potentially influence the operator into a wrong decision resulting in a hazardous condition (design-induced human error). Moving from hazards to software causal factors and consequently design requirements to eliminate or control the hazard is very practical, logical, and adds utility to the software development process. It can also be performed in a more timely manner as much of the analysis is accomplished to influence preliminary design activities.

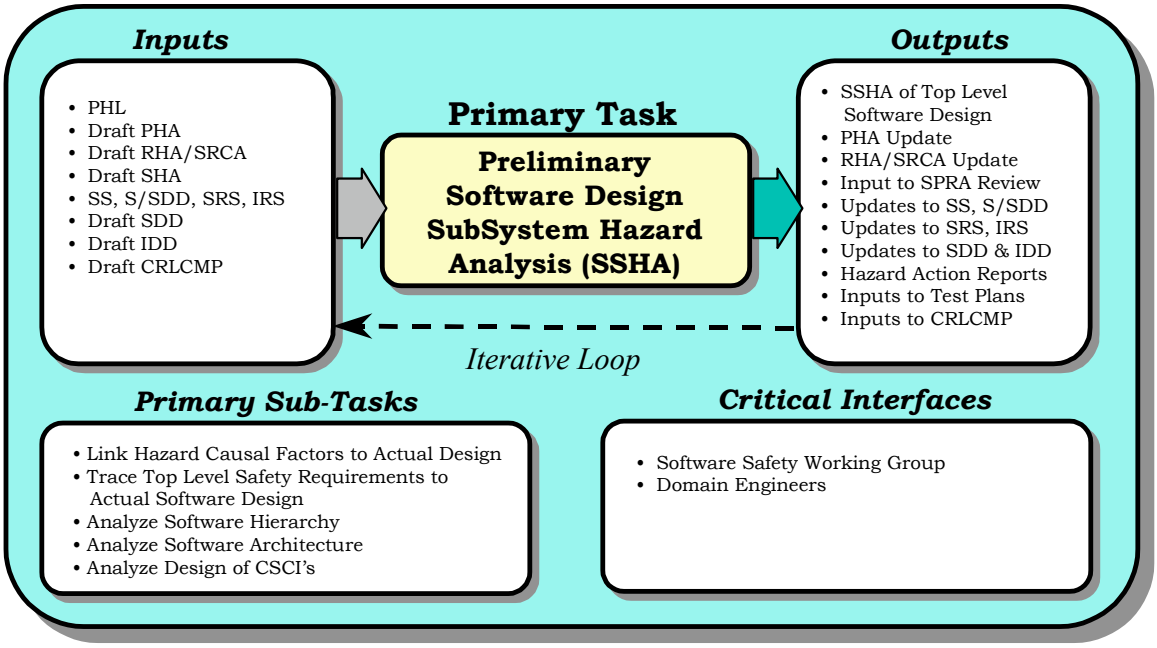


Figure 4-28: Preliminary Software Design Analysis

The specifics of how to perform the SSHA or SHA are briefly described in Appendix C, Paragraphs C.1.6 and C.1.7. MIL-STD-882C, Tasks 204 and 205 and other reference texts in Appendix B provide a more complete description of the processes. The fundamental basis and foundation of a SSP is a systematic and complete hazard analysis process.

One of the most helpful steps within a credible software safety program is to categorize the specific causes of the hazards and software inputs in each of the analyses (PHA, SSHA, SHA, and O&SHA). Hazard causes can be those caused by hardware (or hardware components), software inputs (or lack of software input), human error, software-influenced human error, or

hardware or human errors propagating through the software. Hazards may result from one specific cause or any combination of causes. As an example, "loss of thrust" on an aircraft may have causal factors in all four categories. Examples are as follows:

Hardware: foreign object ingestion,

Software: software commands engine shutdown in the wrong operational scenario,

Human error: pilot inadvertently commands engine shutdown, and

Software-influenced pilot error: computer provides incorrect information insufficient or incomplete data to the pilot causing the pilot to execute a shutdown.

Whatever the cause, the safety engineer must identify and define hazard control considerations (PHA phase) and requirements (SSHA, SHA, and O&SHA phases) for the design and development engineers. He/she communicates hardware-related causes to the appropriate hardware design engineers, software-related causes to the software development and design team, and human error-related causes to the Human Factors organization or to hardware and/or software design team. He/she must also report all requirements, along with supporting rationale, to the systems engineering team for their evaluation, tracking, and/or disposition.

The preliminary software design SSHA begins upon the identification of the software subsystem and uses the derived system-specific SSRs. The purpose is to analyze the system and software architecture and preliminary CSCI design. At this point, the analyst has identified (or should have identified) all SSRs (i.e., safety design requirements, generics, and functional derived requirements and hazard control requirements) and begins allocating them to the identified safety-critical functions and tracing them to the design.

The allocation of the SSRs to the identified hazards can be accomplished through the development of SSR verification trees (Figure 4-29) which links safety-critical and safety-significant SSRs to each SCF. The SCFs in turn are already identified and linked to each hazard. Therefore, by ensuring that the SCF has been safely implemented the hazard will be controlled. The tree allows the SwSE to verify that controls have been designed into the system to eliminate or control/mitigate the SCF. The root node of the tree represents one SCF. The safety analyst needs to develop a verification tree for each system-level SCF. The second level nodes are the safety-critical SSRs linked to each system-level SCF. The third-, fourth-, and lower level nodes represent the safety-critical and/or safety-significant SSRs allocated to each SCF. The fourth- and fifth-level nodes are developed as required to fulfill the level of detail required by the SSS Team. By verifying the nodes through analysis and/or testing, the safety analyst essentially verifies the correct design implementation of the requirements. The choice of analysis and/or testing to verify that the SSRs is up to the individual safety engineer whose decision is based on the criticality of the requirement to the overall safety of the system and the nature of the SSR.

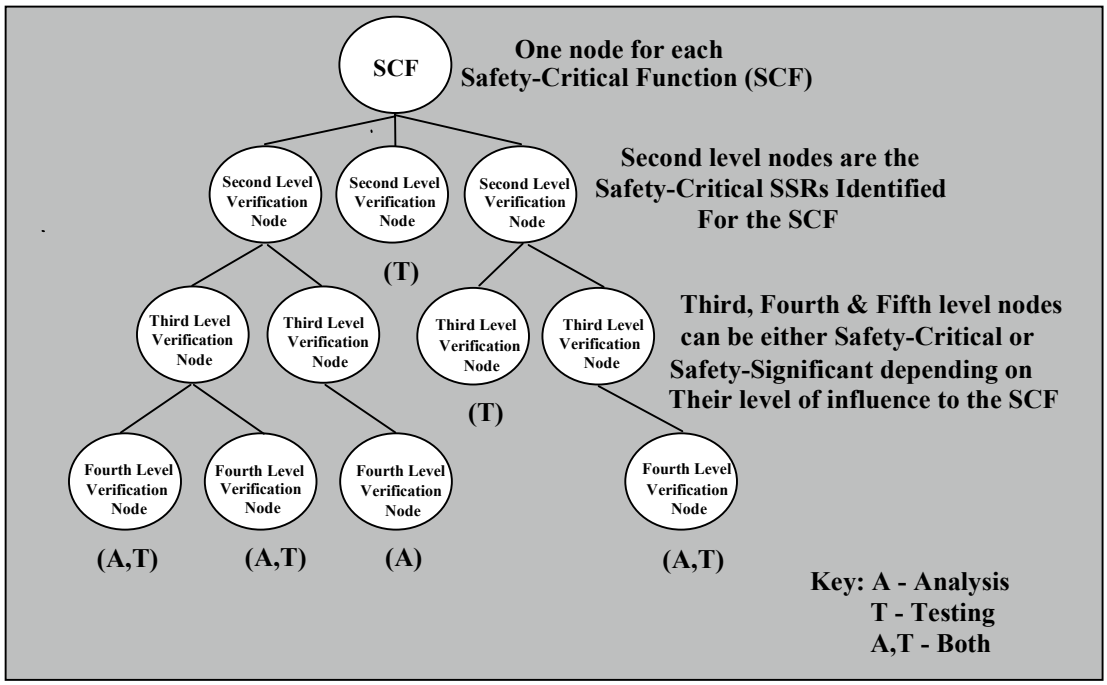


Figure 4-29: Software Safety Requirements Verification Tree

Whenever possible, the safety engineer should use testing for verification. He/she can develop an SSR Verification Matrix, similar to Table 4-2, to track the verification of each SSR or directly document the verification in the RTM. The choice is largely dependent on the size and complexity of the system. The SSR matrix, if developed, should be included as an appendix to the SRCA; and the data should feed directly into the RTM. The safety analyst should also update the hazard tracking database and SAFs with the analysis and test results once the verification is complete.

Table 4-2: Example of a Software Safety Requirements Verification Matrix

SSR	Test/Analysis	Verified (Date)	Comments
1	**Test:** (TP4-1 & TP72-1) **Analysis:** (CSCI/CSU Name)	7/14/97	**Test Passed**. Test Data found on Data Extract Iomega Jaz Disk #10
1.2	**Test:** (TP2-2)	9/1/97	**Test Failed**: STR/PTR JDB002 generated & submitted to design team
1.3	**Analysis:** (CSCI/CSU Name)	9/23/97	Analysis of CSCI/CSU (***Name***) indicated successful implementation of the algorithm identified by SSR 1.3

The next step of the preliminary design analysis is to trace the identified SSRs and causal factors to the design (to the actual CSCIs and CSUs). The RTM is the easiest tool to accomplish this task (see Table 4-3). Other methods of preliminary design hazard analysis include Module Safety-Criticality Analysis and Program Structure Analysis, which are discussed below.

Table 4-3: Example of a RTM

SSR	Requirement Description	CSCI	CSU	Test Procedure	Test Results

4.3.5.1 Module Safety-Criticality Analysis

The purpose of module (CSCI or CSU) safety-criticality analysis is to determine which CSCIs or CSUs are safety-critical to the system in order to assist the safety engineer in prioritizing the level of analysis to be performed on each module. The safety analyst bases the priority on the degree at which each CSCI or CSU implements a specific safety-critical function. The analyst should develop a matrix (example in Table 4-4) to illustrate the relationship each CSCI or CSU has with the safety-critical functions. The matrix should include all CSCIs and CSUs required to perform a safety-critical function, such as math library routines which perform calculations on safety-critical data items. The criticality matrix should list each routine and indicate which safety-critical functions are implemented. Symbols could be used to note the importance with respect to accomplishing a safety-critical function.

Table 4-4: Safety-critical Function Matrix

	Safety-Critical Functions						
CSCI/CSU Name	**1**	**2**	**3**	**4**	**5**	**6**	**Rating**
INIT	M		M	M			M
SIGNAL		H	M				H
D1HZ					H		H
CLEAR				H		H	H
BYTE							N

H - High: The CSCI or CSU is directly involved with a critical factor
M - Medium: The CSCI or CSU is indirectly involved or subordinate to a critical factor
N - None: The CSCI or CSU does not impact a safety-critical function.

The last column in the matrix is the overall criticality rating of the CSCI or CSU. The analyst should place an "H," "M," or "N" in this column based on the highest level of criticality for that routine. However, if a CSCI or CSU has a medium criticality over a number of SCFs, the cumulative rating may increase to the next level.

4.3.5.2 Program Structure Analysis

The purpose of program structure analysis is to reconstruct the program hierarchy (architecture) and overlay structure, and to determine if any safety related errors or concerns exist in the structure. The program hierarchy should be reconstructed on a CSCI level in the form of a control tree. The SSE begins by identifying the highest CSU and its call to other CSUs. He/she

performs this process for each level of CSUs. When this control flow is complete, the safety engineer identifies recursive calls, extraneous CSUs, inappropriate levels of calls, discrepancies within the design, calls to system and library CSUs, calls to other CSCIs, overlays, or CSUs not called. CSUs called by more than one name, and units with more than one entry point are also identified. Figure 4-30 provides an example hierarchy tree.

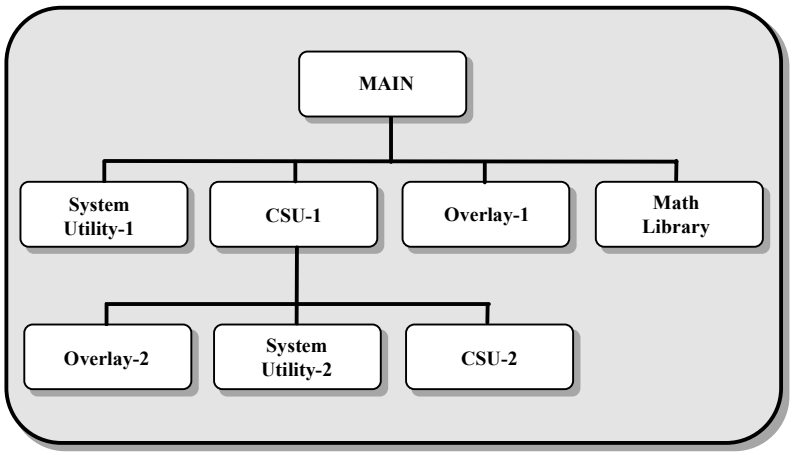


Figure 4-30: Hierarchy Tree Example

All overlays should be identified. After identification, the following issues should be considered.

- Overlaying is not performed unnecessarily (An overlay should not just load another overlay).
- Safety-critical code should not be in overlays.
- All overlay loads should be verified and proper actions taken if an overlay cannot be loaded. (In some cases, the system will halt; while in others, some recovery is sufficient, depending on the criticality and impact of the failure.)
- The effect that the overlays structure has on the time-critical code.
- Interrupts are enabled when an overlay is loaded.
- A review of which overlays are loaded all the time, to determine if they should be made into resident code to cut down on system overhead.
- Overlays comply with the guidelines of STANAG 4404.

4.3.5.3 Traceability Analysis

The analyst develops and analyzes the RTM to identify where the SSRs are implemented in the code, SSRs that are not being implemented, and code that does not fulfill the intent of the SSRs. The traced SSRs should not just be those identified by the top-level specifications, but those SSRs identified by the SRS, Software Design Document (SDD), and Interface Control Document (ICD)/Interface Design Specification (IDS). This trace provides the basis for the analysis and

test planning by identifying the SSRs associated with all of the code. This analysis also ties in nicely with the SRCA (see Paragraph 4.3.5), which not only traces SSRs from specifications to design and test but also identifies what is safety-critical and what is not.

Tracing encompasses two distinct activities: a requirement-to-code trace and a code-to-requirement trace. The forward trace, requirement-to-code, first identifies the requirements that belong to the functional area (if they are not already identified through requirement analysis). The forward trace then locates the code implementation for each requirement. A requirement may be implemented in more than one place thus making the matrix format very useful.

The backward trace, code-to-requirement, is performed by identifying the code that does not support a requirement or a necessary "housekeeping" function. In other words, the code is extraneous (e.g., "debugging" code left over from the software development process). The safety analyst performs this trace through an audit of the applicable code after he/she has a good understanding of the corresponding requirements and system processing. Code that is not traceable should be documented and eliminated if practical. The following items should be documented for this activity:

- Requirement-to-code trace,
- Unit(s) [code] implementing each requirement,
- Requirements that are not implemented,
- Requirements that are incompletely implemented,
- Code-to-requirement trace, and
- Unit(s) [code] that are not directly or indirectly traceable to requirements or necessary "housekeeping" functions.

4.3.6 Detailed Software Design, Subsystem Hazard Analysis

Detailed design level analysis (Figure 4-31) follows the preliminary design process where the SSRs were traced to the CSCI level and is initiated after the completion of the PDR. Prior to performing this process, the safety engineer should have completed the development of the fault trees for all of the top-level hazards, identifying all of the potential causal factors and deriving generic and functional SSRs for each causal factor.

This section will provide the necessary guidance to perform a Detailed Design Analysis (DDA) at the software architecture level. It is during this process that the SSE works closely with the software developer, and IV&V engineers to ensure that the SSRs have been implemented as intended, and to ensure that their implementation has not introduced any other potential safety concerns.

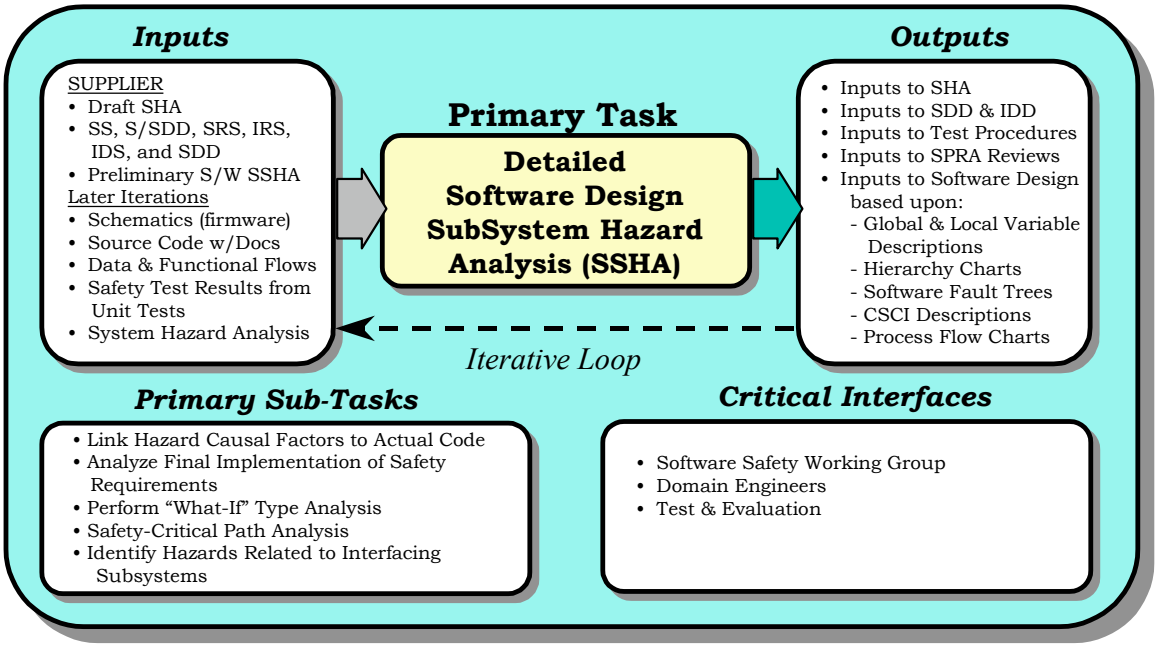


Figure 4-31: Detailed Software Design Analysis

4.3.6.1 Participate in Software Design Maturation

DDA provides the SSS engineer and the software development experts an opportunity to analyze the implementation of the SSRs at the CSU level. DDA takes the SwSE from the CSCI level determined from the preliminary design analysis one step further into the computer software architecture. As the software development process progresses from preliminary design to detailed design and code, it is the responsibility of the safety engineer to communicate the SSRs to the appropriate engineers and programmers of the software development team. In addition, the safety engineer must monitor the software design process to ensure that the software engineers are implementing the requirements into the architectural design concepts. This can only be accomplished by interactive communication between safety engineering and software engineering. It is essential that the software developer and the SwSE work together in the analysis and verification of the SSRs. In today's software environment, the SwSE cannot be expected to be an expert in all computer languages and architectures. Software design reviews, code walkthroughs, and technical interchange meetings will help to provide a conduit of information flow for the "wellness" assessment of the software development program from a safety perspective. "Wellness" in this situation would include how well the software design/programming team understands the system hazards and failure modes attributed to software inputs or influences. It also includes their willingness to assist in the derivation of safety-specific requirements, their ability to implement the requirements, and their ability to derive test cases and scenarios to verify the resolution of the safety hazard. Most programs today are resource limited. This includes most support functions and disciplines to include system safety engineering. If this is the case, there will not be sufficient time in the day, week, or program for the safety team to attend each and every design meeting. It is the responsibility of the safety manager to prioritize those meetings and reviews which have the most "value added"

input to the safety resolution function. This is very dependent on the amount of communication and trust between disciplines, and among team members.

It is important to remember that there is a link between the SSRs and causal factors identified by the FTA during the PHA phase of the software safety process. There are three methods of verifying SSRs: analysis, testing, or both as illustrated in Figure 4-32. Recommended approaches and techniques for analysis will be discussed in the subsequent paragraphs, while approaches for SSR verification through testing will be discussed in Subsection 4.4.

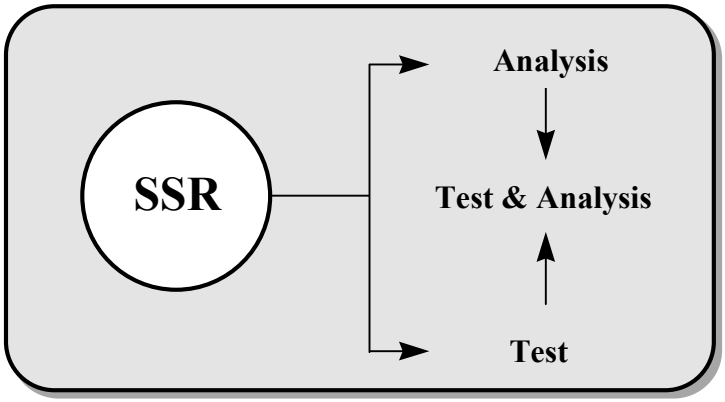


Figure 4-32: Verification Methods

4.3.6.2 Detailed Design Software Safety Analysis

One of the primary analyses performed during DDA is to identify the CSU where an SSR is implemented. The term CSU refers to the code-level routine, function, or module. The best way to accomplish this task is for the SwSE to sit down with the software developer, IV&V engineer, or QA engineer and to begin to tag/link individual SSRs to CSUs, as illustrated in Figure 4-33. This accomplishes two goals. First, it helps focus the SwSE on the safety-related processing, which is more important on large-scale development projects than on smaller, less complex programs. Secondly, it provides an opportunity to continue development of the RTM.

The RTM contains multiple columns, with the left-most column containing the list of SSRs. Adjacent to this column is a description of the SSR, and the name of the CSCI where the individual SSR has been implemented. Adjacent to this column is a column containing the name of the CSU where the SSR has been implemented. The next column contains the name of the test procedure that verifies the implementation of the SSR, and the last column documents test results with pertinent comments. As previously discussed, Table 4-3 illustrates an example of an the RTM. In some cases, it may only be possible to tag/link the SSR to the CSCI level due to the algorithms employed or the implementation of the SSR. If this is the case, the SSR will probably not be verified through analysis, but by an extensive testing effort. The RTM should also be included as part of the SRCA report to provide the evidence that all of the safety requirements have been identified and traced to the design and test.

Once the RTM has been populated and all SSRs have been tagged/linked to the application code, it is time to start analyzing the individual CSUs to ensure that the intent of the SSR has been

satisfied. Again, in order to accomplish this task, it is best to have the appropriate developers and/or engineers available for consultation. Process flow charts and DFDs are excellent examples of soft tools that can aid in this process. These tools can help the engineer review and analyze software safety interlocks such as checks, flags, and firewalls that have been implemented in the design. Process flows and DFDs are also useful in performing "What If" types of analyses, performing safety-critical path analyses, and identifying potential hazards related to interfacing systems. The SAFs should include the products resulting from these safety tasks

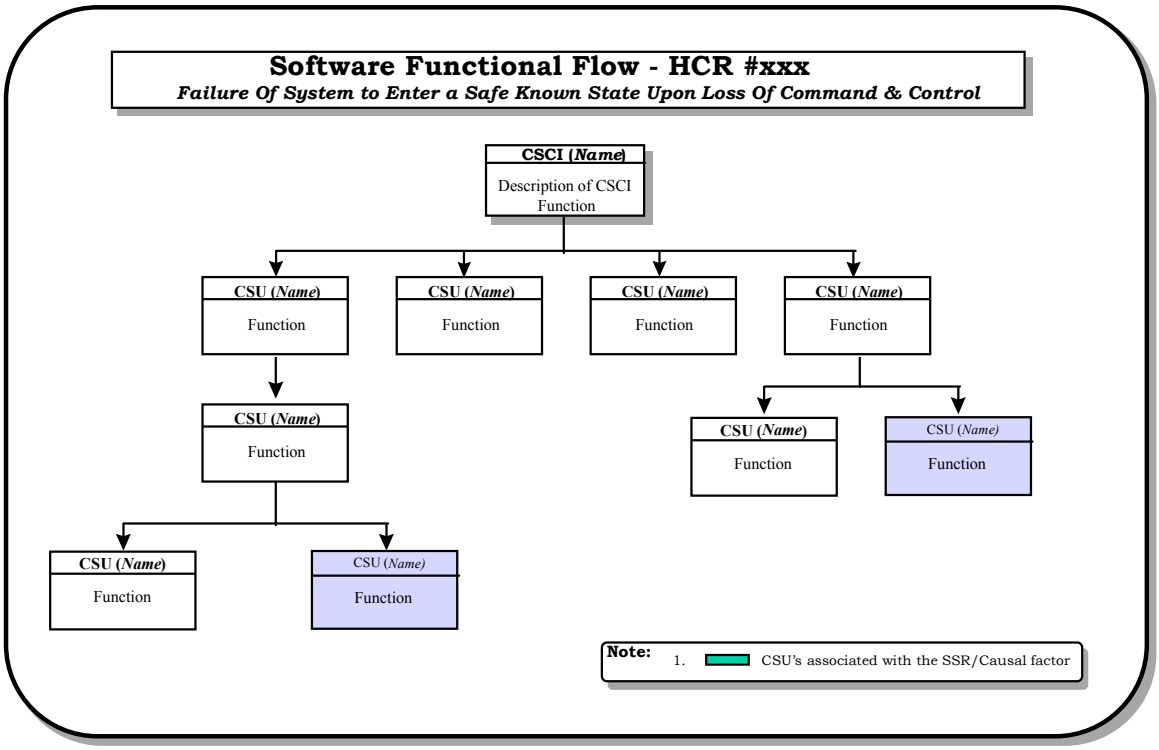


Figure 4-33: Identification of Safety-Related CSUs

4.3.6.2.1 Safety Interlocks

Safety interlocks can either be hardware or software oriented. As an example, a hardware safety interlock would be a key switch that controls a safe/arm switch. Software interlocks generally require the presence of two or more software signals from independent sources to implement a particular function. Examples of software interlocks are checks and flags, firewalls, come-from-programming techniques, and bit combinations.

4.3.6.2.1.1 Checks and Flags

Checks and flags can be analyzed by reviewing the variables utilized by a CSU and ensuring that the variable types are declared and used accurately from CSU to CSU and that they have been logically implemented. As an example, lets look at a simple computerized bank checkbook problem containing two CSUs: Debit and Credit. The Debit CSU utilizes a Boolean variable as a

flag, called "DBT," which it initializes as "1111" or "True" and sets this variable every time the user wishes to make a withdrawal. Meanwhile, the Credit CSU utilizes the same flag for making deposits; however, it sets it as "1111" or "True" every time the user wishes to make a deposit. This is a simple logic error that could have been the result of two separate programmers not communicating or possibly one programmer making a logic error. This was a simple error and although not life threatening, could cost either the bank or user money simply because the programmer did not utilize unique flags for both the Credit and Debit CSUs. A more life threatening example might be found in a hospital that utilizes computers to administer medication to patients. Within this system, there is one particular CSU that sets a flag every 6 hours that signals the machine to administer medication. However, due to the flag being implemented within the wrong timer routine (i.e., logic error), the machine sets the flag every hour resulting in an overdose of medication to the patient.

4.3.6.2.1.2 Firewalls

Software developers, to isolate one area of software processing from another, utilize firewalls. Generally, they are used by software developers to isolate safety-critical processing from non-safety-critical processing. As an example, lets assume that in our medication dosage example there is a series of signals that must be present in order for the medication to be administered. The actual administration of the medication would be considered as safety-critical processing, while the general processing of preparing the series of signals would be non-critical. However, it does take the combination of all of the signals to activate the administration of the medication. The absence of any one of those signals would inhibit the medication from being administered. Therefore, it would take multiple failures to cause the catastrophic event. In our checks and flags example, this type of safety interlock would have prevented the failure of one event causing an overdose.

4.3.6.2.1.3 Come-From Programming

Come-From programming is another example of the implementation of safety interlocks; however, it is extremely rigorous to implement; and there are very few compilers available on the market that will support this technique. Come-from programming is just another way of protecting or isolating safety-critical code from non-safety-critical code. The difference in this technique is that it requires the application processing to know where it is at all times by using a Program Counter (PC) and to know where it has been (i.e., where it has "Come-From"). By knowing where it is and what the previous processing has been, the application can make validity checks to determine if the processing has stepped outside of its intended bounds. Lets use the medication example again. This time lets require the safety-critical processing CSU, "ADMIN" to only accept an "administer dose" request from CSU "GO." The "ADMIN" CSU would then perform a validity check on the origin of the request. An answer of NOT "GO" would result in a reject and "ADMIN" would either ignore the request, or perform some type of error processing. This type of processing also prevents inadvertent jumps from initiating safety-critical functions. In our example, if we suddenly had an inadvertent jump into the "ADMIN" routine, the value of our new PC would be compared to the value of our previous PC. Having preprogrammed ADMIN to only accept the PC from the "GO" CSU, ADMIN would recognize the error and perform the appropriate error processing.

4.3.6.2.1.4 Bit Combinations

Bit combinations are another example of implementing safety interlocks in software. Bit combinations allow the programmer to concatenate two or more variables together to produce one variable. This one variable would be the safety-critical variable/signal, which would not be possible without the exact combination of bits present in the two variables that were concatenated together.

4.3.6.2.2 "What If" Analysis

"What If" types of analyses are an excellent way to speculate how certain processing will react given a set of conditions. These types of analyses allow the SSE to determine if all possible combinations of events have occurred and to test how these combinations would react under credible and non-credible events. For example, how would the system react to power fluctuation/interrupt in the middle of processing. Would the state of the system be maintained? Would processing restart at the interrupting PC + 1? Would all variables and data be corrupted? These are questions that need to be asked of code which is performing safety-critical processing to ensure that the programmer has accounted for these types of scenarios and system environments. "What If" analysis should also be performed on all "IF," "CASE," and "CONDITIONAL" statements used in safety-critical code to ensure that all possible combinations and code paths have been accounted for, or to avoid any extraneous or undesired code execution. In addition, this will allow the analyst to verify that there are no fragmented "IF," "CASE," or "CONDITIONAL" statements and that the code has been programmed top-down and properly structured.

4.3.6.2.3 Safety-Critical Path Analysis

Safety-critical path analysis allows the SSE the opportunity to review and identify all of the possible processing paths within the software and to identify which paths are safety-critical based on the identified system-level hazards and the predetermined safety-critical functions of the system. In this case, a path would be defined as a series of events that when performed in a series (one after the other) would cause the software to perform a particular function. Safety-critical path analyses uses the identified system-level hazards to determine whether or not a particular function is safety-critical or not safety-critical. Functional Flow Diagrams (FFD) and DFDs are excellent tools for identifying safety-critical processing paths and functions. In most cases, these types of diagrams can be obtained from the software developers or the IV&V team in order to save cost and schedule of redevelopment.

4.3.6.2.4 Identifying Potential Hazards Related to Interfacing Systems

DDA also allows the SSE an opportunity to identify potential hazards that would be related to interfacing systems. This is accomplished through interface analysis at the IDS/ICD level. Erroneous safety-critical data transfer between system-level interfaces can be a contributing factor (causal factor) to a hazardous event. Interface analysis should include an identification of all safety-critical data variables while ensuring that strong data typing has been implemented for all variables deemed safety-critical. The interface analysis should also include a review of the

error processing associated with interface message traffic and the identification of any potential failure modes that would result if the interface fails or the data transferred is erroneous. Failure modes identified should be tied or linked back to the identified system-level hazards.

4.3.6.3 Detailed Design Analysis Related Sub-processes

4.3.6.3.1 Process Flow Diagram Development

Process Flow Diagram (PFD) development is a line-by-line regeneration of the code into flow chart form. They can be developed by using a standard IBM flow chart template or by freehand drawing. PFDs provide the link between the FFD and the DFD and allow the SSE to review processing of the entire system in a step-by-step logical sequence. PFD development is extremely time consuming and costly, which is one of the reasons it is treated as a related sub-process to DDA. If the diagrams can be obtained from the software developers or IV&V team, it is an added bonus; but the benefits of reverse engineering the design into process flow chart form do not provide a lot of value to the safety effort in performing hazard causal factor analysis. The real value of PFD development lies in the verification and validation that the system is performing the way that it was designed to perform. The primary benefit to process flow chart development from a system safety viewpoint is that it allows the SSE an opportunity to really develop a thorough understanding of the system processing, which is essential when performing hazard identification and causal factor analysis. A secondary benefit is that by reverse engineering the coded program into flow chart form, it provides an opportunity for the SSE to verify that all of the software safety interlocks and safety-critical functionality have been implemented correctly and as intended by the top-level design specifications.

4.3.6.3.2 Code-Level Analysis

Code-level analysis is generally reserved only for highly safety-critical code due to the time, cost, and resources required to conduct the analysis. However, in very small applications, code-level analysis may be required to provide adequate assessment of the product. A variety of techniques and tools may be applied to the analysis of code, largely depending on the programming language, criticality of the software, and resources available to the software safety program. The most common method is analysis by inspection. Use of structured analysis methodologies, such as FTA, Petri Nets, data and control flow analyses, and formal methods is also common at all levels of design and complexity. None of the techniques are comprehensive enough to be applied in every situation, and are often used together to complement each other.

Code-level analysis always begins with an analysis of the architecture to determine the flow of the program, calls made by the executive routine, the structure of the modules, the logic flow of each module, and finally the implementation in the code. Regardless of the technique used to analyze the code, the analyst must first understand the structure of the software, how it interacts with the system, and how it interacts with other software modules.

4.3.6.3.2.1 Data Structure Analysis

The purpose of data structure analysis is to verify the consistency and accuracy of the data items utilized by a particular program. This includes how the data items are defined and that this definition is used consistently throughout the code. One of the best ways to analyze data is to construct a table (Table 4-5) consisting of all of the data items utilized. The table should contain the name of the data item; the data type (Integer, Real, Boolean); the variable dimension (16, 32, 64 bit); the names of routines accessing the data item, whether the data item is local or global; and the name of the common block (if utilized). The appropriate SAF should contain the product produced from these safety tasks.

Table 4-5: Data Item Example

Data Item Name	Data Type	Data Dimension	Routine Accessing	Global/Local	Common Block
JINCOM	Integer	32 bit	INIT	Global	None
			PROC1 TAB2		

4.3.6.3.2.2 Data Flow Analysis

The purpose of data flow analysis is to identify errors in the use of data that is accessed by multiple routines. Except for a very small application, it would be extremely difficult to determine the data flow path for every data item. Therefore, it is essential to differentiate between those data items that will affect or control the safety-critical functions of a system from those that will not.

DFDs (Figure 4-34) should be developed for all safety-critical data items at both the module and system level to illustrate the flow of data. The appropriate SAF should contain the product produced from DFDs. Data is generally passed between modules in one of two ways, globally (common blocks) and locally (parameter passing). Parameter passing is much easier to analyze, since the program explicitly declares which routines are passing data. Data into and out of common blocks should also be traced, but further information will often have to be recorded to understand which subroutines are involved. A table should be developed to aid in the understanding of data flowing through common blocks and data passing through several layers of parameters. This table should describe for each variable the subroutine accessing the variable, and how the variable is being used or modified. The table should include all safety-critical variables and any other variables whose use is not clear from the DFD and included with the appropriate SAF.

An example of errors that can be found from developing both the data item table and the DFDs is as follows:

- Data which is utilized by a system prior to being initialized,
- Data which is utilized by a system prior to being reset,
- Conditions where data is to be reset prior to its use,
- Unused data items, and
- Unintended data item modification.

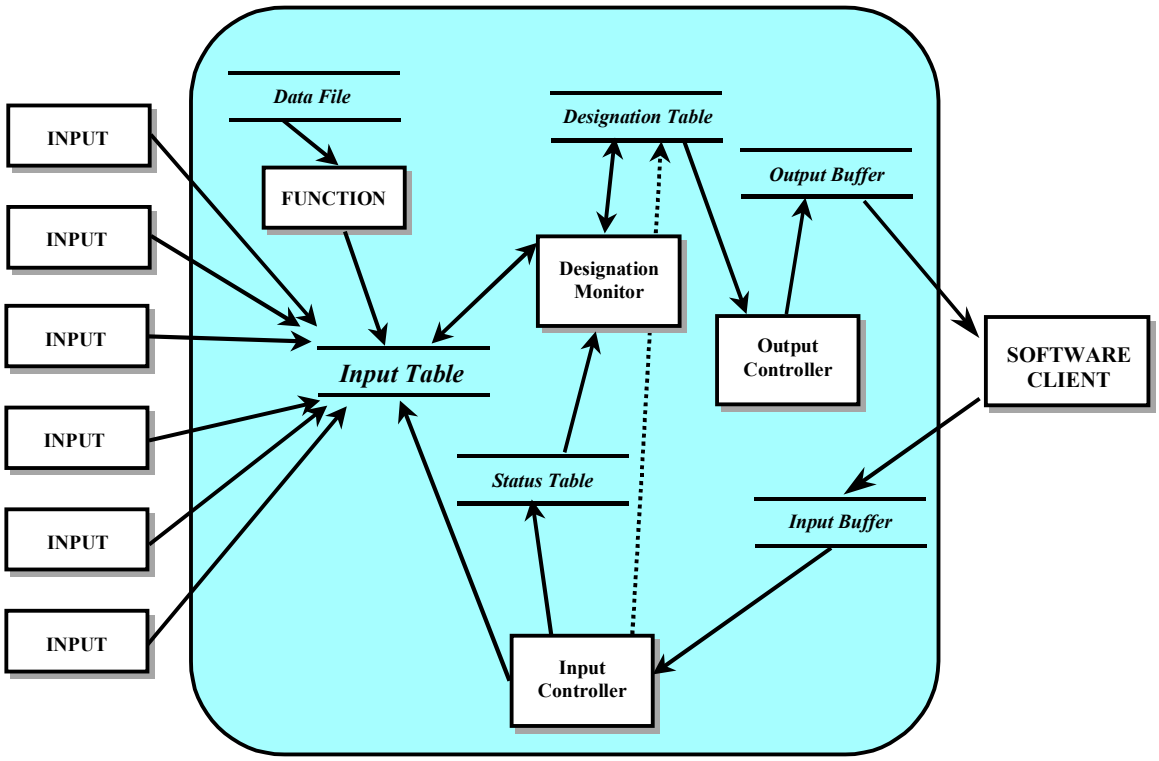


Figure 4-34: Example of a Data Flow Diagram

4.3.6.3.2.3 Control Flow Analysis

The purpose of control flow analysis (flow charting) is to reconstruct and examine the logic of the Program Design Language (PDL) and/or code. Constructing flow charts is one of the first analytical activities that the analyst can perform as it enables the analyst to become familiar with the code and its design architecture. The drawback to flowcharting is that it is generally costly and time consuming. A better approach is to use the team concept and have the safety engineers interface directly with the software developers and system engineers in order to understand system processing. In most cases, the software developers and/or system engineers already have control flow charts developed which can be made available for review by the safety engineer as needed. Each case needs to be evaluated to determine which process would be more beneficial and cost effective to the program. In either case, flow-charting should be done at a conceptual level. Each block of the flow chart should describe a single activity (either single line of code or several lines of code) in a high-level verbal manner, as opposed to simply repeating each line of code verbatim. Examples of both good and bad flow charts can be found in Figure 4-35.

4.3.6.3.2.4 Interface Analysis

The purpose of interface analysis is to verify that the system-level interfaces have been encoded in accordance with the IDS/ICD specifications. Interface analysis should verify that safety-critical data transferred between system-level interfaces is handled properly. Analyses should be performed to verify how the system functionality will perform if the interface is lost (i.e., casualty mode processing). Analyses should also address timing and interrupt analysis issues in

regards to interfaces with safety-critical functions and system components. Performing system-level testing and analyzing the data traffic across safety-critical interfaces is generally the best way to verify a particular interface. Data should be extracted when the system is under heavy stress and low stress conditions to ensure that the message integrity in maintained is accordance with the IDS/ICD.

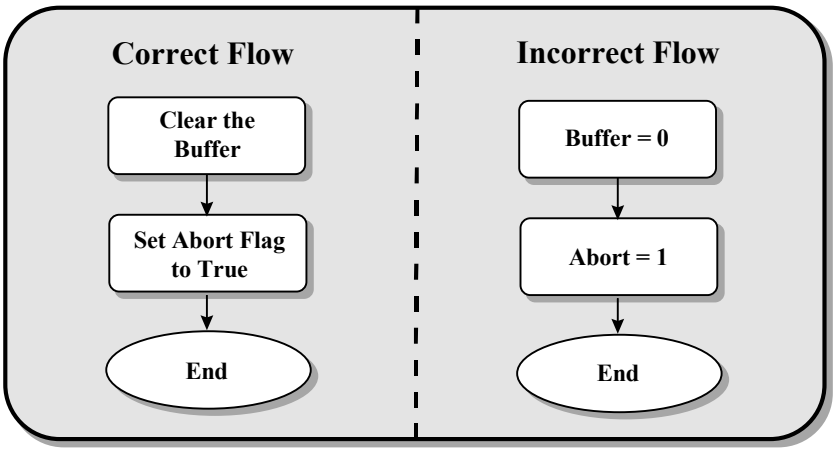


Figure 4-35: Flow Chart Examples

4.3.6.3.2.5 Interrupt Analysis

The purpose of interrupt analysis is two-fold. The SSE must first determine the impact of the interrupts on the code; and second, determine the impact of the prioritization of the program tasks. Flow charts and PDLs are often used to determine what will happen if an interrupt occurs inside a specific code segment. If interrupts are locked out of a particular segment, the safety engineer must investigate how deep the software architecture will allow the interrupts to be stacked, so that none will be lost. If interrupts are not locked out, the safety engineer must determine if data can be corrupted by a low-priority task/process interrupting a high-priority task/process, which changes the value of the same data item.

Performing interrupt analysis in a multi-task environment is a little more difficult, since it is possible for any task to be interrupted at any given point of execution. It is impossible to analyze the effect of an interrupt on every instruction. In this case, it is necessary to determine segments of code that are tightly linked, such as the setting of several related variables. Interrupt analysis should be limited to those segments in a multi-task environment. Items to consider in order to perform interrupt analysis include the following:

- Program segments in which interrupts are locked out
  - ✓ Identify the period of time interrupts are locked out.
  - ✓ Identify the impacts of interrupts being locked out (such as lost messages and lost interrupts).
  - ✓ Identify possible infinite loops (including loops caused by hardware problems).

- Re-entrant code
  - ✓ Are sufficient data saved for each activation?
  - ✓ Is the correct amount of data and system state restored?
  - ✓ Are units that should be re-entrant implemented as re-entrant?
- Code segments which are interruptible
  - ✓ Can the interrupted code be continued correctly?
  - ✓ Will interrupt delay time-critical actions (e.g., missile abort signal)?
  - ✓ Are there any sequences of instructions which should not under any circumstance be interrupted?
- Overall program prioritization
  - ✓ Are functions such as real-time tasks properly prioritized, so that any time-critical events will always be assured of execution?
  - ✓ Is the operator interface of a proper priority to ensure that the operator is monitoring?
- Undefined interrupts
  - ✓ Are they ignored?
  - ✓ Is any error processing needed?

4.3.6.3.2.6 Analysis By Inspection

Although inspection is the most commonly used method of code-level analysis, it is also the least rigorous. That does not lessen its value to the overall safety assessment process. Analysis by inspection is particularly useful for software that is less critical where a less rigorous methodology is appropriate. Analysis by inspection is also frequently used with other techniques, such as FTAs, control flow analyses, etc., to provide a more thorough assessment of the software. Experience shows that these analysis types are generally complementary, each having strong and weak points. Therefore, they often complement each other to a degree. As noted earlier, a single analysis technique is usually not totally sufficient to meet the defined safety objectives.

Analysis by inspection is exactly as its name implies - a process whereby the analyst reviews the software source code (high-level language, assembly, etc.) to determine if there are any errors or structures that could present a potential problem, either in the execution or in the presence of adverse occurrences, such as inadvertent instruction jumps. To some degree, analysis by inspection relies on heuristics, "clue lists," and engineering judgment.

The ability of the analyst to understand the code as written provides an indication of the ability of future software maintainers to understand it for future modifications, upgrades, or corrections. Code should be well structured and programmed in a top-down approach. Code that is

incomprehensible to the trained analyst will likely be incomprehensible to future software maintainers. The code should not be patched. Patched or modified code provides an opportunity for a high probability of errors, since it was rewritten without the benefit of an attendant safety analysis and assessment. The net result is a potentially unsafe program being introduced into a previously "certified" system. Patching the software introduces potential problems associated with the configuration management and control of the software.

"Clue lists" are simply lists of items that have historically caused problems (such as conditional GO-TO statements), or are likely to be problem areas (such as boundary conditions that are not fully controlled). Clue lists are developed over a long period of time and are generally based on the experiences of the analyst, the software development team, or the testing organization. The list below contains several items that have historically been the cause of software problems.

NOTE: It is up to the individual safety engineer to tailor or append this list based on the language and software architecture being utilized.

a. Ensure that all variables are properly defined, and data types are maintained throughout the program.

b. Ensure that, for maintainability, variables are properly defined and named.

c. Ensure that all safety-critical data variables and processing are identified.

d. Ensure that all code documentation (comments) is accurate and that CSCI/CSU headers reflect the correct processing and safety-criticality of the software.

e. Ensure code modifications identified by the STR and date modifications are made.

f. Ensure that processing loops have correct starting and stopping criteria (indices or conditions).

g. Ensure that array subscripts do not go out of bounds.

h. Ensure that variables are correct in procedure call lines (number, type, size, order).

i. Ensure that, for parameters passed in procedure call lines, Input-Only data is not altered, output data is set correctly, and arrays are handled properly.

j. Ensure that all mixed modes of operation are necessary, and clearly documented.

k. Ensure that self-modifying code does not exist.

l. Ensure that there is no extraneous or unexecutable code.

m. Ensure that local variables in different units do not share the same storage locations.

n. Ensure that expressions are not nested beyond 5 levels, and procedures/modules/ subroutine are less than 25 lines of executable code.

o. Ensure that all logical expressions are used correctly.

p. Ensure that processing control is not transferred into the middle of a loop.

q. Ensure that equations are encoded properly in accordance with specifications.

r. Ensure that exceptions are processed correctly. In particular, if the "ELSE" condition is not processed, will the results be satisfactory?

s. Ensure that comparisons are made correctly.

t. Ensure that common blocks are declared properly for each routine they are used in.

u. Ensure that all variables are properly initialized before use.

The thoroughness and effectiveness of an analysis performed by inspection is very dependent on the analyst's expertise; his/her experience with the language; and, to a lesser degree, the availability of the above mentioned clue lists. Clue lists can be developed over a period of time and passed on to other analysts. Unfortunately, analysts often tend to keep these lists secret. Many of the design guidelines and requirements of STANAG 4404 are based on such clue lists. However, in this document, the individual clues have been transformed into design requirements (see Appendix E).

The language used for software development and the tools available to support that development affect the ability to effectively analyze the program. Some languages, such as Ada and Pascal, force a structured methodology, strong information hiding, and variable declaration. However, they introduce complexities and do not support certain functions that are often necessary in system development. Therefore, other languages often augment them. Other languages, such as C and C++, easily support object-oriented programming, data input/output structures, and provide substantial flexibility in coding. However, they provide for the construction of extremely complex program statements and information hiding that are often incomprehensible even to the programmer, while not enforcing structured programming and modularization.

Hardware, especially the microprocessor or micro-controller, can have a significant influence on the safety of the system irrespective of the computer program. Unique aspects of the hardware may also affect the operation of the machine code in an unexpected manner. Design engineers, especially those often referred to as "bit-fiddlers," take great pride in being able to use unique characteristics of the hardware to increase the efficiency of the code or to make the reading of the machine code as obscure as possible. Occasionally, assemblers also use these unique hardware aspects to increase the efficiency and compactness of the machine code; however, it can pose limitations and possible safety risks.

4.3.6.3.2.7 Code Analysis Software Tools

A couple of software tools used to aid in the code analysis process are a tool called "WINDIFF" and one called "TXTPAD32."

WINDIFF, which is a purchased MS product provides color-coded, in-line source-code comparisons for performing delta analysis. Shareware equivalents do exist for this capability.

TXTPAD32 is a "shareware" 32-bit text editor that has many features that expedite code analysis, such as Find In Files for string searches in multiple files and directories, DOS command

and macro execution, bracket matching (parenthesis, braces, brackets), line numbering, color-coding, etc.

4.3.7 System Hazard Analysis

The SHA is accomplished in much the same way as the SSHA. That is, hazards and hazard causal factors are identified; hazard mitigation requirements communicated to the design engineers for implementation; and the implementation of the SSRs are verified. However, several differences between the SSHA and SHA are evident. First, the SHA is accomplished during the acquisition life cycle where the hardware and software design architecture matures. Second, where the SSHA focused on subsystem-level hazards, the SHA refocuses on system-level hazards that were initially identified by the PHA. In most instances, the SHA activity will identify additional hazards and hazardous conditions; because the analyst is assessing a more mature design than that which was assessed during the PHA activity. And third, the SHA activity will put primary emphasis on the physical and functional interfaces between subsystems, operational scenarios, and human interfaces.

Figure 4-36 graphically represents the primary sub-tasks associated with the SHA activity. Due to the rapid maturation of system design, the analysis performed at this time must be in-depth and as timely as possible for the incorporation of any SSRs derived to eliminate or control the system-level hazards. As with the PHA and the SSHA, the SHA must consider all possible causes of these hazards. This includes hardware causes, software causes, human error causes, and software-influenced human error causes. The activity of analyzing hazard causal factors to the level, or depth, necessary to derive mitigation requirements will aid in the identification of physical, functional, and zonal interfaces.

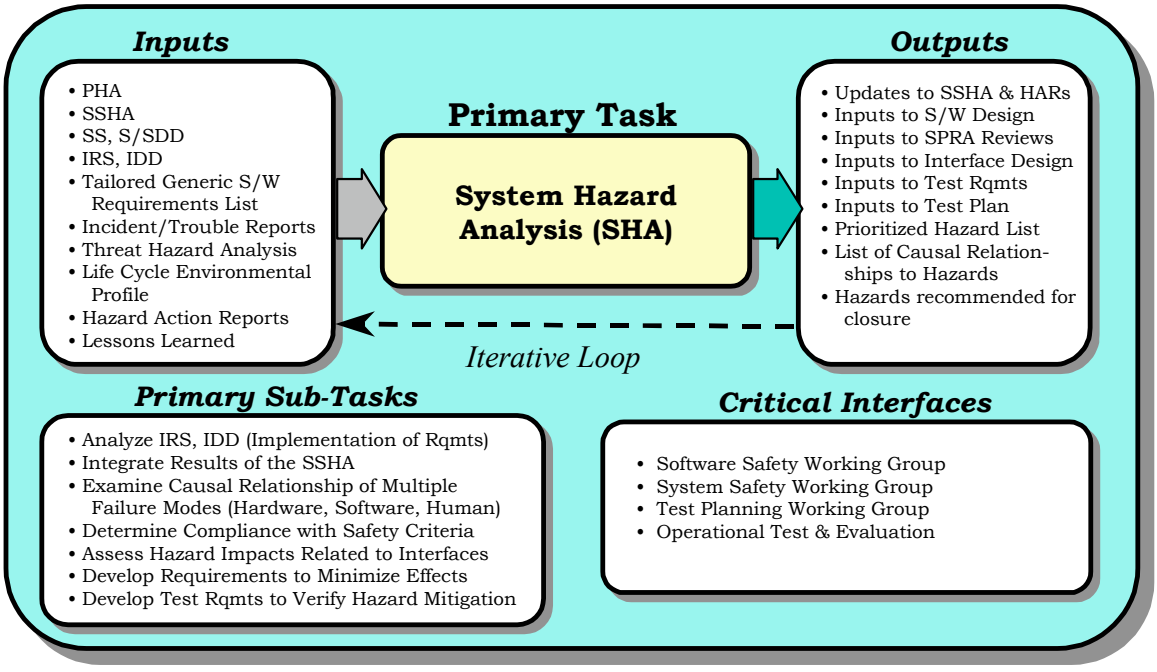


Figure 4-36: System Hazard Analysis

In a majority of the hazards, the in-depth causal factor analysis will identify failure modes (or causal factor pathways) which will cross physical subsystem interfaces, functional subsystem interfaces, and even contractor/subcontractor interfaces. This is graphically depicted in Figure 4-37.

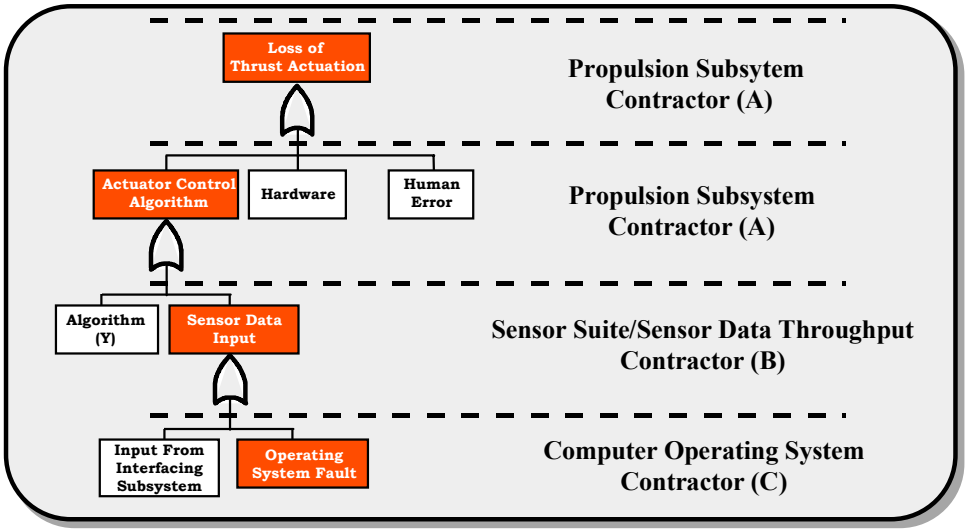


Figure 4-37: Example of a System Hazard Analysis Interface Analysis

In this example, the analyst uses a fault tree approach to analyze a system-level hazard "Loss of Thrust Actuation." This hazard is depicted as the top event of the fault tree. The SHA activity analyzes all causes to the hazard to include the software branch which is a branch of the "OR" gate to the top-level event. Although this hazard would possess hardware causes (actuator control arm failure) or human error causes (pilot commands shutdown of control unit), the software contribution to the hazard will be the branch discussed.

In this example, "Thrust Actuation" is a function of the propulsion system and administratively controlled by the Propulsion IPT of Contractor "A." The computer hardware and software controlling the thrust actuators are also within the engineering boundaries of the same IPT. However, the software safety analyst has determined, in this case, that a fault condition in the computer operating system (OS) is the primary causal factor of this failure mode. This OS fault did not allow actuator sensor data to be read into sensor limit tables and allowed an overwrite to occur in the table. The actuator control algorithm was utilizing this sensor data. In turn, the actuator control computer software component functional architecture could not compensate for loss of credible sensor data which transitioned the system to the hazardous condition. In this example, the actuator and controlling software are designed by Contractor A; the sensor suite and throughput data bus are designed by Contractor B; and the computer OS is developed by Contractor C.

Demonstrated in this example, is the safety analysis performed by Contractor C. If Contractor C is contractually obligated to perform a safety analysis (and specifically a software safety analysis) on the computer OS, the ability to bridge (Bottom-Up Analysis) from an OS software fault to a hazardous event in the propulsion system is extremely difficult. The analysis may identify the potential fault condition, but not identify its system-level effects. The analysis methodology

must rely on the "clients," of the software OS, or Contractor A, to perform the Top-Down analysis for the determination of causal factors at the lowest level of granularity.

In-depth causal factor analysis during the SHA activities will provide a springboard into the functional interface analysis required at this phase of the acquisition life cycle. In addition, the physical and zonal (if appropriate) interfaces must be addressed. Within the software safety activities, this deals primarily with the computer hardware, data busses, memory, and data throughput. The safety analyst must ensure that the hardware and software design architecture is in compliance with the criteria set by the design specification. As the preceding paragraphs pertaining to PHA and the SSHA (preliminary and detailed code analysis) addressed analysis techniques, they will not be presented here. This section will focus primarily on the maturation of the hazard analysis and the evidence audit trail to prove the successful mitigation of system, subsystem, and interface hazards.

Hazard causal factor analysis and the derivation of safety-specific hazard mitigation requirements have been discussed previously in terms of the PHA and SSHA development. In addition, these sections demonstrated a method of documenting all analysis activity in a hazard tracking database to provide the evidence of hazard identification, mitigation, and residual risk. Figure 4-29 (of the PHA) specifically depicted the documentation of hazard causes in terms of hardware, software, human error, and software-influenced human error. As the system design architecture matures, each safety requirement that helps either eliminate or control the hazard must be formally documented (Figure 4-38) and communicated to the design engineers. In addition, the SHA activities must also formally document the results of the interface hazard analysis.

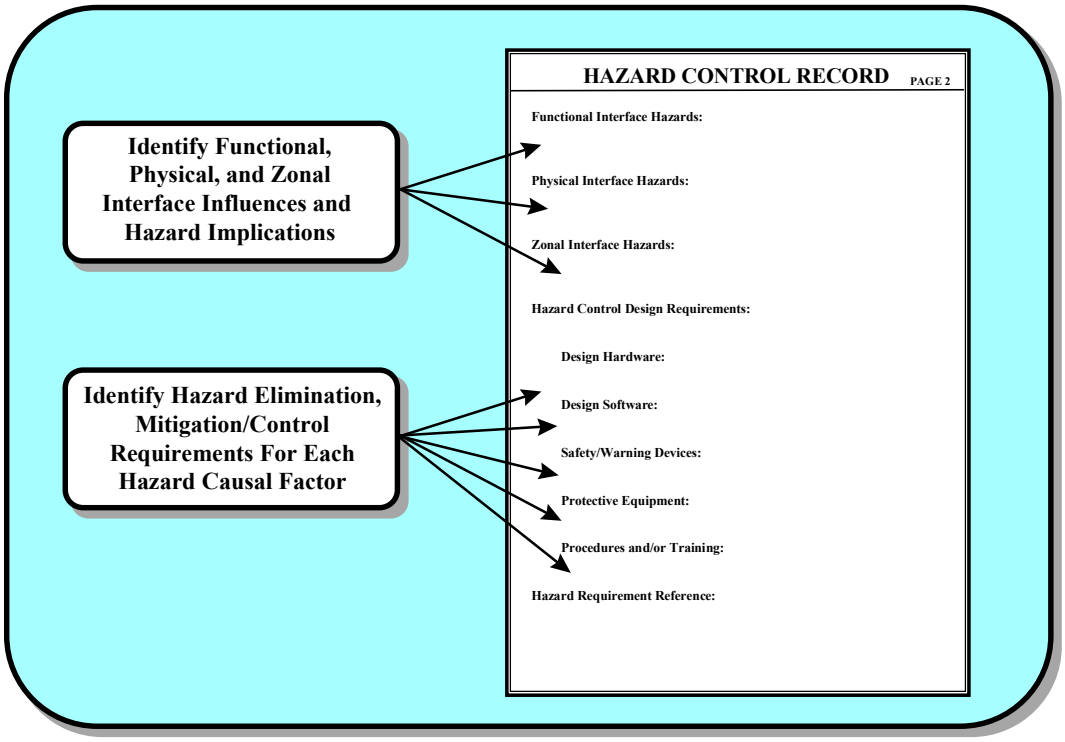


Figure 4-38: Documentation of Interface Hazards and Safety Requirements

At this point, the safety analyst must focus on the ability to define safety test and verification requirements. The primary purpose of this activity is to provide the evidence that all safety requirements identified for hazard elimination or control have been successfully implemented in the system design. It is quite possible that the analyst will discover that some requirements have been implemented in total, others partially, and a few which were not implemented. This is why active involvement in the design, code, and test activities is paramount to the success of the safety effort.

The ability to assess the system design compliance to specific safety criteria is predicated on the ability to verify SSRs through test activities. Figure 4-39 depicts the information required in the Hazard Control Records (HCR) of the database to provide the evidence trail required for the risk assessment activities.

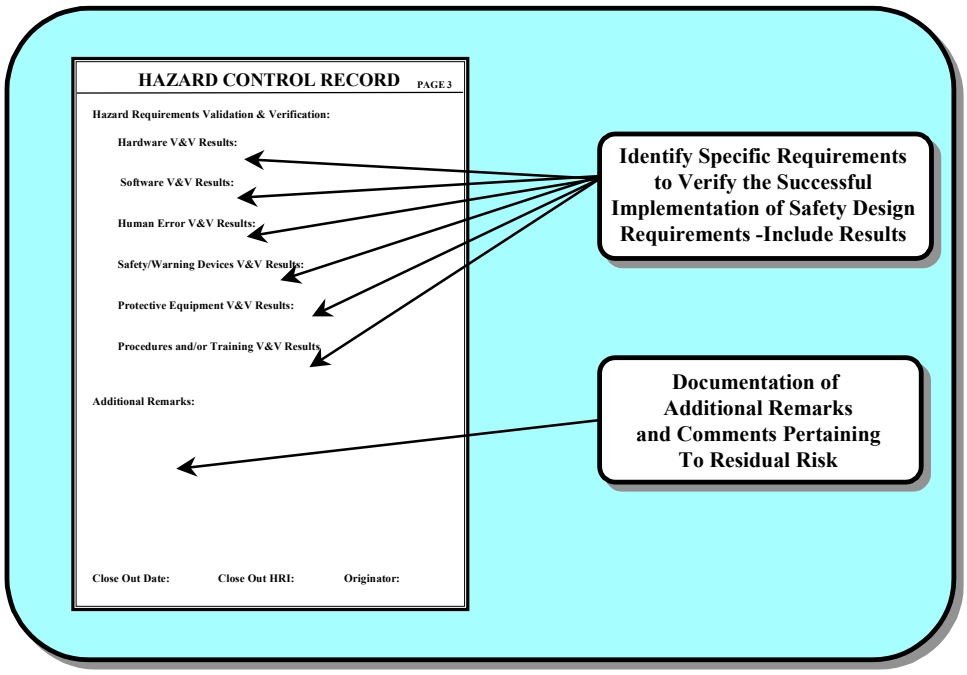


Figure 4-39: Documenting Evidence of Hazard Mitigation

4.4 Software Safety Testing & Risk Assessment

4.4.1 Software Safety Test Planning

This Handbook has focused on the analytical aspects of the SSS program to this point. Analysis represents about half of the total effort in the SSS process. The other half consists of verification and validation of the SSR implementation and the determination and reporting of the residual risk. However, the software safety engineers do not need to perform the testing: that is best left to the testing experts, the test engineering or verification and validation team. Software testing is an integral part of any software development effort. Testing should address not only performance-related requirements, but the SSRs as well. The SSS Team must interface directly with the software developers and verification and validation team to ensure that the code

correctly implements all of the SSRs and that the system functions in accordance with the design specifications.

Illustrated in Figure 4-40, the SSS Team should integrate the safety testing with the normal system testing effort to save time, cost, and resources. The software safety engineers in cooperation with the verification and validation team can identify a significant portion of the SSRs that can be verified through testing.

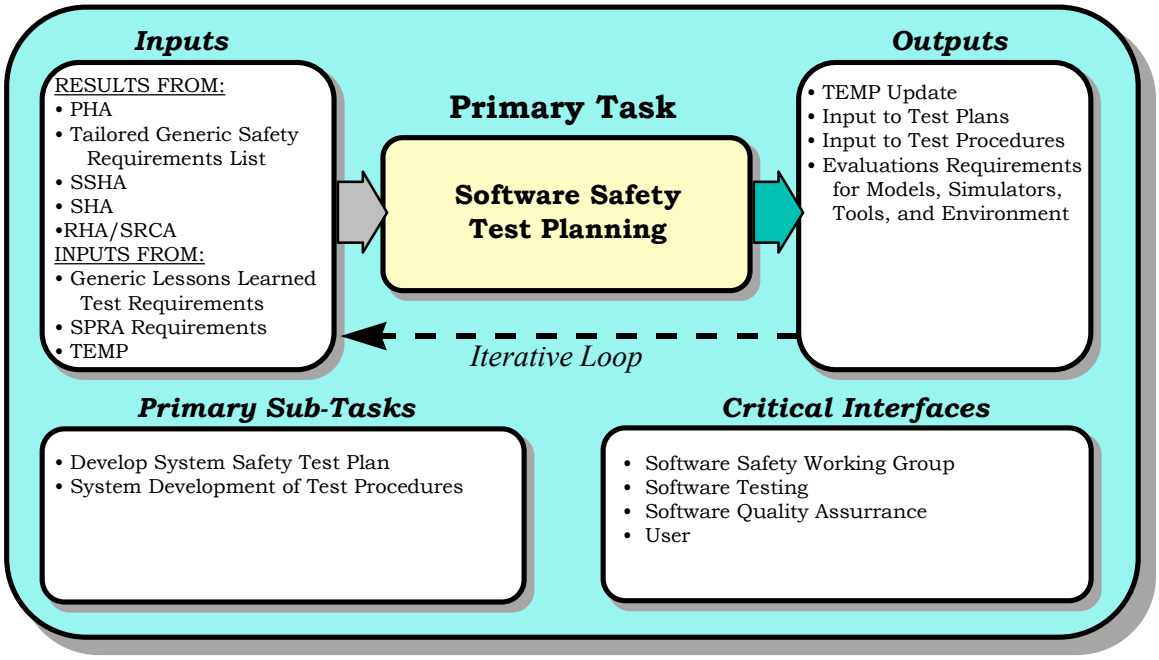


Figure 4-40: Software Safety Test Planning

The software test planning process must address all of the simulators, models, emulators, and software tools that will be utilized by the test team (whether for verification and validation or safety testing) in order to ensure that all processes, requirements, and procedures for validation are in place. This validation must occur prior to use to ensure that the data that they manipulate and interact with is processed as intended. Invalid models, simulators, etc. will invalidate the test results. It is also important that the software safety test plan address how the SwSE will participate in Test Working Group (TWG) meetings and how inputs will be provided to the TRR and SPRA.

Outputs from this software safety test planning process include an updated verification and validation plans, updates to the Test and Evaluation Master Plan (TEMP), as well as evaluation requirements for simulators, models, emulators, test environments and tools. It also includes updates to the Software Test Plan (STP) and recommendations for modifications to test procedures to ensure coverage of all the SSRs identified for test by the RTM and software verification trees within the SRCA.

The safety manager must integrate software safety test planning activities into the overall software testing plan and the TEMP. This integration should include the identification of all safety-critical code and SSRs. The SRCA (Appendix C, Paragraph C.1.8) must include all of the

SSRs. The software safety test planning must also address the testing schedule [Functional Qualification Testing (FQT) and system-level testing] of all SSRs. Safety must integrate this schedule into the overall software test schedule and TEMP. The software safety test schedule will largely depend on the software test schedule and the safety analysis. Beware of test schedule compression due to late software development: the safety schedule must allow sufficient time for the effective analysis of test results. It is also important that the SSS Team identifies system-level test procedures or benchmark tests to verify that the hazards identified during the analyses (SHA, SSHA) have been eliminated, mitigated, or controlled.

During the software safety test planning process, system safety must update the RTM developed during the SRCA by linking requirements to test procedures. Each SSR should have at least one verification and validation test procedure. Many SSRs will link to multiple test procedures. Linking SSRs to test procedures allows the safety engineer to verify that all safety-related software will be tested. If requirements exist that system safety cannot link to existing test procedures, the safety engineer can either recommend that the SSS Team develop test procedures or recommend that an additional test procedure be added to the verification and validation test procedures. There may be cases where SSRs cannot be tested due to the nature of the requirement or limitations in the test setup. In this case, software DDA must be performed.

The SSS Team must review all of the safety-related verification and validation test procedures to ensure that the intent of the SSR will be satisfied by the test procedures. This is also for the development of new test procedures. If a specific test procedure fails to address the intent of a requirement, it is the responsibility of the SSS Team to recommend the appropriate modifications to the verification and validation team during TWG meetings.

4.4.2 Software Safety Test Analysis

Software testing (Figure 4-41) is generally grouped into three levels: unit testing, integration testing, and system integration testing. However, within each level, there are often numerous sublevels. This is especially true in integration testing. In addition to the software under development, the Software Engineering IPT may develop support software to include simulation, emulation, stimulation, run-time environment, data extraction and reduction software, and mathematical models.

The SSS Team must participate in the specification of these programs, just as they do for the application software under development. They will need to specify the capabilities of the simulators and stimulators, such as the ability to induce faults into the system and to test its response to those conditions. The SSS Team must also specify the parameters to be extracted to perform the necessary safety assessment. To a large extent, this process needs to occur up front to permit the software engineering team adequate time to develop these programs.

At the unit level, testing is generally limited to the functionality of the unit or module. However, to test that functionality, the developer requires test drivers and receivers to either stimulate the unit under test or simulate programs with which the unit or module communicates. Integration testing is performed at several levels beginning with building modules from units and gradually progressing up to system-level testing. Integration testing begins with interfacing units that have completed unit-level testing to ensure that they interact properly. Additional units are added until

a configuration item is completely tested. Several sets of integration testing may occur in parallel in complex systems involving several CSCIs. CSCIs are then integrated at the next level until the complete software system is tested. Finally, the total system, hardware and software, is subjected to integration testing at the system level with simulators, stimulators, and run-time environments. Regardless of the degree of sophistication in the testing, laboratory testing is limited by the fact that the environment is very different from the actual environment that the system will be deployed into. This is caused by the limitations on the simulators that can be developed. It may not be practical or desirable to implement many requirements due to the inherent complexity and difficulty in validating these programs. An example of this type of problem occurred in one U.S. Navy system. The system had undergone extensive laboratory integration and system-level testing, however, when the system was fielded for developmental testing, the system shutdown due to the ship's pitch and roll. The laboratory environment had not exercised the ships motion interlocks properly.

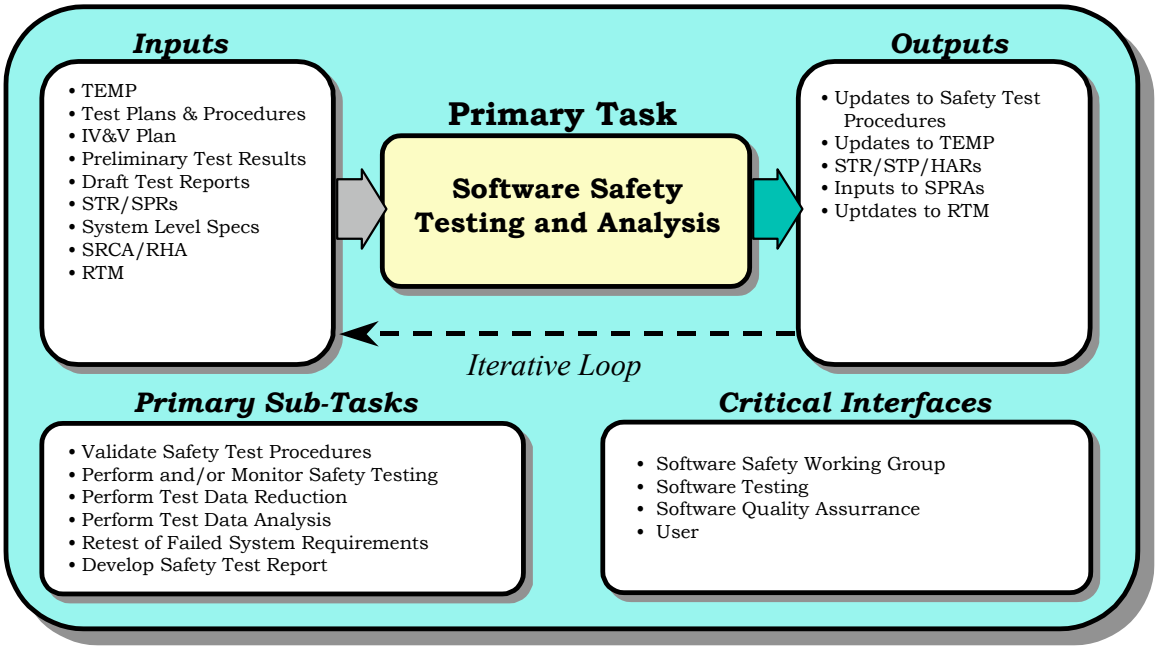


Figure 4-41: Software Safety Testing and Analysis

Testing can represent a large portion of the overall software safety effort. The development of safety design requirements, both generic and system-specific, and the subsequent analysis of their implementation requires that they be verified and validated. Detailed analyses of the design often result in the need to perform tests to verify characteristics, both desirable and undesirable that cannot be verified through analysis. This is especially true of timing and queuing requirements. The testing phase is complete when the SSS Team completes its analysis of the test results and assesses the residual risk associated with the software application in the system.

As noted during the requirements analysis phase, the SSS Team must analyze the implementation of the SSRs to ensure that the intent of the requirement is met. Likewise, in reviewing the test cases and procedures, the SSS Team must ensure that they will validate the correct implementation of the requirements. "Correct" in this context means that the test cases and

procedures verify the intent, not just the letter of the requirement. It differs from the term "correctness" in software engineering that means that the resulting code fulfills the specifications and meets the users' needs. The software safety analyst must ensure that the test environment, test cases, and test procedures will provide the required data to assess safety of the code. Often, the SSS Team can incorporate safety test requirements into the routine testing of the software modules and configuration items at the system level with no impact on the testing schedule or process.

Inherent to the software safety testing process is the need for the SSS Team to provide as much information as possible to the software testers at the module, integration, and system level, regarding the safety-critical aspects of the system and its software. They must identify those portions of the code that are safety-critical, at both a module level and at a functional level. They must also identify those tests that are safety specific or those portions of other tests that have a bearing on the safety aspects of the system. The most effective method of identifying safety-critical code is to establish early in the system development process a means of marking it in the system documentation. Providing the RTM (and updates) to the test team also provides them an opportunity to review all the SSRs and to understand which requirements are safety-critical and which is safety-significant. With a little training, the test team can identify safety anomalies and provide safety trouble reports against the SSRs and other requirements.

As part of the safety test cases, the SSS Team must provide testers with a list of the test environment conditions (e.g., simulators, drivers, etc.), test procedures, expected and undesired outcomes, and the data extraction requirements. Often, the testing organization will assume the responsibility for this development if the SSS Team provides them the necessary guidance and training mentioned above. Although this may seem prohibitive from a time standpoint, the software testers possess the detailed knowledge and experience necessary to design the most effective, efficient tests to help streamline the effort. However, even if the testers assume this responsibility, the SSS Team must review the test procedures and the test results to ensure their completeness and accuracy.

Test procedure validation requires that the analysts examine the procedures being established and verify that these procedures will completely test the SSRs or verify those potential failure modes or causal factors cannot result in a hazardous condition. The SSS Team must monitor the safety testing to both validate the procedures and make any adjustments necessary to spot anomalies that may have safety implications. This does not mean that a member of the SSS Team need be present for every test run. Often, this responsibility can be assigned to a member of the test team as long as they have the appropriate guidance and training. The software test team must annotate anomalies noted during the tests on the STR, as they relate to test procedures or test outcomes. With the proper training and guidance, they will be able to identify potential safety-related anomalies.

A majority of the anomalies will not be uncovered until the testing organization reduces the test data into a usable form. Data reduction involves extracting the data recorded during the tests; processing the data such that performance and other parameters can be derived; and presenting the information in a readable, understandable format. For example, in testing an Operational Flight Program (OFP) for an aircraft, the displays and data presented to the testers may appear valid. However, until that data is compared to that generated by a mathematical model, the

results are uncertain. Upon comparison, the testing organization may discover anomalies in the data (e.g., altimeter readout: errors of a few feet during landing can be disastrous yet appear acceptable in the laboratory). At this point, the software development organization must determine whether the anomalies are a result of errors in the test procedures, the test environment, the OFP, or in the mathematical model itself. If the errors are in the procedures or environment, the testing organization can make the appropriate changes and re-run the test. If the errors are in either the OFP or the mathematical model, the analysts must determine what the anomaly is and the necessary correction. In either case, the proposed changes must be approved by the CM team and submitted through the appropriate configuration control process (of which system safety is a part), including the analysis and testing regimen, before being implemented. The software testing organization performs regression testing and then repeats the test case(s) that failed. The correction of errors often unmasks more errors or introduces new ones. One study found that for every two errors found and corrected one was either unmasked or a new one introduced. Therefore, regression testing is an essential part of the overall testing process. It ensures that the modifications made do not adversely affect any other functionality or safety characteristics of the software. The SSS Team must participate in the development of the regression test series, at all levels, to ensure that SSRs are revalidated for each change.

Obviously, in a complex system, the above-described process will result in a substantial number of STRs and modifications to the software. Often, the software configuration control board will group several STRs affecting a single module or configuration item together and develop a one-time fix. As discussed in Appendix C-9, the SSS Team must be in the STR review process to ensure that the modifications do not adversely affect the safety of the software and to permit updating analyses as required.

In reviewing extracted data, the SSS Team must know those parameters that are safety-critical, and the values that may indicate a potential safety problem. Therefore, in the development of the data extraction, the software testers and SSS Team must work closely to ensure that the necessary data be extracted and that the data reduction programs will provide the necessary data in a readable format for use by the software safety and software testing groups. This is why the data analysis process discussed in Paragraph 4.3.7.3.2.2 is so important. If this analysis is done properly, all of the safety-critical data items will have been identified and are available for the software test team. If data analysis was not done, system safety will require interaction with domain experts to provide the necessary detailed knowledge of significant parameters. If the data extraction and reduction programs are written correctly with sufficient input from the SSS Team, the resulting printout will identify those parameters or combinations of parameters that represent a potentially hazardous condition. However, there is no substitute for analysis of the data since not all potentially hazardous conditions can be identified in advance.

Part of the data extraction may be needed to monitor conditions that cause the software to execute specific paths through the program (such as NO-GO paths). Often, these parameters are difficult or impossible to extract unless the program contains code specially designed to output that data (write statements, test stubs, breaks) or unless the test environment allows the test group to obtain a snapshot of the computer state (generally limited to monitoring computer registers). Unfortunately, test stubs often change the conditions in the executing software and may result in other errors being created or masked. Therefore, their use should be minimized. If used, the

software testing organization must perform regression testing after these stubs have been removed to ensure that errors have not been masked, additional errors introduced by their removal, or timing errors created by their removal.

The purpose of data extraction and analysis is to identify safety-related anomalies and subsequently the causal factors. The causal factors may be errors in the code, design, implementation, test cases, procedures, and/or test environment. If the casual factor cannot be identified as a procedural test error, test case error, or test environment error, the software safety analyst must analyze the source code to identify the root causes of the anomaly. This is the most common reason for performing code-level analysis. As described in Paragraph 4.3.7, once the causal factor is identified, the analyst develops a recommended correction, in coordination with the software developer, and presents it via the STR process to the software configuration control board. The analyst also needs to determine whether specific generic guidelines and requirements have been adhered to.

Occasionally, the software safety analyst will identify additional SSRs that must be incorporated into the system or software design. These are submitted as recommended corrective actions through the STR process. The SSS Team must perform a preliminary assessment of the risk associated with the new requirement and present it as part of a trade-off study to the Systems Engineering IPT. The SSS Team will write up the recommended addition as an ECP and, if approved, will undergo the same process for analyzing and testing the modification as for an STR.

SSRs cannot always be validated through testing. Often, this will show up as a failure in the test for a variety of reasons or a no-test. For example, limitations on the capabilities of simulators, stimulators, or the laboratory environment may preclude passing or completing certain tests. The SSS Team will need to make an engineering judgment as to whether the testing that has been completed is adequate. Additional tests may be required to provide sufficient assurance that the function is safe or to provide the desired level of safety assurance.

Throughout the testing process, the SSS Team will interact closely with the software testing organization to ensure that safety requirements are addressed at all levels. Together, the groups will assess the results of the testing performed and begin developing the safety test report. The report must identify the tests performed and the results of the analysis of the tests. References to test reports from the software testing group, STRs, ECPs, and anomalies detected and corrected should all be included in the test report. At the conclusion of the testing, the SSS Team uses the results to update the RTM in the SRCA and the various preliminary and detailed analyses performed on the system and its software. The software safety test report will be appended to the system safety test report and will form a part of the basis for the final system SAR.

4.4.3 Software Standards and Criteria Assessment

This paragraph provides guidance to the SSS Team to verify that software is developed in accordance with applicable safety-related standards and criteria. The assessment (Figure 4-42) begins very early in the development process as design requirements are tailored and implemented into system-level and top tier software specifications and continues through the

analysis of test results and various reports from other IPTs. Ultimately, the assessment becomes an integral part of the overall SAR.

Standards and criteria include those extracted from the generic documents such as STANAG 4404 and military, federal, and industry standards and handbooks; lessons learned; safety programs on similar systems; internal company documents; and other sources. Verification that the software is developed in accordance with syntactic restrictions and applicable software engineering standards and criteria is largely a function that the SSS Team can delegate to the SQA, Software Configuration Management (SCM), and Software Testing (including the verification and validation) teams. This is especially true with many of the generic software engineering requirements from STANAG 4404, IEEE Standard 1498, and other related documents. The SQA and SCM processes include these requirements as a routine part of their normal compliance assessment process. The software developers and testers will test generic or system-specific safety test requirements as a normal part of the software testing process. System safety must review test cases and test procedures and make recommendations for additional or modified procedures and additional tests to ensure complete coverage of applicable requirements. However, review of test cases and procedures alone may not be sufficient. A number of the generic requirements call for the safety analyst to ensure that the code meets their intent versus the letter of the safety requirement. As noted earlier, due to the ambiguous nature of the English language, specifications and safety requirements may be interpreted differently by the software developer thus not meeting the intent of the requirement. In some instances, this requires an examination of the source code (see Paragraph 4.3.7).

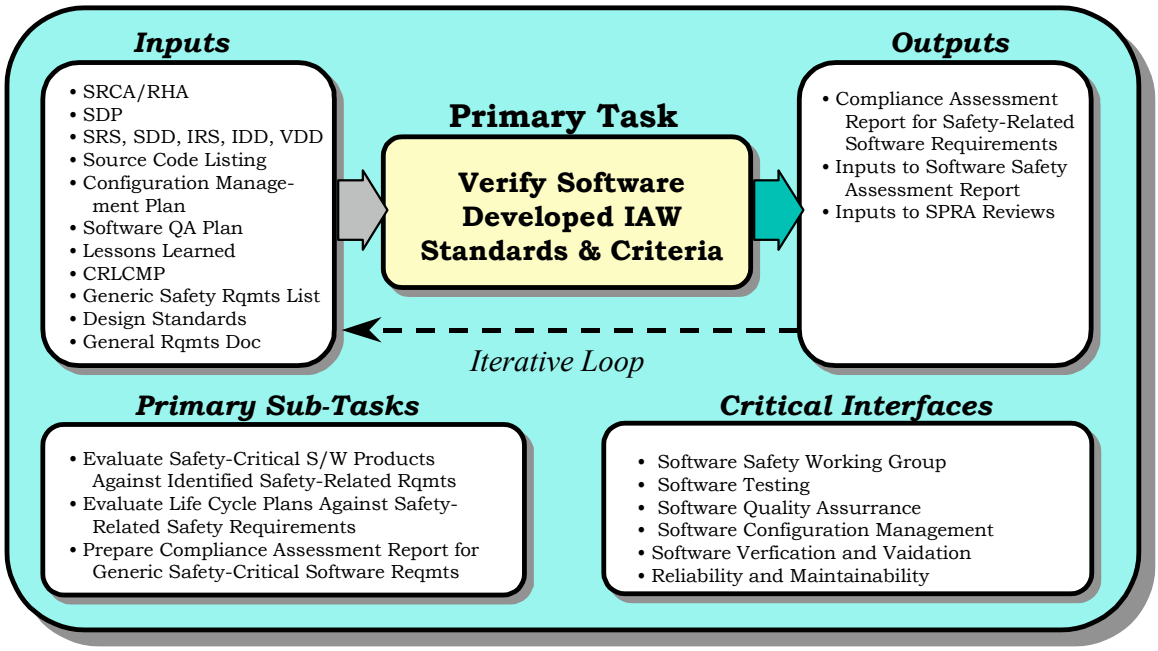


Figure 4-42: Software Requirements Verification

The generic software development requirements and guidelines are provided to the SQA team for incorporation into their assessment process, plans, and reviews. Although many of the specific test requirements may be assigned to the individual software development teams for unit and

integration testing, the software testing team generally assumes responsibility for incorporating generic test requirements into their test planning process. The CM Team is responsible for the integration of requirements related to CM into their plans and processes that include participation by safety. The latter include the participation by safety in the CM process.

The SSS Team reviews the assessment performed by the SQA team, incorporating the results for the safety-related criteria into the final safety assessment. This assessment should occur on a real-time basis using representatives from the SSS Team (or a member of the SQA team assigned responsibility for software safety). In order to possess the degree of compliance, these representatives should participate in code reviews, walk-through processes, peer reviews, and the review of the software development process. The assessment should include both the degrees of compliance or the rationale for non-compliance with each of the criteria.

Software safety participates on a real-time basis with the CM process. Therefore, the assessment against applicable criteria can occur during this process. To a large extent, the degree of compliance depends on the degree of involvement of system safety in the CM process and their thoroughness in fulfilling their roles.

The Compliance Assessment Report, as its name implies, is a compilation of the above compliance assessments with a final assessment as to whether or not the system satisfies the applicable safety requirements. The compliance assessment is simply a portion of the overall safety assessment process used to ascertain the residual risk associated with the system.

4.4.4 Software Safety Residual Risk Assessment

The safety risk assessment of software is not as straightforward a process as it is for hardware. The hardware risk assessment relies on the severity of the hazards and the probabilities, whether qualitative or quantitative, coupled with the remaining conditions required resulting in a hazardous condition or an accident. Engineering judgment and experience with similar systems provide the basis for qualitative probabilities while statistical measurements, such as reliability predictions, provide the basis for quantitative probabilities. The safety analyst uses this information to compile a probability of a hazard occurring over the life of the system (hence the residual risk associated with the system). Coupled with estimates of the likelihood of satisfying the remaining conditions that result in an accident or mishap, an estimate of the risk results. Unfortunately, reliability metrics for software are often meaningless; therefore, qualitative risk assessment must be applied. The latter is based on an assessment by the analyst that sufficient analysis and testing have been performed. This means sufficient analysis to identify the hazards, develop and incorporate safety requirements into the design, analyze SSR implementation including sufficient testing (and analysis of test results) to provide a reasonable degree of assurance that the software will have a sufficiently low level of risk. The software safety assessment begins early in the system development process, largely starting with the compliance assessment of the safety requirements discussed previously. However, the assessment cannot be completed until system-level testing in an operational environment is complete. This includes operational test and evaluation, and the analyses that conclude that all identified hazards have been resolved. The software safety assessment process, illustrated in Figure 4-43, is generally complete when it is integrated with the SAR.

As described in Paragraph 4.3.5, the analyst identifies the software causal factors early in the analytical phase and assigns a hazard severity and software control category (e.g., SHRI) to each. The result is a software HRI for that causal factor. However, this is not a measure of the safety risk but an indication of the potential programmatic risk associated with the software. It also provides an indication of the degree of assurance required in the assessment of the software to ensure that it will execute safely in the system context. It provides guidance on the amount of analysis and testing required verifying and validating the software associated with that function or causal factor. The SHRI does not change unless the design is modified to reduce the degree of control that the software exercises over the potentially hazardous function. However, analysis and testing performed on the software reduce the actual risk associated with it in the system application. In this manner, a qualitative HRI may be assigned to the specific function based on engineering judgment. The SSS Team needs to document these engineering judgments made for that function and its associated hazard(s) within the hazard tracking database.

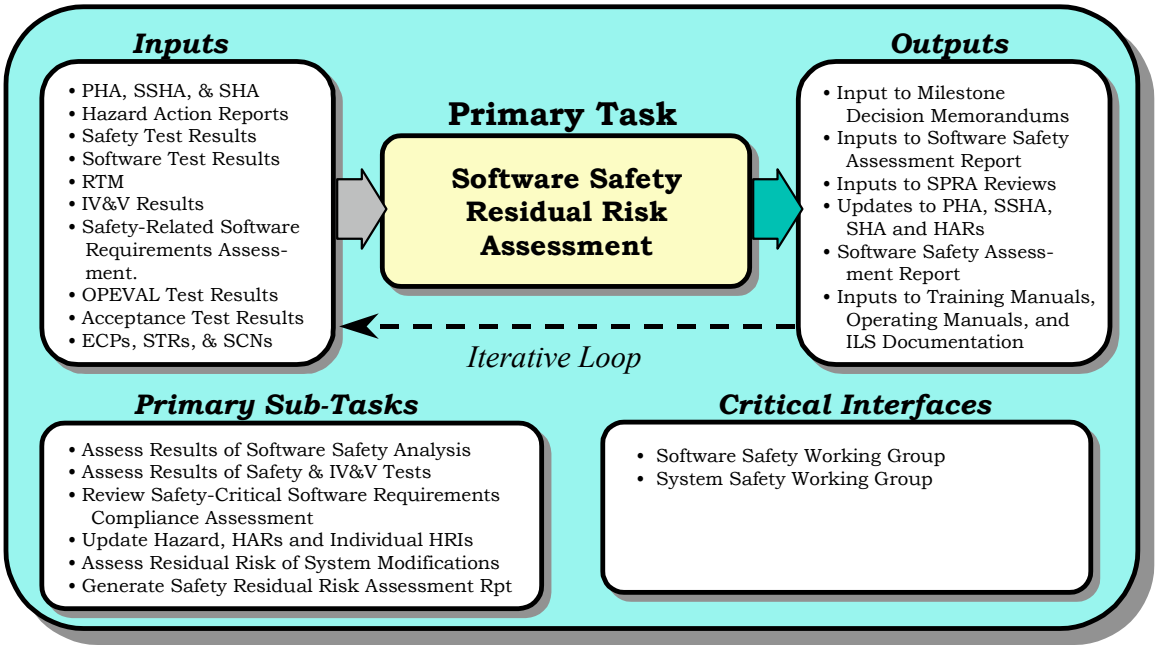


Figure 4-43: Residual Safety Risk Assessment

As with any hazard analysis, closure of the hazard requires that the analyst review the results of the analyses performed and the tests conducted at both a component and system level. Closure of hazards occurs on a real-time basis as the design progresses. The SSS Team analyzes the design and implementation of the functions, both safety-critical and safety-related, and determines whether it meets the intent of the SSR. It also determines whether the implementation (hardware and/or software) provides sufficient interlocks, checks, and balances to ensure that the function will not contribute to a hazardous condition. Coupled with the results of testing performed on these functions, the analyst uses his or her best judgment as to whether the risk is sufficiently mitigated.

In performing the assessment of safety and verification and validation testing, the software safety analyst must examine a variety of metrics associated with testing. These include path coverage,

overall coverage of the program, and usage-based testing. In general, testing that is limited to the usage base is inadequate for safety. Safety-critical modules are often those that execute only when an anomaly occurs. This results in a very low predicted usage, and consequently, usage-based testing performs very little testing on those functions. The result is that anomalies may be present and undetected. The SSS Team should assess the degree of coverage and determine its adequacy. Generally, if the software testers are aware of the need for additional test coverage of safety-critical functions, they will be incorporated into the routine testing.

The safety program must subject new requirements identified during the analysis and testing phases to the same level of rigor as those in the original design. However, the SSS Team must pay particular attention to these areas since they are the areas most likely to contain errors in the latter stages of development. This is more a function of introducing requirements late, and reducing the time available for analysis and testing. In addition, the potential interactions with other portions of the system interfaces may be unknown and may not receive the same degree of attention (especially at the integration testing level) as the original requirements.

The SSS Team must keep in mind throughout the safety assessment process, the ultimate definition of acceptable risk as defined by the customer. Where unacceptable or undesirable risks are identified, the SSS Team, in coordination with the SSWG, must provide the rationale for recommending to the customer and/or the Safety Review Authority (SRA) acceptance of that risk. Even for systems which comply with the level of risk defined by the customer's requirements, the rationale for that assessment and the supporting data must be provided. This material is also documented in the SAR.

The SAR contains a summary of the analyses performed and their results, the tests conducted and their results, and the compliance assessment described in Paragraph 4.4.3.

4.5 Safety Assessment Report

The SAR is generally a CDRL item for the safety analysis performed on a given system. The purpose of the report is to provide management an overall assessment of the risk associated with the system including the software executing within the system context of an operational environment. This is accomplished by providing detailed analysis and testing evidence, that all of the software related hazards have been identified and have been either eliminated or mitigated/controlled to levels acceptable to the AE, PM and PFS/Safety Manager. It is paramount that this assessment report be developed as an encapsulation of all of the analysis performed as a result of the recommendations provided in the previous sections.

The SAR shall contain a summary of the analyses performed and their results, the tests conducted and their results, and the compliance assessment. Paragraph 4.5.1 is a sample outline for the SAR. Paragraphs within the SAR need to encompass the following items:

- The safety criteria and methodology used to classify and rank software related hazards (causal factors). This includes any assumptions made from which the criteria and methodologies were derived;
- The results of the analyses and testing performed;

- The hazards that have an identified residual risk and the assessment of that risk;
- The list of significant hazards and the specific safety recommendations or precautions required to reduce their safety risk; and
- A discussion of the engineering decisions made that affect the residual risk at a system level.

The final section of the SAR should be a statement by the PFS/Safety Manager describing the overall risk associated with the software in the system context and their acceptance of that risk.

4.5.1 Safety Assessment Report Table of Contents

1. ***Introduction (i.e., Purpose, Scope, and Background)***
2. ***System Overview/Concept of Operations***
   - ♦ Provide a High-level Hardware Architecture Overview
   - ♦ Describe System, Subsystem, and Interface Functionality
   - ♦ Provide Software Architecture Detailed Description
   - ♦ Describe Architecture/Processors/Coding Language Implemented
   - ♦ Describe CSCIs/CSUs and Their Functional Interfaces
   - ♦ Identify Safety-Critical Functionality & Messages/Data
3. ***System Description***
4. ***Hazard Analysis Methodology***
   - ♦ Describe Hazard-Based Approach and Structure of the Hazard Tracking Database
   - ♦ Describe SSR Identification Process (Initial RTM Development)
   - ♦ Define Hazard Assessment Approach (i.e., HRI and SHRI Matrices)
   - ♦ Define Tools, Methods, and Techniques Used in the Software Safety Analyses
   - ♦ Include Table With PHA Level Hazards & HRIs
   - ♦ Identify Safety-critical Functions That Are Software Related and Indicate the Hazards That They Affect
5. ***Document Hazard Analysis Results***
   - ♦ Provide Detailed Records of System-Level Hazards
   - ♦ Provide Detailed Records of Subsystem-Level Hazards
   - ♦ Provide In-depth Evidence of Hazard Causes to the Level Required to Mitigate or Control the Hazards Effectively, Efficiently, and Economically.

- Identify Hazards That Have Software Influence or Causes (i.e., Software Causal Factors in the Functional, Physical, or Process Context of the Hazard.)

6. Identify Hazard Elimination or Mitigation/Control Requirements

- Describe Sources of SSRs (i.e., safety design requirements, Generics, Functionally Derived and Hazard Control)
- Identify the Initial SSRs for the System
- Provide Evidence of SSR Traceability (RTM and Hazard Tracking Database Updates)
- Identify Functionally Derived SSRs Based on Detailed Hazard Cause Analysis.

7. Provide Evidence of SSR Implementation in Design

- Describe SSR Verification Process
- Describe SSR Analysis, Testing and Test Results Analysis
- Provide Evidence of SSR Verification (RTM and Hazard Tracking Database Updates)

8. Provide a final Software Safety Assessment

- Provide an Assessment of Risk Associated with Each hazard in Regards to Software.
- Identify Any Remaining Open Issues/Concerns

9. Appendices

- SRCA (Include RTM, SSR Verification Trees, and SSR Test Results)
- Hazard Tracking Database Worksheets
- Any FTA Analysis Reports Generated to Identify Software Effects on Hazards

A. DEFINITION OF TERMS

A.1 ACRONYMS

AE	-	Acquisition Executive
AECL	-	Atomic Energy of Canada Limited
AFTI	-	Advanced Fighter Technology Integration
ALARP	-	As Low As Reasonably Possible
ARP	-	Aerospace Recommended Practice
ARTE	-	Ada Runtime Environment
CAE	-	Component Acquisition Executive
CASE	-	Computer-Aided Software Engineering
CCB	-	Configuration Control Board
CDR	-	Critical Design Review
CDRL	-	Contract Deliverable Requirements List
CHI	-	Computer/Human Interface
CI	-	Configuration Item
CM	-	Configuration Management
COTS	-	Commercial-Off-The-Shelf
CPAF	-	Cost-Plus-Award-Fee
CPU	-	Central Processing Unit
CRC	-	Cyclic Redundancy Check
CRISD	-	Computer Resource Integrated Support Document
CRWG	-	Computer Resource Working Group
CSCI	-	Computer Software Configuration Item
CSU	-	Computer Software Unit
CSR	-	Component Safety Requirement
CTA	-	Critical Task Analysis
DA	-	Developing Agency
DAD	-	Defense Acquisition Deskbook
DAL	-	Development Assurance Level
DDA	-	Detailed Design Analysis
DFD	-	Data Flow Diagram
DID	-	Data Item Description
DOD	-	Department of Defense
DOT	-	Department of Transportation
DSMC	-	Defense Systems Management College
DU	-	Depleted Uranium
ECP	-	Engineering Change Proposal
E/E/PES	-	Electrical/Electronic/Programmable Electronic Systems
EIA	-	Electronic Industries Association
EMP	-	Electro-Magnetic Pulse
EOD	-	Explosive Ordnance Disposal
ESH	-	Environmental Safety and Health
FAA	-	Federal Aviation Administration
FCA	-	Functional Configuration Audit
FFD	-	Functional Flow Diagram

FQT	-	Functional Qualification Test
FTA	-	Fault Tree Analysis
GOTS	-	Government-Off-The-Shelf
GSSRL	-	Generic Software Safety Requirements List
HHA	-	Health Hazard Assessment
HMI	-	Human/Machine Interface
HRI	-	Hazard Risk Index
HAR	-	Hazard Action Record
ICD	-	Interface Control Document
ICWG	-	Interface Control Working Group
IDS	-	Interface Design Specification
IEC	-	International Electotechnical Commission
IEEE	-	Institute of Electrical and Electronic Engineering
ILS	-	Integrated Logistics Support
IPD	-	Integrated Product Development
IPT	-	Integrated Product Team
IV&V	-	Independent Verification & Validation
LOT	-	Level of Trust
MA	-	Managing Authority
MAA	-	Mission Area Analysis
MAIS	-	Major Automated Information System
MAPP	-	Major Acquisition Policies and Procedures
MDAP	-	Major Defense Acquisition Programs
MIL-STD	-	Military Standard
NASA	-	National Aeronautics and Space Administration
NDI	-	Non-Developmental Item
NSS	-	NASA Safety Standard
O&SHA	-	Operating and Support Hazard Analysis
OS	-	Operating System
PA	-	Procuring Authority
PCA	-	Physical Configuration Audit
PDL	-	Program Design Language
PDR	-	Preliminary Design Review
PEO	-	Program Executive Officer
PFD	-	Process Flow Diagram
PFS	-	Principal for Safety
PHA	-	Preliminary Hazard Analysis
PHL	-	Preliminary Hazard List
PM	-	Program Manager
PMR	-	Program Management Review
POA&M	-	Plan of Actions & Milestones
PTR	-	Program Trouble Report
QA	-	Quality Assurance
QAP	-	Quality Assurance Plan
QC	-	Quality Control

REP	-	Reliability Engineering Plan
RFP	-	Request for Proposal
RMP	-	Risk Management Plan
ROM	-	Read Only Memory
RTCA	-	RTCA, Inc.
RTM	-	Requirements Traceability Matrix
SAF	-	Software Analysis Folder
SAR	-	Safety Assessment Report
SCCSF	-	Safety-Critical Computing System Functions
SCFL	-	Safety-Critical Functions List
SCM	-	Software Configuration Management
SCN	-	Software Change Notice
SDL	-	Safety Data Library
SDP	-	Software Development Plan
SDR	-	System Design Review
SEDS	-	Systems Engineering Detailed Schedule
SEE	-	Software Engineering Environment
SEMP	-	Systems Engineering Management Plan
SEMS	-	Systems Engineering Master Schedule
SHA	-	System Hazard Analysis
SHCM	-	Software Hazard Criticality Matrix
SHRI	-	Software Hazard Risk Index
SIL	-	Safety Integrity Level
SIP	-	Software Installation Plan
SON	-	Statement of Operational Need
SOO	-	Statement of Objectives
SOW	-	Statement of Work
SPRA	-	Safety Program Review Authority
SQA	-	Software Quality Assurance
SRA	-	Safety Review Authority
SRCA	-	Safety Requirements Criteria Analysis
SRS	-	Software Requirements Specifications
SSE	-	Software Safety Engineer
SSG	-	System Safety Group
SSHA	-	Subsystem Hazard Analysis
SSMP	-	System Safety Management Plan
SSP	-	System Safety Program
SSPP	-	System Safety Program Plan
SSR	-	Software Safety Requirements
SSS	-	Software System Safety
SSSH	-	Software System Safety Handbook
SSWG	-	System Safety Working Group
STP	-	Software Test Plan
STR	-	Software Trouble Report
STSC	-	Software Technology Support Center
SwSE	-	Software Safety Engineer
SwSPP	-	Software Safety Program Plan
SwSSP	-	Software System Safety Program
SwSSWG	-	Software System Safety Working Group
TEMP	-	Test and Evaluation Master Plan

TRR	-	Test Readiness Review
TWG	-	Test Working Group
WBS	-	Work Breakdown Structure

A.2 DEFINITIONS

NOTE: All definitions used in this handbook have either been extracted from MIL-STD-882C or MIL-STD-498. A [882] or [498] references each definition. While many may disagree with aspects of, or the entire definition in total, these were used because of their common use on DOD programs.

Acceptance. An action by an authorized representative of the acquirer by which the acquirer assumes ownership of software products as partial or complete performance of a contract. [498]

Acquiring Agency. An organization that produces software products for itself or another organization. [498]

Architecture. The organizational structure of a system or CSCI, identifying its components, their interfaces, and concept of execution among them. [498]

Behavioral Design. The design of how an overall system or CSCI will behave, from a user's point of view, in meeting its requirements, ignoring the internal implementation of the system or CSCI. This design contrasts with architectural design, which identifies the internal components of the system or CSCI, and with the detailed design of those components. [498]

Build. (1) A version of software that meets a specified subset of the requirements that the completed software will meet. (2) The period of time during which such a version is developed. [498]

Computer Hardware. Devices capable of accepting and storing computer data, executing a systematic sequence of operations on computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions. [498]

Computer Program. A combination of computer instructions and data definitions that enables computer hardware to perform computational or control functions. [498]

Computer Software Configuration Item. An aggregation of software that satisfies an end-use function and is designated for separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, developer, support concept, plans or reuse, criticality, interface considerations, need to be separately documented and controlled, and other factors. [498]

Condition. An existing or potential state such as exposure to harm, toxicity, energy source, activity, etc. [882]

Configuration Item. An aggregation of hardware, software, or both that satisfies an end use function and is designated for separate configuration management by the acquirer. [498]

Contractor. A private sector enterprise or the organizational element of DOD or any other government agency engaged to provide services or products within agreed limits specified by the MA. [882]

Data Type. A class of data characterized by the members of the class and operations that can be applied to them; for example, integer, real, or logical. [IEEE 729-1983]

Deliverable Software Product. A software product that is required by the contract to be delivered to the acquirer or other designated recipient. [498]

Design. Those characteristics of a system or CSCI that are selected by the developer in response to the requirements. Some will match the requirements; others will be elaborations of requirements, such as definitions of all error messages; others will be implementation related, such as decisions, about what software units and logic to use to satisfy the requirements. [498]

Fail Safe. A design feature that ensures that the system remains safe or in the event of a failure will cause the system to revert to a state which will not cause a mishap. [882]

Firmware. The combination of a hardware device and computer instructions and/or computer data that reside as read-only software on the hardware device. [498]

Hazard. A condition that is a prerequisite to a mishap. [882]

Hazard Probability. The aggregate probability of occurrence of the individual events that create a specific hazard. [882]

Hazard Severity. An assessment of the consequences of the worst credible mishap that could be caused by a specific hazard. [882]

Independent Verification & Validation. Systematic evaluation of software products and activities by an agency that is not responsible for developing the product or performing the activity being evaluated. [498]

Managing Activity. The organizational element of DOD assigned acquisition management responsibility for the system, or prime or associate contractors or subcontractors who impose system safety tasks on their suppliers. [882]

Mishap. An unplanned event or series of events resulting in death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment. An accident. [882]

Process. An organized set of activities performed for a given purpose. [498]

Qualification Test. Testing performed to demonstrate to the acquirer that a CSCI or a system meets its specified requirements. [498]

Reengineering. The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering (analyzing a system and producing a representation at a higher level of abstraction, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), redocumentation (analyzing a system and producing user or support documentation), forward engineering (using software products derived from an existing system, together with new requirements, to produce a new system), retargeting (transforming a system to install it on a different target system), and

translation (transforming source code from one language to another, or from one version of a language to another). [498]

Requirement. (1) A characteristic that a system or CSCI must possess in order to be acceptable to the acquirer. (2) A mandatory statement in contractual binding document (i.e., standard, or contract). [498]

Reusable Software Products. A software product developed for one use but having other uses, or one developed specifically to be usable on multiple projects or in multiple roles on one project. Examples include, but are not limited to, commercial-off-the-shelf software products, acquirer-furnished software product, software products in reuse libraries, and pre-existing developer software products. Each use may include all or part of the software product and may involve its modification. [498]

Risk. An expression of the possibility/impact of a mishap in terms of hazard severity and hazard probability. [882]

Risk Assessment. A comprehensive evaluation of the risk and its associated impact. [882]

Safety. Freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment. [882]

Safety-Critical. A term applied to a condition, event, operation, process, or item of whose proper recognition, control, performance or tolerance is essential to safe system operation or use, e.g., safety-critical function, safety-critical path, safety-critical component. [882]

Safety-Critical Computer Software Components. Those computer software components and units whose errors can result in a potential hazard, or loss of predictability or control of a system. [882]

Software Development. A set of activities that results in software products. Software development may include new development, modification, reuse, reengineering, maintenance, or any other activities that result in software products. [498]

Software Engineering. In general usage, a synonym for software development. As used in MIL-STD 498, a subset of software development consisting of all activities except qualification testing. [498]

Software System. A system consisting solely of software and possibly the computer equipment on which the software resides and operates. [498]

Subsystem. An element of a system that, in itself may constitute a system. [882]

System. A composite, at any level of complexity, of personnel, procedures, materials, tools, equipment, facilities, and software. The elements of this composite entity are used together in the intended operational or support environment to perform a given task or achieve a specific purpose, support, or mission requirement. [882]

System Safety. The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle. [882]

System Safety Engineer. An engineer who is qualified by training and/or experience to perform system safety engineering tasks. [882]

System Safety Engineering. An engineering discipline requiring specialized professional knowledge and skills in applying scientific and engineering principles, criteria, and techniques to identify and eliminate hazards, in order to reduce the associated risk. [882]

System Safety Group/Working Group. A formally chartered group of persons, representing organizations initiated during the system acquisition program, organized to assist the MA system PM in achieving the system safety objectives. Regulations of the military components define requirements, responsibilities, and memberships. [882]

System Safety Management. A management discipline that defines SSP requirements and ensures the planning, implementation and accomplishment of system safety tasks and activities consistent with the overall program requirements. [882]

System Safety Manager. A person responsible to program management for setting up and managing the SSP. [882]

System Safety Program. The combined tasks and activities of system safety management and system safety engineering implemented by acquisition project managers. [882]

System Safety Program Plan. A description of the planned tasks and activities to be used by the contractor to implement the required SSP. This description includes organizational responsibilities, resources, methods of accomplishment, milestones, depth of effort, and integration with other program engineering and management activities and related systems. [882]

B. REFERENCES

B.1 GOVERNMENT REFERENCES

DODD 5000.1, Defense Acquisition, March 15, 1996

DOD 5000.2R, Mandatory Procedures for Major Defense Acquisition Programs and Major Automated Information Systems, March 15, 1996

DOD-STD 2167A, Military Standard Defense System Software Development, February 29, 1988

MIL-STD 882B, System Safety Program Requirements, March 30, 1984

MIL-STD 882C, System Safety Program Requirements, January 19, 1993

MIL-STD 498, Software Development and Documentation, December 5, 1994

FAA Order 1810, Acquisition Policy

FAA Order 8000.70, FAA System Safety Program

RTCA-DO 178B, Software Considerations In Airborne Systems And Equipment Certification, December 1, 1992

COMDTINST M411502D, System Acquisition Manual, December 27, 1994

NSS 1740.13, Interim Software Safety Standard, June 1994

Department of the Air Force, Software Technology Support Center, Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems, Version-2, June 1996, Volumes 1 and 2

AFISC SSH 1-1, Software System Safety Handbook, September 5, 1985

B.2 COMMERICIAL REFERENCES

EIA-6B, G-48, Electronic Industries Association, System Safety Engineering In Software Development1990

IEC 61508: International Electrotechnical Commission. Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-Related Systems, December 1997.

EIC 1508 -(Draft), International Electrotechnical Commission, Functional Safety; Safety-Related System, June 1995

IEEE STD 1228, Institute of Electrical and Electronics Engineers, Inc., Standard For Software Safety Plans, 1994

IEEE STD 829, Institute of Electrical and Electronics Engineers, Inc., Standard for Software Test Documentation, 1983

IEEE STD 830, Institute of Electrical and Electronics Engineers, Inc., Guide to Software Requirements Specification, 1984

IEEE STD 1012, Institute of Electrical and Electronics Engineers, Inc., Standard for Software Verification and Validation Plans, 1987

ISO 12207-1, International Standards Organization, Information Technology-Software, 1994

Society of Automotive Engineers, Aerospace Recommended Practice 4754: Certification Considerations for Highly Integrated or Complex Aircraft Systems, November 1996.

Society of Automotive Engineers, Aerospace Recommended Practice 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

B.3 INDIVIDUAL REFERENCES

Brown, Michael, L., Software Systems Safety and Human Error, Proceedings: COMPASS 1988

Brown, Michael, L., What is Software Safety and Who's Fault Is It Anyway?, Proceedings: COMPASS 1987

Brown, Michael, L., Applications of Commercially Developed Software in Safety Critical Systems, Proceedings of Parari '99, November 1999

Bozarth, John D., Software Safety Requirement Derivation and Verification, Hazard Prevention, Q1, 1998

Bozarth, John D., The MK 53 Decoy Launching System: A "Hazard-Based" Analysis Success, Proceedings: Parari '99, Canberra, Australia

Card, D.N. and **Schultz**, D.J., Implementing a Software Safety Program, Proceedings: COMPASS 1987

Church, Richard, P., Proving A Safe Software System Using A Software Object Model, Proceedings:15th International System Safety Society Conference, 1997

Connolly, Brian, Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example, Proceedings: COMPASS 1989

De Santo, Bob, A Methodology for Analyzing Avionics Software Safety, Proceedings: COMPASS 1988

Dunn, Robert and **Ullman,** Richard, Quality Assurance For Computer Software, McGraw Hill, 1982

Ericson, C.A., Anatomy of a Software Hazard, Briefing Slides, Boeing Computer Services, June 1983

Foley, Frank, History and Lessons Learned on the Northrop-Grumman B-2 Software Safety Program, Paper, Northrop-Grumman Military Aircraft Systems Division, 1996

Forrest, Maurice, and **McGoldrick,** Brendan, Realistic Attributes of Various Software Safety Methodologies, Proceedings: Ninth International System Safety Society, 1989

Gill, Janet A.,Safety Analysis of Heterogeneous-Multiprocessor Control System Software, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

Hammer, William, R., Identifying Hazards In Weapon Systems – The Checklist Approach, Proceedings: Parari '97, Canberra, Australia

Kjos, Kathrin, Development of an Expert System for System Safety Analysis, Proceedings: Eighth International System Safety Conference, Volume II.

Lawrence, J.D., Design Factors for Safety-Critical Software, NUREG/CR-6294, Lawrence Livermore National Laboratory, November 1994

Lawrence, J.D., Survey of Industry Methods for Producing Highly Reliable Software, NUREG/CR-6278, Lawrence Livermore National Laboratory, November 1994.

Leveson, Nancy, G, SAFEWARE; System Safety and Computers, A Guide to Preventing Accidents and Losses Caused By Technology, Addison Wesley, 1995

Leveson, Nancy, G., Software Safety: Why, What, and How, Computing Surveys, Vol 18, No. 2, June 1986

Littlewood, Bev and **Strigini**, Lorenzo, The Risks of Software, Scientific American, November 1992

Mattern, S.F., Software Safety, Masters Thesis, Webster University, St. Louis, MO. 1988

Mattern, S.F. Capt., Defining Software Requirements for Safety-Critical Functions, Proceedings: Twelfth International System Safety Conference, 1994

Mills, Harland, D., Engineering Discipline For Software Procurement, Proceedings: COMPASS 1987

Moriarty, Brian and **Roland**, Harold, E., System Safety Engineering and Management, Second Edition, John Wiley & Sons, 1990

Russo, Leonard, Identification, Integration, and Tracking of Software System Safety Requirements, Proceedings: Twelfth International System Safety Conference, 1994

Unknown Author: Briefing on the Vertical Launch ASROC (VLA), Minutes, 2nd Software System Safety Working Group (SwSSWG), March 1984

B.4 OTHER REFERENCES

DEF(AUST) 5679, Army Standardization (ASA), The Procurement Of Computer-Based Safety Critical Systems, May 1999

UK Ministry of Defence. Interim DEF STAN 00-54: Requirements for Safety Related Electronic Hardware in Defence Equipment, April 1999.

UK Ministry of Defence. Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment, Issue 2, 1997

UK Ministry of Defence. Defence Standard 00-56: Safety Management Requirements for Defence Systems, Issue 2, 1996

International Electrotechnical Commission, IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, draft 61508-2 Ed 1.0., 1998

Document ID: CA38809-101, International Standards Survey and Comparison to Def(Aust) 5679, Issue: 1.1, Dated 12 May 1999

C. HANDBOOK SUPPLEMENTAL INFORMATION

C.1 PROPOSED CONTENTS OF THE SYSTEM SAFETY DATA LIBRARY

C.1.1 SYSTEM SAFETY PROGRAM PLAN

The SSPP is a requirement of MIL-STD-882 for DOD procurements. It describes in detail, tasks and activities of the system safety engineering and management program established by the supplier. It also describes the engineering processes to be employed to identify, document, evaluate, and eliminate and/or control system hazards to the levels of acceptable risk for the program. The approved plan (by the customer) provides the formal basis of understanding and agreement between the supplier and the customer on how the program will be executed to meet contractual requirements. Specific provisions of the SSPP include program scope and objectives, system safety organization and program interfaces, program milestones, general safety requirements and provisions; and specific hazard analyses to be performed. It must explain in detail the methods and processes to be employed on the program to identify hazards and failure modes, derive design requirement to eliminate or control the hazard, and the test requirements and verifications methods to be used to ensure that hazards are controlled to acceptable levels. *Even if the contract does not require a SSPP, the safety manager of the development agency should produce this document to reduce interface and safety process misconceptions as it applies to the design and test groups.* An excellent template for the SSPP format is DID DI-SAFT-80100, or MIL-STD-882C, January 1993.

Specifically, the plan must describe:

- General description of the program,
- System safety organization,
- SSP milestones,
- SSP requirements,
- Hazard analyses to be performed,
- Hazard analysis processes to be implemented,
- Hazard analyses data to be obtained,
- Method of safety verification,
- Training Requirements,
- Applicable audit methods,
- Mishap prevention, reporting, and investigation methods, and

- System safety interfaces.

If the SSPP is not accomplished, numerous problems can surface. Most importantly, a simple "roadmap" of hazard identification, mitigation, elimination, and risk reduction effort is not presented for the program. The result is the reduction of programmatic and technical support and resource allocation to the SSP. The SSPP is not simply a map of task descriptions of what the safety engineer is to accomplish on a program, but an integration of the safety engineering across critical management and technical interfaces. It allows all integrated program team members to assess and agree to the necessary support required by all technical disciplines of the development team to support the safety program. Without this support, the safety design effort will fail.

C.1.2 SOFTWARE SAFETY PROGRAM PLAN

Some contractual deliverable obligations will include the development of a SwSPP. This was a common requirement in the mid-1980s through the early-1990s. The original intent was to formally document that a development program (that was software-intensive, or that had software performing safety-critical functions) considered in detail, the processes, tasks, interfaces, and methods to ensure software was taken into consideration from a safety perspective. In addition, the intent was to ensure that steps were taken in the design, code, test, and IV&V activities to minimize, eliminate and/or control the safety risk to a predetermined level. At the same time, many within the safety and software communities mistakenly considered software safety to be a completely separate engineering task than the original system safety engineering activities.

Today, it is recognized that software must be rendered safe (to the greatest extent possible equating to the lowest safety risk) through a "systems" methodology. It has been determined, and agreed upon, that the ramifications of software performing safety-critical functions can be assessed and controlled through sound system safety engineering and software engineering techniques. In fact, without the identification of specific software-caused, or software-influenced, system hazards or failure modes, the assessment of residual safety risk of software cannot be accomplished. Although software error identification, error trapping, fault tolerance, and/or error removal is essential in software development, without a direct tie to a system hazard or failure mode, the product is usually not safety related - it is reliability and system availability.

If possible, and in accordance with MIL-STD-882, specific tasks, processes, and activities associated with the elimination or minimization of software-specific safety risk should be integrated into the SSPP. If a customer is convinced that a separate program plan is required for software safety specifically, this too can be accomplished. It should contain a description of the safety, systems, and software engineering processes to be employed to identify, document, evaluate, and eliminate and/or control system hazards (that are software-caused or software-influenced) to the levels of acceptable risk for the program. It should describe program interfaces, lines of communication between technical and programmatic functions, and document the suspense and milestone activities according to an approved schedule. As a reminder, the software safety schedule of events should correspond with the software (and hardware) development life cycle, and be in concert with each development and test plan published by other technical disciplines associated with the program. IEEE STD 1228-1994, IEEE Standard For Software Safety Plans, provides an industry standard for the preparation and contents of a

SwSPP. An outline based upon this standard is presented in Figure C.1 and can be used as a guide for plan development.

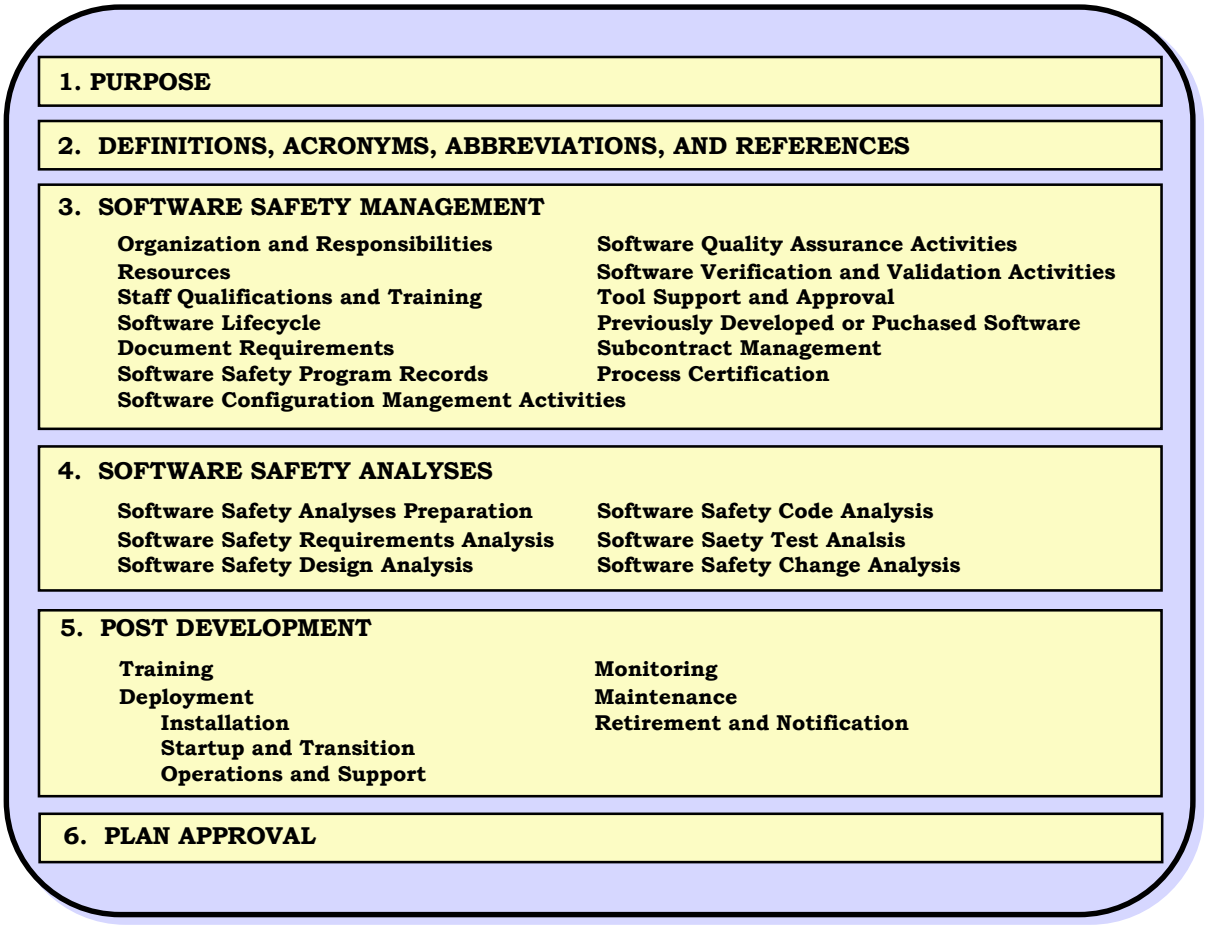


Figure C.1: Contents of a SwSPP - IEEE STD 1228-1994

C.1.3 PRELIMINARY HAZARD LIST

The purpose of a PHL is for the compilation of a list of preliminary hazards of the system as early in the development life cycle as possible. The source of information that assists the analyst in compiling a preliminary list is

- Similar system hazard analysis,
- Lessons learned,
- Trade study results,
- Preliminary requirements and specifications,
- Design requirements from design handbooks (i.e., AFSC, DH 1-6, System Safety),
- Potential hazards identified by safety team brainstorming,

- Generic SSRs and guidelines, and
- Common sense.

The list of preliminary hazards of the proposed system becomes the basis of the PHA and the consideration and development of design alternatives. The PHL must be used as inputs to proposed design alternatives and trade studies. As the design matures, the list is "scrubbed" to eliminate those hazards that are not applicable for the proposed system, and to document and categorize those hazards deemed to contain inherent (potential) safety risk.

Lessons learned information can be extracted from databases specifically established to document lessons learned in design, manufacture, fabrication, test, or operation activities, or actual mishap information from organizations such as the service safety agencies. Each hazard that is identified should be recorded on the list and contain its source of reference.

C.1.4 SAFETY-CRITICAL FUNCTIONS LIST

Although not identified in MIL-STD-882, the introduction of a Safety-Critical Functions List (SCFL) historically became more important as specific process steps to perform software safety tasks became more mature and defined. The design, code, test, IV&V, implementation, and support of software code can become expensive and resource limited. Software code that performs safety-critical functions requires an extensive protocol, or level of effort, within the design, code, test, and support activities. This added structure, discipline, and level of effort, add cost to the program and should be performed first on those modules of code that are most critical from a safety perspective. Conversely, code developed with no identified inherent safety risk, would require a lesser level of design, test, and verification protocol. By documenting the safety-critical functions early in the concept exploration phase, it identifies those software functions or modules that are safety-critical by definition.

The identification of the SCFL is a multi-discipline activity that is initiated by the safety manager. Identification of the preliminary safety functions of the system requires inputs from systems, design, safety, reliability engineering, project managers, and the user. It requires an early assessment of the system concepts, specifications, requirements, and lessons learned. Assessing the system design concepts includes analysis of:

- Operational functions,
- Maintenance and support functions,
- Test activities,
- Transportation and handling,
- Operator/maintainer personnel safety,
- Software/hardware interfaces,
- Software/human interfaces,

- Environmental health and safety,
- Explosive constraints, and
- Product loss prevention.

Safety-critical functions identified should be documented, tracked, and matured as the design matures. This is to say that a distinct safety-critical function in preliminary design maybe completely eliminated in detailed design, or a function could be added to the preliminary list as either the design matures, or customer requirements change.

The point of the SCFL is to ensure that software code, or modules of code that perform safety-critical functions are defined and prioritized as safety-critical code/modules. This is based on credible, system-level, functions that have been identified as safety-critical. The safety-critical identifier on software establishes the design, code, test, and IV&V activities that must be accomplished to ensure the software is safety risk minimized.

C.1.5 PRELIMINARY HAZARD ANALYSIS

The PHA activity is a safety engineering and software safety engineering function that is performed to identify the hazards and their preliminary casual factors of the system in development. The hazards are formally documented to include information regarding the description of the hazard, casual factors, the effects of the hazard, and preliminary design considerations for hazard control by mitigating each cause. Performing the analysis includes assessing hazardous components, safety-related interfaces between subsystems, environmental constraints, operation, test and support activities, emergency procedures, test and support facilities, and safety-related equipment and safeguards. The PHA format is defined in DI-SAFT-80101A of MIL-STD-882C. This DID also defines the format and contents of *all* hazard analysis reports. An example of a PHA hazard record is provided in Figure 4-24.

The analysis also provides an initial assessment of hazard severity and probability of occurrence. The probability assessment at this point is usually subjective and qualitative.

To support the tasks and activities of a software safety effort, the “causes” of the root hazard must be assessed and analyzed. These causes should be separated in four separate categories:

- Hardware initiated causes,
- Software initiated causes,
- Human error initiated causes, and
- Human error causes that were influenced by software input to the user/operator.

This categorization of causes assists in the separation and derivation of specific design requirements that are attributed to software. Both software-initiated causes, and human error causes influenced by software input must be adequately communicated to the systems engineers and software engineers for the purpose of the identification of software design requirements to preclude the initiation of the root hazard identified in the analysis.

The PHA document itself is a living document that must be revised and updated as the system design and development progresses. It becomes the input document and information for all other hazard analyses performed on the system. This includes the SSHA, SHA, Health Hazard Assessment (HHA), and O&SHA.

C.1.6 SUBSYSTEM HAZARD ANALYSIS

The hazard analysis performed on individual subsystems of the (total) system is the SSHA. This analysis is "launched" from the individual hazard records of the PHA which were identified as a logically distinct portion of a subsystem. Although, the PHA is the starting point of the SSHA, it must be only that - a starting point. The SSHA is a more in-depth analysis of the functional relationships between components and equipment (this also includes the software) of the subsystem. Areas of consideration in the analysis include performance, performance degradation, functional failures, timing errors, design errors, or inadvertent functioning.

As previously stated, the SSHA is a more in-depth analysis than the PHA. This analysis begins to provide the evidence of requirement implementation by matching hazard causal factors to "what is" in the design to prove or disprove hazard mitigation. The information that must be recorded in the SSHA include, but is not limited to, hazard description, all hazard causes (hardware, software, human error, or software-influenced human error), hazard effect, and derived requirements to either eliminate or risk-reduce the hazard by mitigating each causal factor. The inverse of a hazard cause can usually result in a derived requirement. The analysis should also define preliminary requirements for safety warning or control systems, protective equipment, and procedures and training. Also of importance in the data record is the documentation of design phase of the program, component(s) affected, component identification per drawing number, initial hazard HRI (which includes probability and severity prior to implementation of design requirements), and the record status (opened, closed, monitored, deferred, etc.).

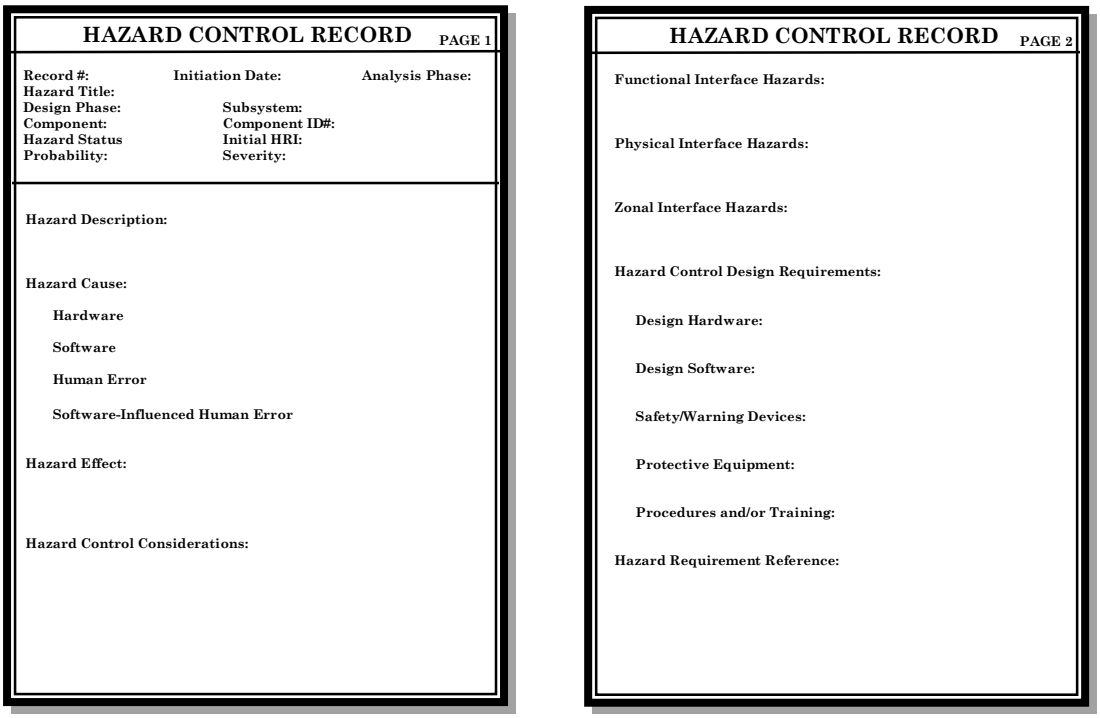
From a software safety perspective, the SSHA must define those hazards or failure modes that are specifically caused by erroneous, incomplete or missing specifications (including control software algorithm elements, interface inputs and outputs, and threshold numbers), software inputs, or human error (influenced by software furnished information). These records must furnish the basis for the derivation and identification of software requirements that eliminate or minimize the safety risk associated with the hazard. It also must initiate resolution of how the system, or subsystem, will react given the software error does occur. Fault resolution scenarios must consider the reaction of the subsystem and/or system if a potential software error (failure mode) becomes a reality. For example, if a potential error occurs does the system power down, detect the error and correct it, go to a lesser operational state, fail soft, fail safe, fail operational, fail catastrophic, or some combination of these.

C.1.7 SYSTEM HAZARD ANALYSIS

The SHA provides documentary evidence of safety analyses of the subsystem interfaces and system functional, physical, and zonal requirements. As the SSHA identifies the specific and unique hazards of the subsystem, the SHA identifies those hazards introduced to the system by the interfaces between subsystems, man/machine, and hardware/software. It assesses the entire

system as a unit and the hazards and failure modes that could be introduced through system physical integration and system functional integration.

Although interface identification criteria is not required or defined in the SSHA, it is to be hoped that preliminary data is available, and provided by the SSHA analysis format to assist in a "first cut" of the SHA. The SHA is accomplished later in the design life cycle (after PDR, and before CDR) which increases the cost of design requirements that may be introduced as an output of this analysis. Introducing new requirements this late in the development process also reduces the possibility of completely eliminating the hazard through the implementation of design requirements. It is therefore recommended that initial interface is considered as early as possible in the PHA and SSHA phases of the safety analysis. Having this data in preliminary form allows the maturation of the analysis in the SHA phase of the program to be timelier. An example of the minimum information to be documented in an SSHA and SHA is provided in Figure C.2.



HAZARD CONTROL RECORD PAGE 1

Record #: Initiation Date: Analysis Phase:
Hazard Title:
Design Phase: Subsystem:
Component: Component ID#:
Hazard Status Initial HRI:
Probability: Severity:

Hazard Description:

Hazard Cause:

Hardware

Software

Human Error

Software-Influenced Human Error

Hazard Effect:

Hazard Control Considerations:

HAZARD CONTROL RECORD PAGE 2

Functional Interface Hazards:

Physical Interface Hazards:

Zonal Interface Hazards:

Hazard Control Design Requirements:

Design Hardware:

Design Software:

Safety/Warning Devices:

Protective Equipment:

Procedures and/or Training:

Hazard Requirement Reference:

Figure C.2: SSHA & SHA Hazard Record Example

C.1.8 SAFETY REQUIREMENTS CRITERIA ANALYSIS

Periodically during the development activities, the requirements generated (generic), allocated, or derived from the safety analysis must be communicated and delivered to the design engineering function for both hardware and software. This is accomplished via the systems engineering function for product development. System engineering is responsible for the assurance that the system definition and design reflect requirements for all system elements to include equipment, software, personnel, facilities, and data. Each functionally derived requirement is "tied" or "traced" to a specific hazard record or multiple records. This helps communicate the requirement rationale to those designers that do not understand the necessity or the intent of the requirement.

If the safety manager has implemented a hazard record database for tracking hazards, it should contain database fields for the purpose of the documentation of derived requirements for each identified hazard and its related causes. If these requirements are documented in the database, it should be relatively simple to produce a SRCA using the report generation function of the database. The main point is that safety requirements will not be implemented in the design if they are not delivered and communicated to the design engineers. If the design team does not understand a unique requirement, it must be traced back to a specific hazard record for review and discussion. At this point, two alternatives exist; first, the hazard record is incorrect based on misconceptions in the design; or the hazard record is correct. If the first scenario is true, the record can be corrected and the requirement changed, or conversely, the record is correct and the requirements derived by the analysis remain valid. In either case, the necessary requirements are adequately and efficiently communicated to those domain design groups that are responsible for their implementation.

As previously stated, the safety engineers must be aware that the derivation of requirement to eliminate a specific hazard may introduce a unique and separate hazard. Some procurement activities will address this issue through a requirement for a "requirements hazard analysis." This analysis is for the purpose of insuring that requirements identified in the analysis process do not introduce hazards as a byproduct of the initial analysis and requirements definition process.

C.1.9 SAFETY REQUIREMENTS VERIFICATION REPORT

A common misconception of system safety engineering is that the most important products produced by the analysis is the hazard analysis reports (PHA, SSHA, SHA, HHA, and O&SHA). This, however, is only partially correct. The primary product of system safety engineering analysis is the identification and communication of requirements to either eliminate or reduce the safety risk associated with the design, manufacture, fabrication, test, operation, and support of the system. Once identified, these requirements must be verified to be necessary, complete, correct, and testable for the system design. Not only is it the responsibility of the system safety function to identify, document, track, and trace hazard mitigation requirements to the design, but also to identify, document, participate in, or possibly perform, the verification activities. These activities can be wide and varied according to the depth of importance communicated by program management, design and systems engineering, and system safety. If a program is hindered by limited resources, this verification may be as simple as having the design engineers review the hazard records under their responsibility, verify that all entries are correct, and verify that the designers have indeed incorporated all the derived requirements associated with the record. On the other hand, the verification activities may be as complicated as initializing and analyzing specific testing activities and analyzing test results; the physical checking of as-built drawings and schematics; the physical checking of components and subsystems during manufacture; and/or the physical checking of technical orders, procedures, and training documentation.

The Safety Requirements Verification Report formally provides the audit trail for the formal closure of hazard records at the System Safety Group (SSG) meetings. This report is not a mandatory requirement if the hazard tracking database format contains the necessary fields documenting the evidence of safety requirement implementation (Figure C.3). The PM, before closing a hazard record, must be assured that requirements functionally derived or identified via

the hazard record have been incorporated and implemented. There must also be assurance that generic SSRs have been adequately implemented. Those requirements not implemented must also be identified in the assessment of residual risk and integrated in the "final HRI" of the hazard record. In addition, this formal documentation is required information that must be presented to the test and user organizations. They must be convinced that all identified hazards and failure modes have been minimized to acceptable levels of safety risk prior to system-level testing.

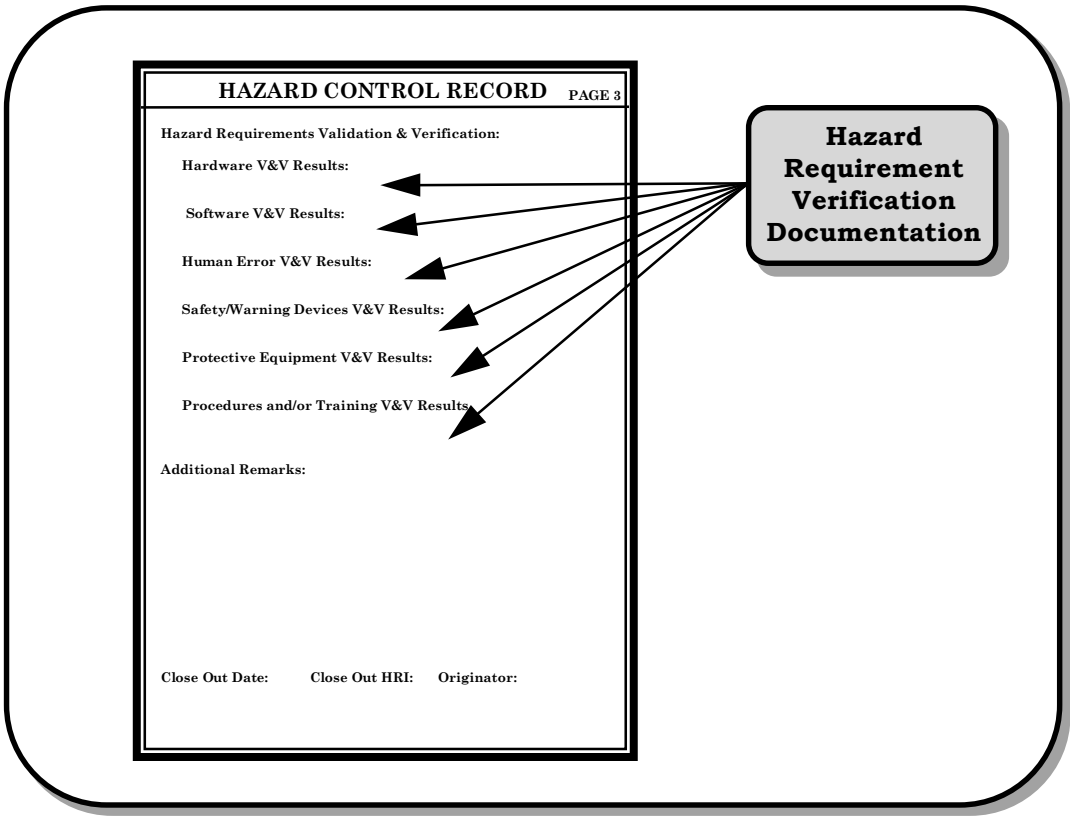


Figure C.3: Hazard Requirement Verification Document Example

C.1.10 SAFETY ASSESSMENT REPORT

From the perspective of the PM, the SAR is one of the most important products that is produced by the safety team. It documents the residual safety risks associated with the test, operations, and support of the system. The SAR formally identifies all safety requirements not implemented (or partially implemented), and those hazards that were risk minimized via the safety order of precedence activities. In addition, it documents the ramifications of not proceeding further with designing "out" the residual hazardous conditions of the system. The decision not to continue is usually based upon the lack of financial resources, technological restrictions, or programmatic decisions.

The SAR also helps to establish operational test scenario bounds and limits based upon residual safety risk in subsystems and their interfaces. This assessment can assist the PM and the test manager in critical decisions regarding the breadth and depth of initial testing. The analysis can

identify which test functions and scenarios must be tested incrementally based on sound safety data. Specific guidelines for the accomplishment of a safety assessment are documented in Task 301 of MIL-STD 882C, and DID DI-SAFT-80102A.

C.2 CONTRACTUAL DOCUMENTATION

C.2.1 STATEMENT OF OPERATIONAL NEED

A product life cycle begins with the Statement of Operational Need (SON), which is a product of the Mission Area Analysis (MAA). The MAA identifies deficiencies and shortfalls in defense capabilities or defines more effective ways of accomplishing existing tasks. The purpose of the SON is "*to describe each need in operational terms relating to planned operations and support concepts."* (AFR 57-1) This document will provide the software safety team with a definition of operational need of the product and an appreciation of where the concept was originated (to include assumptions).

Most SONs do **not** have specific or specified statements regarding design, manufacture, fabrication, test, operational and support, or software safety. If they did, the planning, support, and funding for system safety engineering would be a lot easier to secure in the initial phases of the systems development. Since the SON will most likely be void of any safety-related statement, it should be reviewed for the purpose of understanding the background, needs, desires, requirements, and specifications of the ultimate system user. This helps in the identification of scope, breadth, and depth of the SSP and software safety program, and an understanding of what the user considers to be acceptable in terms of safety risk. With little (or no) verbiage regarding safety in the SON, a communication link with the user (and customer if different from the user) is essential.

C.2.2 REQUEST FOR PROPOSAL

Although not always addressed specifically, many modern-day RFPs include an "implied" requirement for a system safety engineering program which may include software safety activities. In today's environment of software-intensive systems, an "implied" requirement is no longer acceptable. The user must request in detail a specified SSP, applicable safety criteria, definitions of acceptable risk, levels of required support, and anticipated deliverables required by the customer.

The primary reason for detailed safety criteria in the RFP is for the establishment (and documentation) of the defined scope and the level-of-effort for a SSP. The importance of this is simple. It establishes for the management team, a design definition that can be accurately planned, budgeted, and implemented. One of the biggest obstacles for most SSPs is the lack of sufficient budget and program resources to adequately accomplish the system safety task in sufficient detail. This is due to insufficient criteria detail in the RFP, SOW, and contract. Without this detail, the developer may incorrectly bid the scope of the system safety and software safety portion of the contract. This level of detail within the RFP is the responsibility of the *user*.

Planning at this stage of the program for the *developer* consists primarily of a dialog with the customer to specifically identify those requirements or activities that they (the user) perceive to

be essential for reducing the safety risk of software performing safety-critical functions. It is at this point that the safety managers in conjunction with the software development manager assess the specific SSRs to fully understand the customer desires needs, and requirements. It is to be hoped that this is predicated on detailed safety criteria documented in the RFP. Once again, dialog and communication are required by the developer with the customer to ensure that each safety program requirement is specifically addressed in the proposal. A sample RFP statement for safety is included in Appendix G.

C.2.3 CONTRACT

Specific contract implications regarding system safety and software safety engineering and management is predicated on proposal language, type of contract, and specific contract deliverables. The software safety team must ensure that all outside (the software safety team) influences are assessed and analyzed for impact to the breadth and depth of the program. For example, a development contract may be Cost-Plus-Award-Fee (CPAF). Specific performance criteria of the safety and software team may be tied directly to the contract award fee formula. If this is the case, the software safety team will be influenced by contractual language. Additionally, the specifics of the contract deliverables regarding system safety criteria for the program will also influence the details defined in the SSPP. The software safety team must establish the program to meet the contract, and contractual deliverable requirements

C.2.4 STATEMENT OF WORK

The program SOW, and its parent contract, is mandatory reading for the SwSE and the software safety team members associated with the program. Planning and scoping the software safety program, processes and tasks, and products to be produced are all predicated on the contract and SOW. They define contractual requirements, scope of activity, and required deliverables. The contract and SOW will become the "launch pad" for the development of the SSPP and either the software safety appendix, or the SwSPP. If you recall, the SSPP defines how the developer will meet all program objectives, accomplish/produce the required contract deliverables, and meet scheduled milestone activities. When approved it is normally a contractually binding document.

Developing and coordinating the system safety paragraph contents of a SOW, in many instances, is one of the most unplanned and uncoordinated activities that exists between the customer and the supplier. On numerous occasions, the customer does not know specifically what they want, so the SOW is intentionally vague, or conversely calls for all tasks of a regulatory standard (i.e., MIL-STD-882). In other instances the SOW is left vague as to "not stifle contractor innovation." The fact is that most system safety activities required by the SOW are not coordinated and agreed upon by the contractor or the customer. This problem would be significantly minimized if sufficient details were provided in the RFP. PMs and technical managers must realize that specific criterion for a program's breadth and depth can be adequately stated in a RFP and SOW without dictating to the developer *how* to accomplish the specified activities.

Unfortunately, the RFP and SOW seldom establish the baseline criteria for an adequate SSP. With this in mind, two facts must be considered. First, the SOW is usually an approved and contractually binding document that defines the breadth of the safety work to be performed. Second, it is usually the depth of the work to be performed that becomes the potential for

disagreement. This is often predicated on the lack of specified criteria in contractual documents and lack of allocated critical resources by program management to support the activities in question. Planning for the system safety activities that will be required contractually should be an activity initiated by the PA. It is the responsibility of the customer to understand the resource, expertise, and schedule limitations and constraints of the contractor performing the safety analysis. This knowledge can be useful in scoping the SOW requirements to ensure that a practical program is accomplished which identifies, documents, tracks, and resolves the hazards associated with the design, development, manufacture, test, and operation of a system. A well-coordinated SOW safety paragraph should accurately define the depth of the SSP and define the necessary contract deliverables. The depth of the system safety and software safety program can be scoped in the contractual language if the customer adequately defines the criteria for the following:

- The goals and objectives of the system safety design, test, and implementation effort,
- Technical and educational requirements of the system safety manager and engineers,
- The allocated "system" loss rate requirements to be allocated for the subsystem design efforts,
- The specific category definitions pertaining to hazard probability and severity for the HRI and the SHCM,
- The defined system safety engineering process and methods to be incorporated,
- The specific scope of effort and closeout criteria required for each category of HRI hazards, and
- The required contractual safety deliverables, including acceptable format requirements.

For DOD procurements, current versions of both MIL-STD-882 and MIL-STD-498 should be used in the preparation of the SOW software safety tasks, as shown in Figure C.4. While these military standards may not be available in future years, their commercial counterpart should be available for use. It must be understood that these two standards are very complementary in their criteria for a software safety program. While MIL-STD-882 provides a "total system" approach to system safety engineering, MIL-STD-498, Paragraph 4.2.4.1, ensures that a strategy is in place for a software development to identify and resolve safety-critical CSCIs whose failure could lead to a hazardous system state. The safety requirements in MIL-STD-498 are totally in concert with the methods and philosophy of system safety engineering.

Although not recommended, if the contractor is tasked with providing a draft SOW as part of the proposal activity, the customer (safety manager) should carefully review the proposed paragraphs pertaining to system safety and resolve potential conflicts prior to the approval of the document. Examples of SOW/SOO system safety paragraphs are provided in Appendix G.

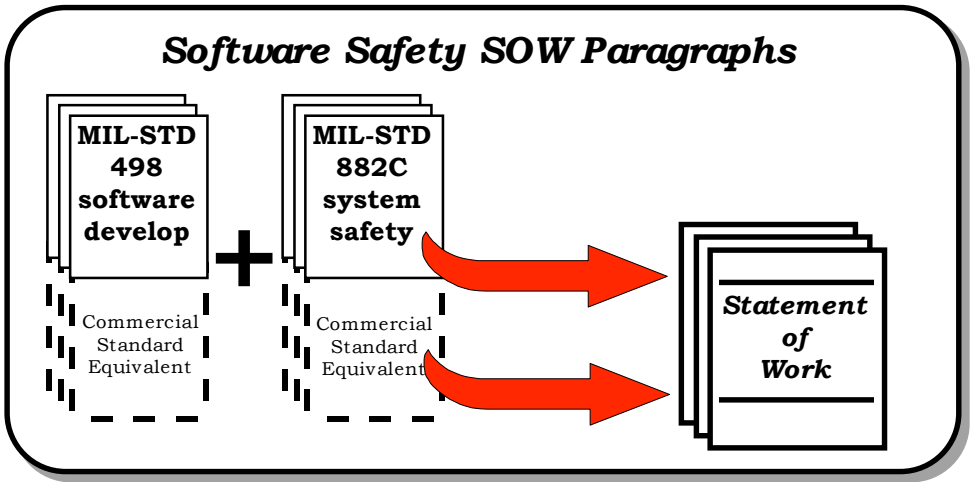


Figure C.4: Software Safety SOW Paragraphs

C.2.5 SYSTEM AND PRODUCT SPECIFICATION

The system specification identifies the technical and mission performance requirements of the product to be produced. It allocates requirements to functional areas, documents design constraints, and defines the interfaces between or among the functional areas. [DSMC, 1990] The system specification must be thoroughly reviewed by the safety manager and the software safety team members to ensure those physical and functional specifications are completely understood. This will assist in the identification and scope (breadth and depth) of the analysis to be performed. It also provides an understanding of the technologies involved, the magnitude of the managerial and technical effort, assumptions, limitations, and engineering challenges, and the inherent safety risk of the program.

The system specification should identify and document any quantified safety-critical requirements that must be addressed. As an example, an aircraft development program may have a system specification for "vehicle loss rate" of $1x10^{-6}$. It is the responsibility of the safety manager, in concert with design engineering, to allocate this loss rate to each effected subsystem. Each major subsystem would be allocated a "portion" of the loss rate requirement, such that, any design activity that causes a negative impact to the loss rate allocation would be "flagged" and resolved. The safety and engineering leads are required then, to track how close each design activity is to their vehicle loss rate allocation. If each design team meets, or exceeds, their allocation, the aircraft will meet its design specification. An interesting side note to this example is that Air Force accident records indicate that, in reality, the vehicle loss rate of operational aircraft is an order of magnitude less than the vehicle loss rate from which the aircraft was originally designed.

As one can see, there is information in the system specification that will influence the methods, techniques, and activities of the system safety and software safety engineering teams. The SSPP and/or SwSPP must reflect specific activities to meet the defined requirements of the system and user specifications.

C.2.6 SYSTEM AND SUBSYSTEM REQUIREMENTS

Another essential set of documents that must be reviewed by the system safety manager and engineers is the initial requirement documentation. In some instances, these initial requirements are delivered with the RFP, or they can be derived as a joint activity between the developer and the customer. Regardless, the knowledge and understanding of the initial system and subsystem requirements allow the safety engineers to begin the activities associated with the PHL and the PHA. This understanding allows for a greater fidelity in the initial analysis and reduces the time required in assessing design concepts that may not be in the design at PDR.

For the preliminary software design activities, the generic SSRs and guidelines must be a part of the initial requirements. An example of these preliminary (generic) requirements and guidelines are provided in Appendix D. This list, or similar lists, must be thoroughly reviewed, analyzed, and only those that are deemed appropriate provided to the software design team.

C.3 PLANNING INTERFACES

C.3.1 ENGINEERING MANAGEMENT

The lead engineer, or engineering manager, is pivotal in ensuring that each subsystem, system, interface, or support engineer provides the required support to the system safety team. This support must be timely to support the analyses, contractual deliverables, and schedule milestones of the contract and the safety SOW. Also, the engineering support must be timely to effectively influence the design, development, and test of the system, each subsystem, and their associated interfaces. The engineering manager has direct control over the allocation of the engineering resources (including the engineers themselves) for the direct support of system safety engineering analyses. Without the support of the lead engineer, analyses performed by the safety team would have to be based on “best guess” assumptions and inaccurate engineering data and information. Engineering Management:

- Coordinates the activities of the supporting technical disciplines,
- Manages technical resource allocation,
- Provides technical input to the PM,
- Defines, reviews, and comments on technical adequacy of safety analyses, and
- Ensures that safety is an integral part of system engineering and is allocated sufficient resources to conduct analyses.

C.3.2 DESIGN ENGINEERING

The actual hardware and software design engineers must rely on the completeness and correctness of requirements derived by support functions such as safety, reliability, maintainability, and quality. The derivation of these requirements cannot be provided from a vacuum (i.e., a lone analyst producing requirements without understanding the functional and physical interfaces of the system). Therefore, design engineers are also dependent on the validity

and fidelity of the system safety process to produce credible requirements. Safety-specific requirements are obtained from two sources; first, generic safety requirements and guidelines (see Appendix D) and second, derived requirements produced through hazard analysis and failure mode analysis. Generic safety requirements are normally derivatives of lessons learned and requirements identified on similar systems. Regardless of the source of the safety requirement, it is essential that the design engineer completely understand the intent of each requirement and the ramifications of not implementing that requirement in the design.

Correctness, completeness, and testability are also mandatory attributes of safety requirements. The correctness and completeness of safety requirements, in most instances, are predicated on the communication of credible hazards and failure modes to the designer of a particular subsystem. Once the designer understands the hazard, specific requirements to eliminate or control the hazard are derived in concert by the safety engineer and design engineer.

The ultimate product of a system safety engineering activity is the elimination and/or control of the risk associated with hazards and failure modes. The incorporation and implementation of safety-specific design requirements accomplish this. Design engineering must have intimate knowledge of the system safety, and software safety activities and the intent of the safety requirements derived. The knowledge of the safety processes and the intent of the requirements by the design team establish credibility for the safety engineer to actively perform within the design function.

C.3.3 SYSTEMS ENGINEERING

Systems Engineering is defined as:

"An interdisciplinary approach to evolve and verify an integrated and optimally balanced set of product and process designs that satisfy user needs and provide information for management decision making." [MIL-STD-499B Draft]

"Systems engineering, by definition, involves design and management of a total system comprised of both hardware and software. Hardware and software are given equal weight in analysis, tradeoffs, and engineering methodology. In the past, the software portion was viewed as a subsidiary, follow-on activity. The new focus in systems engineering is to treat both software and hardware concurrently in an integrated manner. At the point in the system design where the hardware and software components are addressed separately, modern engineering concepts and practices must be employed for software, the same as hardware." [Software Technology Support Center (STSC) 1, 1994]

The overall objectives of systems engineering, [DSMC 1990], are to perform the following:

- Ensure that system definition and design reflects requirements for all system elements: equipment, software, personnel, facilities, and data,
- Integrate technical efforts of the design team specialists to produce an optimally balanced design,

- Provide a comprehensive indentured framework of system requirements for use as performance, design, interface, support, production, and test criteria,
- Provide source data for development of technical plans and contract work statements,
- Provide a system framework for logistic analysis, ILS trade studies, and logistic documentation,
- Provide a system framework for production engineering analysis, producibility trade studies, and production manufacturing documentation, and
- Ensure that life cycle cost consideration and requirements are fully considered in all phases of the design process.

Each of the preceding is important to system safety engineering and has the potential, if not accomplished, to impact the overall safety of the system in development. Ultimately, system engineering has the responsibility for developing and incorporating all design requirements that meet the system operational specifications. This includes the safety-specific requirements for hardware, software, and human interfaces. Systems engineering must be assured that system safety has identified and implemented a sound approach for identifying and resolving system hazards and failure modes. A systems engineering approach supports system safety engineering and the MIL-STD-882 safety precedence to design for minimum risk

Systems engineering must be in agreement with the processes and methods of the safety engineer to ensure safety requirements are incorporated in the design of the system. This process must have the ability to efficiently identify, document, track, trace, test, and validate requirements which reduce the safety risk of the system.

C.3.4 SOFTWARE DEVELOPMENT

The software engineering/development interface is one of the most critical interfaces to be cultivated for a successful software safety engineering activity. The software engineering team must understand the need associated with the analysis and review tasks of system safety engineering. They must also comprehend the methods to be used and the utility and products of the methods in the fulfillment of the software safety tasks. This interface is new to most software developers. Although they usually understand the necessity of controlling the safety risk of software performing in safety-critical systems, their view of system safety engineering principles is somewhat limited. It the responsibility of the SSS Team to assess the software development acquisition process and determine when, where, and how to be most effective in tracing SSRs to test. The software engineering team must support the SSS Team by allocating specific personnel to the team to assist in the safety activities. This usually consists of personnel that can address software design, code, test, IV&V, CM, and quality control (QC) functions.

C.3.5 INTEGRATED LOGISTICS SUPPORT

An inherent difference between hardware support and software support is that hardware support is based on the finished product, while software support must mimic the development process. Hardware support must use the tools necessary to repair a finished product, not tools required to

build one. Software support, on the other hand, must use tools functionally identical to those used during the development process.

Life cycle support strategies typically span the support spectrum from sole source contractor to full government organic, with each strategy presenting different advantages and disadvantages needing evaluation. A high level IPT consisting of the operational user, the PEO, and the acquisition agent must make the support decision prior to Milestone I. This focuses attention on the software support process and allows the acquisition agent to begin planning for it earlier in the program.

The Computer Resources Integrated Support Document (CRISD) is the key document for software support. It determines facility requirements, specifies equipment and required support software, and lists personnel number, skills, and required training. It contains information crucial to the establishment of the Software Engineering Environment (SEE), its functionality, and limitations. It is a management tool that accurately characterizes the SEE's evolution over time. [STSC, 1996]

From a software safety perspective, the software support environment must be aware of the safety implications of the software to be maintained and supported in the operational environment. Safety-critical functions and their relationships to controlling software must be fully defined. That is, any software code that directs, functions, or controls safety-critical functions of the system must be fully defined and communicated in the software support documentation. Any software ECP or maintenance support function pertaining to safety-critical modules of code must be thoroughly reviewed by the SSS Team prior to implementation.

C.3.6 OTHER ENGINEERING SUPPORT

Each program development will have unique differences and specific interface requirements. As an example, a potential customer may require that all system hazard probabilities be quantified to a specific confidence level, and that no qualitative engineering estimates will be acceptable. This requirement forces the safety engineering analysis to be heavily predicated on the outputs of reliability engineering. Although interfaces with reliability engineering are common place for most DOD procurements, the example demonstrates the necessity of cultivating programmatic and technical interfaces based on contractual, technical, and programmatic obligations. Other technical interface examples may include, but are not limited to, human factors, QA, reliability engineering, supportability, maintainability, survivability, test and evaluation, IV&V and ILS.

Planning for, and securing agreement with, the managerial and technical interface precludes the necessity of trying to “cram” it into defined processes at a later date. Each program is unique and presents planning challenges and peculiar interfaces.

C.4 MEETINGS AND REVIEWS

C.4.1 PROGRAM MANAGEMENT REVIEWS

Program Management Reviews (PMR) are normally set up on a routine scheduled basis to allow the PM to obtain an update of program status. The update will consist of the review of

significant events since the previous PMR short-term and long-term schedules, financial reports and budgeting forecasts, and technical contributions and limitations in design. Timely and valid inputs to the PMR provide the PM, in concert with other key program individuals, the information necessary to make informed decisions that affect the program. This may include the allocation of critical resources. System safety may, or may not, have briefing items on the PMR agenda. This would be predicated on whether the PM/director specifically requires the presentation of the current status of the safety program, and whether safety issues should be raised to this decision-making level. Caution should be given at this point to only raise safety issues to this level of program management that is deemed essential to resolve an issue of a critical nature (e.g., resolution of an important safety issue could not be attained at any other lower level of technical or managerial support in the time required to solve the problem). It is recommended that the safety manager attend the PMRs for programmatic visibility and to keep abreast of program milestones and limitations as well as technical contributions and limitations.

C.4.2 INTEGRATED PRODUCT TEAM MEETINGS

The IPT is the team of individuals that ensures an Integrated Product Development (IPD). "IPD is a philosophy that systematically employs a teaming of functional disciplines to integrate and concurrently apply all necessary processes to produce an effective and efficient product that satisfies customer's needs."[25] It applies to the entire life cycle of a product.

The IPT provides a technical-managerial framework for a multi-disciplinary team to define the product. IPT emphasizes up-front requirement definition, trade-off studies, and the establishment of a change control process for use throughout the entire life cycle. [STSC #1,1994] From a system safety perspective, IPTs effect how the processes defined in a TEMP are integrated for the development and support required by safety throughout the project. This includes special considerations, methods, and techniques defined by IPT members and supporting technical disciplines.

C.4.3 SYSTEM REQUIREMENTS REVIEWS

System Requirements Reviews are normally conducted during the concept exploration or demonstration/validation phase of a program to ensure that system level functional analysis is relatively mature and that system-level requirements have been allocated. The purpose is to ensure system requirements have been completely and correctly identified, and that mutual agreement is reached between the developer and the customer on system requirements. Particular emphasis is placed on ensuring that adequate consideration has been given to logistic support, safety, software, test, and production constraints.

Primary documents used in this review consist of the documentation products of the system requirement allocation process. This includes functional analysis, trade studies, functional flow block diagrams, requirement allocation sheets, and the requirements allocation reports from other disciplines. This includes disciplines as reliability, supportability, maintainability, human factors, and system safety.

[25] AFMC Manual, Acquisition Management Acquisition Logistics Management, 19 January 1995 (DRAFT)

The system safety manager must assure that the safety requirements have been thoroughly captured and that they cover known hazards. The list of known (or suspected) hazards from the PHL or PHA should be used as a guide to check against each and every system/subsystem requirement. For safety verification of new requirements, the system/subsystem requirements must then be checked individually to ensure that new hazards have not been introduced. Requirements should be traceable to systems, subsystems and their respective interfaces (i.e., human/machine, hardware/software, and system/environment) as well as to specific system/subsystems hazards and failure modes. Traceability of allocated requirements to the capability of the system to meet the mission need and program objectives within planned resource constraints must be demonstrated by correlation of technical and cost information. [DSMC 1990]

It is at this time that requirements considered *safety-critical* be already defined and documented. A formal method to flag these requirements in documentation must be in place and any attempt to change or modify these requirements must be limited. This documentation process must include a check-and-balance that notifies the safety manager if safety-critical functions or requirements are considered for change, modification, or elimination.

C.4.4 SYSTEM/SUBSYSTEM DESIGN REVIEWS

The System Design Review (SDR) is one of the final activities of the Demonstration/Validation phase of the program and thus, becomes the initial review of the Engineering/Manufacturing Development (EMD) phase. The purpose of the SDR is to assess and evaluate the optimization, traceability, correlation, completeness, and risk of the system level design that fulfills the system functional baseline requirements. The review assesses and evaluates total system requirements for hardware, software, facilities, personnel, and preliminary logistic support. This review also assesses the systems engineering activities that help establish the products that define the system. These products include trade studies, functional analysis and allocation, risk analysis, mission requirements, manufacturing methods and processes, system effectiveness, integrated test planning, and configuration control.

C.4.5 PRELIMINARY DESIGN REVIEW

PDRs are normally conducted on each hardware and software Configuration Item (CI) (or functionally grouped CIs), after top-level design efforts are complete and prior to the start of detailed design. Usually, the PDR is held after the approval of the development specifications and also prior to system level reviews. The review focuses on technical risk, preliminary design (including drawings) of system elements, and traceability of technical requirements. Specific documentation for CI reviews include development specifications, trade studies supporting preliminary design, layout drawings, engineering analysis (including safety hazard analysis, human engineering, failure modes and effects analysis, and ILS), interface requirements, mock-ups and prototypes, and computer software top-level design, and software test planning. Special attention is given to interface documentation, high-risk areas, long lead-time procurement, and system level trade studies.

The safety engineer must provide to the PDR process:

- The generic requirements and guidelines which were allocated to the system functional baseline,
- The SCFL,
- Results of any safety-related trade studies,
- The specific requirements derived through the PHA and PHL, and
- The specific requirements derived through the initial efforts of the SSHA, SHA and the O&SHA.

The primary focus of the safety team is to ensure that the generic and the specific derived requirements are documented in the requirements specification, communicated to the appropriate design function, and the safety intent of each is adequately displayed in the proposed design. The primary output of the PDR is an assurance that safety-specific requirements (and their intent) are being adequately implemented in the design. This assurance is provided is a two-way traceability of requirements to specific hazards and failure modes, and to the design itself.

C.4.6 CRITICAL DESIGN REVIEW

The CDR is conducted for each hardware and software CI before release of the design for manufacturing, fabrication, and configuration control. For software, the CDR is conducted prior to coding and preliminary and informal software testing. The CDR discloses the detailed design of each CI as a draft product specification and related engineering drawings. The design becomes the basis for final production planning and initial fabrication. In the case of software, the completion of the CDR initiates the development of source and object code. Specific review items of a CDR include; detailed engineering drawings, interface control drawings, prototype hardware, manufacturing plans and the QA plans.

The safety engineer must provide to the CDR process:

- All safety specific requirements derived from the safety analyses activities. This to include trade studies, functional and physical analyses, and PHA activities,
- All generic safety requirements and guidelines that were communicated to the design team during initial requirement allocations,
- The finalized SCFL, and
- All safety-specific testing requirements to verify the implementation of safety requirements in the design.

The CDR provides the evidence to the safety engineer that the design meets the goals and objectives of the SSP and that the designers have implemented each safety requirement in the final design. An approved CDR normally places the design under formal configuration control.

The safety team must leave the CDR with:

- Assurance of requirements traceability from analysis to design,

- Evidence that the final design meets the goals and objectives of the SSPP.
- Evidence that safety design requirements are verifiable, including requirements that must be verified through testing, and
- Evidence (quantifiable, if practical) of safety risk residual in the design.

C.4.7 TEST READINESS REVIEW

The TRR is a formal review of the developer's readiness to proceed with CI testing. On software developments, it is the readiness to begin CSCI level of that specific CSCI. The TRR shall attempt to assure that the safety requirements have been properly built into the system/subsystem, and that safety test procedures are complete and mature. It also ensures that residual safety risk is defined and understood by program management and engineering. Additionally, the TRR verifies that all of the system safety functionality and requirements have been incorporated through verification methods.

Another safety aspect of the TRR is the actual safety implications of the test itself. There are usually greater safety implications on the test of hardware systems as compared to the testing of software CSCIs and CSUs. Safety inputs to the TRR include:

- Safety analysis of the test itself,
- GO, NO-GO test criteria,
- Emergency test shut-down procedures,
- Emergency response procedures, and
- Test success or failure criteria.

If the specific test is for the verification of safety requirements, safety inputs to the TRR must also include:

- Documentation of the safety requirements to be verified during the test activities,
- Credible "normal" and credible "abnormal" parameters of test inputs,
- Expected response to normal and abnormal test parameters and inputs,
- Methodology to document test outputs and results for data reduction and analysis, and
- Methodology to determine test success or failure.

To support the TRR for embedded software programs, the safety manager must verify the traceability of safety requirements between system hazards and failure modes, and specific modules of code. Failure to identify the hazard-to-requirement will frustrate the ability to ensure that a hazard has been adequately mitigated until the safety test is run or test results are reviewed. This could result in software that does not contain adequate safety control for further testing or

actual delivery. Further possible ramifications include a requirement to re-engineer the code (influencing cost and schedule risks), or result in unknown residual safety risk.

C.4.8 FUNCTIONAL CONFIGURATION AUDIT

The objective of the Functional Configuration Audit (FCA) is to verify that the CIs actual performance complies with the hardware and software development interface requirement specifications. The hardware and software performance is verified by test data in accordance with its functional and allocated configuration. FCAs on complex CIs may be performed on an incremental basis, however, it must be performed prior to release of the configuration to a production activity.

The safety team must provide all functional safety requirements and functional interface requirements recommended for configuration audit. These requirements must be prioritized for safety, and should include first, those that influence safety-critical functions. The safety team must ensure that the configuration verifies the implementation of the safety design requirements and that those requiring verification have indeed been verified. Verification of safety requirements must be traceable throughout the systems functional configuration and to the hazard record.

For software specifically, agreement is reached on the validity and completeness of the Software Test Reports. The FCA process provides technical assurance that the software safely performs the functions against respective CSCI requirements and operational and support documentation.

C.4.9 PHYSICAL CONFIGURATION AUDIT

The Physical Configuration Audit (PCA) is the formal examination of the "as-built" version of the configuration item against the specification documentation that established the product baseline. Upon approval of the PCA activity, the CI is placed under formal CM control. The PCA also establishes the test and QA acceptance requirements and criteria for production units. The PCA process ensures that a detailed audit is performed on documentation associated with engineering drawings, specifications, technical data, and test results. On complex CIs the PCA may be accomplished in three phases; review of the production baseline; operational audit; and customer acceptance of the product baseline.

As with a PCA of hardware, the PCA for software is also a formal technical examination of the as-built product against its design. The design may include attributes that may not be customer requirements. If this situation exists, these attributes must also be assessed from a system safety perspective. The requirements from the SSHA and other software safety analyses (to include physical interface requirements) will be compared with the closure of software hazards as a result of design. Test results will be assessed to ensure that requirements are verified. In addition, the implemented design will be compared to as-built code documentation to verify that it has not been altered after it was tested (except for configuration control changes).

From a safety perspective, the most effective manner to conduct the audit is to target critical safety requirements. It is highly recommended that the PCA auditors choose Category 1 and Category 2 hazards to verify "as-built" safety of the particular CI. It should be noted that the

design, control, and test of safety requirements often involve the most complex, and fault tolerant code and architectures. As a consequence, they are often performed late in the testing schedule, giving a clear picture of the CI status.

C.5 WORKING GROUPS

C.5.1 SYSTEM SAFETY WORKING GROUP

The SSWG consists of individuals and expertise to discuss, develop, and present solutions for unresolved safety problems to program management or design engineering. They investigate engineering problem areas assigned by the SSG and proposed alternative design solutions to minimize safety risk. The requirement to have a SSWG is predicated on the need to resolve programmatic or technical issues. The SSWG charter should describe the intent and workings of the SSWG and the relationships to program management, design engineering, support functions, test agencies, and field activities.

The SSWG assists the safety manager in achieving the system safety and software safety objectives of the SSMP. To prepare for a SSWG, the safety manager for the development organization must develop a working group agenda from an itemized list of issues to be discussed. Also of importance, is the programmatic and technical support required for the meeting. The safety manager must ensure that the appropriate individuals are invited to the working group meeting and have had ample time to prepare to meet the objectives of the agenda.

The results of the SSWG are formally documented in the form of meeting minutes. The minutes should include a list of attendees the final agenda, copies of support documentation (including vu-graphs), documented resolutions, agreements, and recommendations to program management or engineering, and allocated action items.

C.5.2 SOFTWARE SYSTEM SAFETY WORKING GROUP

The Software System Safety Working Group (SwSSWG) is chaired by the software safety point of contact and may be co-chaired by the PA's PFS or SSPM. The SwSSWG is often the primary means of communication between the PA and the developer's software safety program. To be effective, the roles and responsibilities of the SwSSWG members, the overall authority and responsibilities of the SWSSG, and the procedures and processes for operation of the SwSSWG, must be clearly defined. Each SwSSWG must have a charter. This charter is generally appended to the SSPP. The charter describes:

- The authority and responsibilities of the safety and software safety points of contact,
- The roles and responsibilities of the membership,
- The processes to be used by the SwSSWG for accepting HARs and entering them into the hazard log,
- The processes and procedures for risk resolution and acceptance,
- The expected meeting schedule and frequency, and

- The process for closure of hazards for which the SwSSWG has closure authority.

The SwSSWG schedule includes periodic meetings, meetings prior to major program milestones, and splinter meetings as required to fully address the software safety issues

The SwSSWG is an IPT that is responsible for oversight and management of the software safety program. Therefore, membership obviously includes the:

- The safety principals [safety point of contact, PA PFS, safety PM(s)],
- System safety engineering,
- Software safety engineering,
- Systems Engineering,
- Software Engineering,
- Software Testing,
- IV&V,
- SQA,
- SCM, and
- Human Factors Engineering.

To be most effective, the SwSSWG must include representatives from the user community. A core group of the SwSSWG, including the safety Points of Contact (POC), the PA PFS, system engineering, software engineering and the user community should be designated voting members of the IPT if a voting structure is used for decisions making. Ideally, however, all decisions should be by consensus.

The DA's software safety POC chairs or co-chairs (with the PA's PFS for software safety) all formal SwSSWG meetings. The chair, co-chair or a designated secretariat prepares an agenda for each meeting. The agenda ensures that all members, including technical advisors, have time to address topics of concern. Any member of the IPT may request the addition of agenda items. The chair or secretariat forwards the agenda to all IPT members at least a week in advance of the meeting. Members wishing to include additional agenda items may contact the secretariat with the request prior to the meeting, such that the agenda presented at the opening of the meeting includes their topics. The chair must maintain control of the meeting to ensure that side discussions and topics not germane to the software safety program do not prevent accomplishing useful work. However, the key word in the concept of an IPT is "Team": the group should work as a team and not be unnecessarily constrained. The DA must make provisions for preparing and distributing meeting minutes. The meeting minutes formally record the results of the meeting including any decisions made and the rationale behind the decisions. These decisions may include the selection of a hazard resolution or decisions regarding hazard closure and the engineering rationale used to arrive at that decision. The chair or secretariat distributes the meeting minutes to all IPT members and retains copies in the SDL.

C.5.3 TEST INTEGRATION WORKING GROUP/TEST PLANNING WORKING GROUP

The Test Integration Working Group and the Test Planning Working Group may be combined on smaller programs. They exist to ensure that planning is being accomplished on the program to adequately provide test activities for developmental test, requirements verification test, and operational test. Developmental testing verifies the design concepts, configurations, and requirements meet the user and system specification. Operational and support testing verifies that the systems and its related components can be operated and maintained with the support concept of the user. This includes the utilization of user personnel in operating the system and supporting the system with the skill levels defined in the user specification. Operational testing verifies that the system operates as intended.

The safety input to test and integration planning is composed of the assurance that hazards and failure modes identified in design, manufacture, and fabrication have been either eliminated and/or controlled. Those hazards that have not been completely eliminated, and only controlled to specific levels, must be communicated to the test agency along with the assessment of residual safety risk. This allows the test agency to bound the parameters of the testing in conjunction with the safety risk exposure to the asset, the test site, the environment, and test personnel. Test planning must also include those safety-significant requirements that require verification through the test itself.

C.5.4 COMPUTER RESOURCES WORKING GROUP

The Computer Resources Working Group (CRWG) is formed as early as possible during the concept exploration phase but no later than Milestone 1[26]. The CRWG provides advice to the PM regarding the software support concept, computer resources policy, plans, procedures, standards, the TEMP, IV&V, and other development risk controls, all of which are influenced by the level of safety risk. The CRWG also contributes to the program's configuration control with external groups and interfaces. The CRWG includes the implementing agency, using agency, support agencies, and any other DOD agency.

From a system safety perspective, participation on the CRWG is essential to provide and ensure that software safety processes are included in all policies, procedures, and processes of the software development team. This ensures that safety-specific software requirements are integrated into the software design, code and test activities. The CRWG must recognize the importance of the safety input to the design and test functions of developing safety-critical software.

C.5.5 INTERFACE CONTROL WORKING GROUP

The Interface Control Working Group (ICWG) is one of the program office's controls over all external interfaces that effect the system under development.[27] The government ICWG has purview over areas external to the product such as interoperability and operational issues. The

[26] Mission Critical Computer Resources Management Guide, DSMC, Ft. Belvoir, VA
[27] ibid., pg. 10-5

ICWG works in coordination with the CRWG. The ICWG coordinates current and proposed software and hardware interfaces.

The developer's ICWG is, at first, most responsible for the software and hardware interfaces of the system under development and transfers this responsibility to the acquirer as the time for delivery of the software approaches. It is the ICWG that coordinates and signs off the interface requirements between developers and associate developers. All changes are reviewed by the ICWG before submittal to the acquirer's ICWG.

Software safety identifications and annotations to each change must accompany the initial interface coordination and any subsequent change to an interface to ensure preservation of safety and reduce residual safety risk.

C.6 RESOURCE ALLOCATION

C.6.1 SAFETY PERSONNEL

Once the specific tasks are identified in accordance with the contract and SOW, the safety manager must estimate the number of person-months that will be required to perform the safety tasks. This estimate will be based on the breadth and depth of the analysis to be performed, the number of contractual deliverables, and the funds obligated to perform the tasks. If the estimate of person-hours exceeds the allocated budget, the prioritized tasks that will be eliminated must be communicated to program management to ensure that they are in agreement with the decisions made, and that these discussion and agreements are fully documented. Conversely, if program management cannot agree with level of effort to be performed (according to the budget), they must commit to the allocation of supplemental resources to ensure the level of effort necessary to obtain the safety levels contractually required. Once again, specific safety tasks must be prioritized and scoped (breadth and depth) to provide program management (and the customer) with enough information to make informed decisions regarding the level of safety effort for the program.

Desired minimum qualifications of personnel assigned to perform the software safety portion of the system safety task are as follows:

- Undergraduate degree in engineering or technically related subject (i.e., chemistry, physics, mathematics, engineering technology, industrial technology or computer science),
- System safety management course,
- System safety analysis course,
- Software acquisition and development course,
- Systems engineering course, and
- Two to five years experience in system safety engineering or management.

C.6.2 FUNDING

The funds obligated to the program system safety effort must be sufficient to identify, document, and trace system safety requirements to either eliminate or control hazards and failure modes to an acceptable level of safety risk. Paragraph C.2.4 communicates the benefits of prioritizing tasks and program requirements for the purpose of allocating personnel to perform the safety tasks in accordance with the contract and the SOW. Sufficient funds must be allocated for the performance of system safety engineering which meet user and test requirements and specifications. Funding limitations and shortfalls must be communicated to the PM for resolution. Informed decisions by the PM can only be made if system safety processes, tasks, and deliverables are documented and prioritized in detail. This will help facilitate the allocation of funds to the program, or identify the managerial, economical, technical, and safety risk of underfunding the program. In addition, the cost-benefit and technical ramifications of decisions can be formally documented to provide a detailed audit trail for the customer.

C.6.3 SAFETY SCHEDULES AND MILESTONES

The planning and management of a successful SSP is supplemented by the safety engineering and management program schedule. The schedule should include near-term and long-term events, milestones, and contractual deliverables. The schedule should also reflect the system safety management and engineering tasks that are required for each lifecycle phase of the program and to support DOD milestone decisions. Also of importance, is specific safety data that is required to support special safety boards that may be required for compliance and certification purposes. Examples include, but are not limited to, FAA certification, DOD Explosive Safety Board, DNS Nuclear Certification, and the Non-Nuclear Munitions Safety Board. Each event, deliverable, and milestone should be tracked to ensure suspense and safety analysis activities are timely in the development process to help facilitate cost-effective design solutions to meet the desired safety specifications of the system development activity and the ultimate customer.

Planning for the SSP must include the allocation of resources to support travel of safety management and engineers. The contractual obligations of the SOW in concert with the processes stated in the program plans (Paragraph C.2.4) and the required support of program meetings (Paragraph C.4) dictate the scope of safety involvement. With the limited funds and resources of today's programs, the system safety manager must determine and prioritize the level of support that will be allocated to program meetings and reviews. The number of meetings that require support, the number of safety personnel that are scheduled to attend, and the physical location of the meetings must be assessed against the budgeted travel allocations for the safety function. This activity (resource allocation) becomes complicated if priorities are not established "up-front" on the determination of meetings to support. Once priorities are established, meetings that cannot be supported due to budget constraints can be communicated to program management for the purpose of concurrence, or the reallocation of resources with program management direction.

C.6.4 SAFETY TOOLS AND TRAINING

Planning the system safety engineering and management activities of a program should include the specific tools and training that are required for the accomplishment of the program. Individual training requirements should be identified for each member of the safety team that is specifically required for program-specific tasks and for the personal and professional growth of the analyst. Once the required training is identified, planning must be accomplished to secure funding and to inject the training into the program schedule where most appropriate for impact minimization.

Specific tools that may be required to support the system safety activities are, in many instances, program specific. Examples may include a fault-tree analysis program to identify single-point failures and to quantify their probability of occurrence, or a sneak-circuit analysis tool for sneak-circuit analysis. As wide and varied as individual or programmatic needs are, there is one specific requirement that must be budgeted for a safety database. A safety database is required to provide the audit trail necessary for the identification, documentation, tracking, and resolution of hazards and failure modes in the design, development, test and operation of the system. This database should be "flexible" enough to document the derived requirements of the hazards analysis, create specific reports for design engineering, program management and the customer, and document the traceability and verification methodology for the requirements implemented in the design of the system. Traceability includes the functional, physical, and/or logical link between the requirements, the hazard it was derived from, testing requirements to verify the requirement and the specific modules of code that are affected. In addition, the safety database must be able to store the necessary information normally included in HARs specified by MIL-STD-882 and their associated DIDs. Examples of these reports include the PHA, SSHA, SHA, O&SHA, and SARs.

Supplemental tools that may be required for the performance of software safety tasks include software timing analysis tools, CASE tools, functional or data flow analysis tools, or requirements traceability tools. The safety manager must discuss the requirements for software- and safety-related tools with the software development manager to determine their possible availability from within the program.

C.6.5 REQUIRED HARDWARE AND SOFTWARE

Specific hardware and software resources should be identified when planning for the accomplishment of the SSP. Specific tools may be program dependent to include such things as emulators, models and simulations, requirement verification tools, and/or timing/state analysis tools. The expenditures for required hardware and software must be identified, and planned for, up-front (as possible) on the program. Hardware includes the necessary computer hardware (i.e., Central Processing Unit (CPU), monitor, hard drives, printers, etc.) required to support specific software requirements (i.e., FTA software, hazard tracking database, etc.). Development activities can be unique and may require program-specific hardware or software. This planning activity ensures that the safety manager considers all hardware and software requirements that will be required to fulfill the program objectives.

C.7 PROGRAM PLANS

System safety engineering and management activities, although firmly defined in the SSPP, must be thoroughly addressed as they interface with other managerial or technical disciplines. Although not specifically authored by the system safety manager or the software safety team, numerous program plans must have a system safety input for completeness. This safety input assures that formal lines of communication, individual responsibilities and accountabilities, specific safety tasks, and process interfaces are defined, documented, and agreed upon by all functional or domain disciplines that are affected. Having each technical and functional discipline performing to approved and (well) coordinated plans increases the probability of successfully meeting the goals and objectives of the plan. This *is* concurrent, and systems engineering at its best.

Examples of specific program plans that require system safety input are (but not limited to) the Risk Management Plan (RMP), Quality Assurance Plan (QAP), Reliability Engineering Plan (REP), SDP, SEMP, and TEMP. The system safety manager must assess the program management activities to identify other plans that may require interface requirements and inputs. Complete and detailed descriptions can be found in systems design and engineering standards and textbooks. However, your individual program or system development documentation should contain the best descriptions that apply specifically to the system being developed.

C.7.1 RISK MANAGEMENT PLAN

The RMP describes the programmatic aspects of risk planning, identification, assessment, reduction, and management to be performed by the developer. It should relate the developers' approach for handling risk as compared with the options available. Tailoring of the plan should reflect those program areas that have the greatest impact potential. This may be programmatic, technical (including safety), economical, and/or political risk to the program design, development, test, or operations activities. The plan should describe in detail how an iterative risk assessment process is applied at all WBS levels for each identified risk as the design progresses and matures. It should also describe how the risk assessment in used in the technical design review process, configuration control process, and performance monitoring activities. In most instances, and on most programs, safety risk is a subset of technical risk. The risk assessment and risk management activities of the program must include safety inputs on critical issues for informed decision making purposes by both program management and engineering. Safety-critical and safety-significant issues that are not coming to expected or acceptable closure through the defined safety resolution process must be communicated to the risk management group through systems engineering.

In the area of software safety, the following areas of risk must be considered within the RMP:

- Costs associated with the performance of specific software safety analysis tasks. This must include the costs associated with obtaining the necessary training to perform the tasks,
- Schedule impact associated with the identification, implementation, test and verification of safety-critical software requirements,

- Technical risk associated with the failure of identification of safety-critical design requirements early in the design lifecycle, and
- Risk ramifications associated with the failure to implement safety-critical design requirements, or the implementation of incorrect requirements.

C.7.2 QUALITY ASSURANCE PLAN

"Without exception, the second most important goal must be product **quality**." [STSC 1994] Hopefully, implied from this initial quote, one would assume that **safety** is still the most important goal of a program. Regardless, a quality process is ensured by strict adherence to a systems engineering approach to development for both hardware and software systems. QA is a planned and systematic set of actions required to provide confidence that adequate technical requirements are established, products and services conform to established technical requirements, and satisfactory performance is achieved. It includes the qualitative and quantitative degree of excellence in a product. This can only be achieved if there is excellence in the process to produce the product.

The QAP identifies the processes and process improvement methodologies for the development activities. It focuses on requirements identification and implementation (not design solutions), design activities to specifically meet design specifications and requirements, testing to verify requirements, and maintenance and support of the produced product. Safety input to the QA Plan must focus on the integration of safety into the definition of quality of the product to be produced. Safety must become a function of product quality. The safety manager must integrate safety requirement definitions, implementation, tests, and verification methods and processes into the quality improvement goals for the program. This will include any software safety certifications required by the customer.

Guidance on planning SQA and preparing SQA plans can be found in IEEE STD 730-1989 and IEEE STD 983-1986.

C.7.3 RELIABILITY ENGINEERING PLAN

System safety hazards analysis is heavily influenced by reliability engineering data. A sound reliability engineering activity can produce information regarding component failure frequency, design redundancy, sneak circuits, and subsystem and system failure modes. This information has safety impact on design. The REP describes in detail the planning, process, methods, and the scope of the planned reliability effort to be performed on the program. Dependent on the scope (breadth and depth) of the SSP, much of the reliability data produced must be introduced and integrated into the system safety analysis. An example is the specific failure modes of a subsystem, the components (whose failure causes the failure mode) and the criticality of the failure consequence. This information assists in the establishment and refinement of the hazard analysis and can produce the information required in the determination of probability of occurrence for a hazard. Without reliability data, the determination of probability becomes very subjective.

The safety manager must determine whether a sufficient reliability effort is in place that will produce the information required for the system safety effort. If the development effort requires a strict quantification of hazard risk, there must be sufficient component failure data available to meet the scope objectives of the safety program. Numerous research and development projects produce unique (one-of-a-kind) components. If this is the case, reliability engineers can produce forecasts and reliability predictions of failures and failure mechanisms (based on similar components, vendor data, qualification testing, and modeling techniques) which supplements safety information and increases the fidelity of the safety analysis.

Within the discipline of software safety, the reliability-engineering plan must sufficiently address customer specifications regarding the failure reliability associated with safety-critical or safety-significant software code. The plan must address in detail:

- Specific code that will require statistical testing to meet user or system specifications,
- Any perceived limitations of statistical testing for software code,
- Required information to perform statistical testing,
- Methods of performing the statistical tests required to meet defined confidence levels,
- Specific test requirements,
- Test design and implementation, and
- Test execution and test evaluation.

C.7.4 SOFTWARE DEVELOPMENT PLAN

Software-specific safety requirements have little hope of being implemented in the software design if the software developers do not understand the rationale for safety input to the software development process. Safety managers must communicate and educate the software development team on the methods and processes that produces safety requirements for the software programmers and testers. Given the fact that most software developers were not taught that a safety interface was important on a software development program, this activity becomes heavily dependent upon personal salesmanship. The software development team must be *sold* on the utility, benefit, and the logic, of safety producing requirements for the design effort. This can only be accomplished if the software development team is familiar with the system safety engineering process which identifies hazards and failure mode which are either caused by or influenced by software inputs or information produced by software.

MIL-STD-498 introduces the software development team to the system safety interface. The wording exists in the standard to implement an initial effort. This interface must be communicated and matured in the SDP. "The SDP (usually submitted in draft form with the offeror's RFP response) is the key software document reflecting the offeror's overall software development approach. It must include resources, organization, schedules, risk identification and management, data rights, metrics, QA, control of non-deliverable computer resources, and identification of COTS, reuse, and government-furnished software (GFS) the offeror intends to

use. SDP quality and attention to detail is a major source selection evaluation criterion." [STSC, 1994].

A well structured and highly disciplined software development process, and software engineering methodology, helps to facilitate the development of safer software. This is the direct result of sound engineering practices. However, the development of software that specifically meets the safety goals and objectives of a particular design effort must be supplemented with system safety requirements that eliminate or control system hazards and failure modes caused by (or influenced by) software. The SDP must describe the software development/system safety interface and the implementation, test, and verification methods associated with safety-specific software requirements. This includes the methodology of implementing generic safety requirements and guidelines (see Appendix D) and derived safety requirements from hazard analyses. The plan must address the methodology and design, code, and test protocol associated with safety-critical software, safety-significant software, or lower safety risk modules of code. This defined methodology must be in concert methods and process identified and described in the SEMP, SSPP, and SwSPP.

C.7.5 SYSTEMS ENGINEERING MANAGEMENT PLAN

System safety engineering is an integral part of the systems engineering function. The processes and products of the SSP must be an integrated subset of the systems engineering effort.

The SEMP is the basic document governing the systems engineering effort. It is a concise, top-level, technical management plan consisting of System Engineering Management (SEM) and the Systems Engineering Process. "The purpose of the SEMP is to make visible the organization, direction, control mechanisms, and personnel for the attainment of cost, performance, and schedule objectives." [DSMC, 1990] The SEMP should contain; engineering management procedures and practices of the developer; definition of system and subsystem integration requirements and interfaces; relationships between engineering disciplines and specialties; and reflect the tailoring of documentation and technical activities to meet specific program requirements and objectives.

A further breakdown of the SEMP contents includes:

1. **Technical Program Planning and Control**
   a. Program Risk Analysis
   b. Engineering Program Integration
   c. Contract Work Breakdown
   d. Assessment of Responsibility
   e. Program Reviews
   f. Technical Design Reviews
   g. Technical Performance Measurement
2. **Systems Engineering Process**
   a. Functional Analysis
   b. Requirements Allocation
   c. Trade Studies
   d. Design Optimization and Effective Analysis

e. Synthesis
f. Technical Interface Compatibility
g. Logistics Support Analysis
h. Producibility Analysis
i. Generation of Specifications
j. Other Systems Engineering Tasks

3. Engineering Specialties and Integration of Requirements

a. Reliability
b. Maintainability
c. Human Engineering
d. System Safety
e. Standardization
f. Survivability/Vulnerability
g. Electromagnetic Compatibility
h. Electro-Magnetic Pulse (EMP) Hardening
i. ILS
j. Computer Resources Life Cycle Management
k. Producibility
l. Other Engineering Specialty Requirements

The SEMP must define the interface between systems, design, and system safety engineering (to include software safety). There must be an agreement between engineering disciplines on the methods and processes that identify, document, track, trace, test, and verify subsystem and system requirements to meet the system and user specifications. Therefore, the SEMP must describe how requirements will be categorized. From a safety engineering perspective, this includes the categorization of safety-critical requirements and the tracking, design, test, and verification methods for assurance that these requirements are implemented in the system design. Not only must they be verified to exist, but that the *intent* of the requirement is sufficiently incorporated in the design. The lead systems engineer must assist and support the safety engineering and software-engineering interface to ensure that the hardware and software design meet safety, reliability, and quality requirements.

C.7.6 TEST AND EVALUATION MASTER PLAN

Ensuring that safety is an integral part of the test process is a function that should be thoroughly defined in the TEMP. There are three specific aspects of safety that must be addressed.

First, that the test team consider and implement test constraints, bounds, or limitations based on the safety risks identified by the hazards analysis. Test personnel and test management must be fully informed regarding the safety risk they assume/accept during pre-test, test, and post-test activities.

Second, that a specific safety assessment is accomplished for the testing to be accomplished. This assessment/analysis would include the hazards associated with the test environment, the man/machine/environment interfaces, personnel familiarization with the product, and the resolution of hazards (in real-time) that are identified by the test team which were not identified

or documented, in design. The pre-test assessment should also identify emergency back-out procedures, GO/NO-GO criteria, and emergency response plans. It should also identify personal observation limitations and criteria to minimize hazardous exposure to test team personnel or observers.

And last, that the test activities include specific objectives to verify safety requirements identified in the design hazard analysis and provided in the generic requirements and guidelines documentation. The safety engineer must ensure that test activities and objectives reflect the necessary requirement verification methods to demonstrate hazard mitigation and/or control the levels of acceptable risk defined in the analysis. All safety requirement activities and test results must be formally documented in the hazard record for closure verification.

C.7.7 SOFTWARE TEST PLAN

The STP addresses the software developer's approach and methods to testing. This includes necessary resources, organization, and test strategies. Software development includes in its definition (for the context of the STP), new software development, software modifications, reuse, re-engineering, maintenance, and all other activities resulting in software products. The STP must also include the schedule and system integration test requirements. DID DI-IPSC-81427 required to support MIL-STD-498, *Software Development and Documentation*, describes in detail the contents and format of the STP. It should be noted within the contents of the DID that the test developer must describe in the STP, the method or approach for handling safety-critical (or safety-significant) requirements. The software safety engineering input to the STP should assist in the development of this specific approach. This is required to adjust software safety assessments schedule, resources, and delivery of safety test procedures.

STP review(s), to support the development of the STP, should commence not later than PDR to facilitate early planning for the project.

Software safety inputs to the STP must include:

- Safety inputs to testing requirements (especially those relating to safety-specific requirements). This includes test bounds, assumptions, limitations, normal/abnormal inputs, and expected/anticipated results,
- Safety participation in pretest, test, and post-test reviews,
- Requirements for IV&V and regression testing, and
- Acceptance criteria to meet safety goals, objectives, and requirements.

C.7.8 SOFTWARE INSTALLATION PLAN

The Software Installation Plan (SIP) is a plan that addresses the installation of the developed software at the user site. This plan should address the conversion from any existing system, site preparation, and the training requirements and materials for the user. There is minimum safety interface with the development of this plan, except in the area of safety-related training requirements.

Specific safety training is inherent in controlling residual risk not controlled by design activities. Safety risk that will be controlled by training must be delineated in the SIP. In addition, specific safety inputs should be a part of regular field upgrades where safety interlocks, warnings, training or other features have been changed. This is especially true in programs that provide "annual updates."

C.7.9 SOFTWARE TRANSITION PLAN

The software transition plan identifies the hardware, software, and other resources required for deliverable support of the software product. It describes the developer's plan for the smooth transition from the developer to the support agency or contractor. Included in this transition is the delivery of the necessary tools, analysis, and information required to support the delivered software. From a safety perspective, the developer has the responsibility to identify all software design, code, and test activities that were in the development process that had safety implication. This would include the analysis that identified the hazards and failure modes that were influenced or caused by software. The transition package should include the hazard analysis and the hazard tracking database that documented the software specific requirements and traced them to both the affect module(s) of code and to the hazard or failure mode that derived the requirement. Ramifications of not delivering this information during the transition process is the introduction of unidentified hazards, failure modes, and/or safety risk at the time of software upgrades, modifications, or requirement changes. This is particularly important if the code is identified as safety-critical, or becomes safety-critical due to the proposed change. See the Appendix regarding CM for further information regarding this issue.

C.8 HARDWARE AND HUMAN INTERFACE REQUIREMENTS

C.8.1 INTERFACE REQUIREMENTS

The interface analysis is a vital part of the SHA. It also plays a role in the analysis to ensure that software requirements are not circumvented by other subsystems, or within its own subsystem by other components.

The software interfaces will be traced into, from, and within the safety-critical software functions of a subsystem. These interfaces will be analyzed for possible hazards, and the summary of these interfaces and their interaction, or any safety function shall be assessed. Interface addresses of safety-critical functions will be listed and searched to identify access to and from non-safety-critical functions and/or shared resources.

Interfaces to be analyzed include; functional; physical, including the Human/Machine Interface (HMI); and zonal. Typical hazard analysis activities associated with the accomplishment of a SHA include functional and physical interface analysis. Zonal interfaces (especially on aircraft design) can become safety-critical also. Using aircraft designs as an example, there exists the potential of hazards in the zonal interfaces. These zones include, but are not limited to fire compartments, dry-bay compartments, engine compartments, fuel storage compartments, avionics bay, cockpit, etc. Certain conditions that are considered hazardous in one zone may not be hazardous in another. The important aspect of the SHA activity is to ensure each functional,

physical, and zonal interface is analyzed and the hazards documented. Requirements derived from this analysis are then documented in the hazard record and communicated to the design engineering team.

Before beginning this task, definitions regarding the processes using an interface must be defined. This should also include the information passing that interface which affects safety. The definitions of processes are similar to most human factor studies. An example is the Critical Task Analysis (CTA). The CTA assesses which data passes the interface that is critical to the task. Once the data is identified, the hardware that presents the data, and the software that conditions the hardware and transmission, is examined.

It is recommended that as much preliminary interface analysis (as practical) be accomplished as early in the development lifecycle as possible. This allows preliminary design considerations to be assessed early in the design phase. It should be reiterated that requirements are less costly and more readily accepted by the design team if identified early in the design phases of the program. Although many interfaces are not identified in the early stages of design, the safety engineer can recommend preliminary considerations if these interfaces are given consideration during PHA and SSHA activities. From a software safety perspective, interfaces between software, hardware, and the operator (most likely) will contain the highest safety risk potential. These interfaces must be thoroughly analyzed and safety risk minimized.

C.8.2 OPERATIONS AND SUPPORT REQUIREMENTS

Requirements to minimize the safety risk potential are also identified during the accomplishment of the O&SHA. These requirements are identified as they apply to the operations, support, and maintenance activities associated with the system. The requirements are also, and usually, categorized in terms of protective equipment, safety and warning devices, and procedures and training. At this later phase of the development lifecycle, it is extremely difficult to initiate system design changes to eliminate a potential hazard. If this is even an option, it would be a formal configuration change and require an ECP approved by the Configuration Control Board (CCB). If the hazard is serious enough, an ECP is a viable option. However, as previously stated, formal changes to a “frozen” design become extremely expensive to the program.

Those hazards identified by the O&SHA and hazards previously identified (and not completely eliminated by design) by the PHA, SSHA, SHA, and HHA are risk-minimized to a lower HRI by protective equipment, safety warning devices, and procedures and training. A great percentage of these safety requirements affect the operator and the maintainer (the HMI).

C.8.3 SAFETY/WARNING DEVICE REQUIREMENTS

Safety devices and warning devices are also used within the system and in test, operation, and maintenance activities that must be identified during the hazard analysis process. As with the requirements identified for protective equipment, procedures, and training, these requirements should be identified during the concept exploration phase of the program and refined and finalized during the final phases of design. If the original hazard cannot be eliminated or controlled by design changes, safety and warning devices are considered. This will minimize the safety risk, hopefully, to acceptable levels of the HRI matrix.

C.8.4 PROTECTIVE EQUIPMENT REQUIREMENTS

Another function of the O&SHA is to identify special protective equipment requirements for the protection of personnel (test, operator, or maintainer), the equipment and physical resources, and the environment. This can be as extensive as a complicated piece of equipment for a test cell, and as simple as a respirator for an operator. Each hazard identified in the database should be analyzed for the purpose of further controlling safety risk to the extent feasible with the resources available. The control of safety risk should also be as extensive to meet programmatic, technical, and safety goals established in the planning phases of the program

C.8.5 PROCEDURES AND TRAINING REQUIREMENTS

The implementation (setup), test, operation, maintenance, and support of a system requires system-specific procedures and training for the personnel associated with each activity. Environmental Safety and Health (ESH) issues for personnel and the protection of physical program resources and natural resources, facilitate the necessity for safety requirements. Safety requirements that influence the system test, operating, maintenance, and support procedures and personnel training can be derived as early as the PHA and carried forward for resolution and verification in the O&SHA. It is the responsibility of the safety analyst (with the test and ILS personnel) to incorporate the necessary safety inputs to procedures and training documentation.

C.9 MANAGING CHANGE

The old adage “nothing is constant except change” applies to software after the system is developed. Problems encountered during system-level IV&V, and operational testing account for a small percentage of the overall changes. Problems or errors found by the user account for an additional percentage. However, the largest numbers of changes are the result of upgrades, updates, and pre- (or un-) planned product enhancements. Managing change from a safety perspective requires that the SSS Team assess the potential impact of the change to the system. If the change is to correct an identified safety anomaly or the change potentially impacts the safety of the system, the software systems safety assessment process must rely on the analyses and tests previously conducted.

C.9.1 SOFTWARE CONFIGURATION CONTROL BOARD

CM is a system management function wherein the system is divided into manageable physical or functional configurations and grouped into CIs. The CCB controls the design process through the use of management methods and techniques, including identification, control, status accounting, and auditing. CM (Figure C.5) of the development process and products within that process is established once the system has been divided into functional or physical configuration items. The CCB divides any changes into their appropriate classes (Class I or Class II) and ensures that the proper procedures are followed. The expressed purpose of this function is to ensure that project risk is not increased by the introduction of changes by unauthorized, uncontrolled, poorly coordinated, or improper changes. These changes could directly or indirectly affect system safety and, therefore, require verification, validation, and assessment scrutiny.

The CCB assists the PM, design engineer, support engineers, and other acquisition personnel in the control and implementation of Class I and Class II changes. Class I changes are those which affect form, fit, or function and require user concurrence prior to developer implementation. Class II changes are those not classified as Class I. Examples of Class II changes include editorial changes in documentation or material selection changes in hardware.

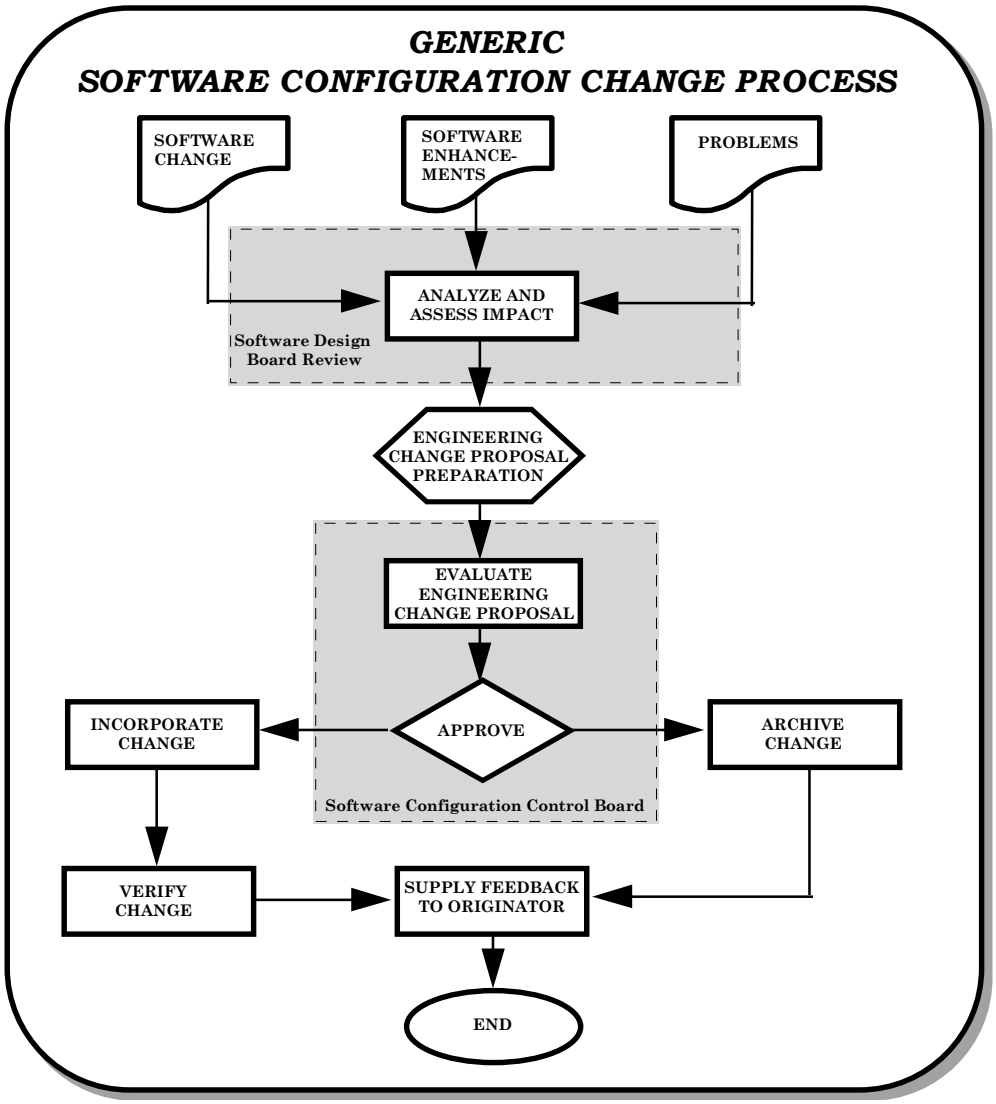


Figure C.5: Generic Software Configuration Change Process

The CCB ensures that the proper procedures for authorized changes to the CI or related products or interfaces are followed and that risk is not increased by the change. The CCB should also ensure that any intermediate step that may halt and expose the project to increased safety risk while halted is controlled. The system safety assessment regarding a configuration change must include:

- Thorough review of the proposed change package ECPs prepared by the engineer responsible for the change,

- Effects of the proposed change on subsystem and system hazards previously identified. This is to include existing and new functional, physical, or zonal interfaces,
- Determination as to whether the proposed change introduces new hazards to the system or to its operations and support functions,
- Determination as to whether the proposed change circumvents existing (or proposed) safety systems, and
- Analysis of all hardware/software and system/operator interfaces.

The SSS Team follows much the same process, on a smaller scale, as they followed during system development. The analyses will have to be updated and appropriate tests re-accomplished, particularly the safety tests related to those portions of the software being modified. The development of the change follows the ECP process complete with the configuration control process. The overall process follows through and concludes with a final safety assessment of the revised product. It is important to remember that some revalidation of the safety of the entire system may be required depending on the extent of the change. One very important aspect of managing change is the change in system functionality.

This includes the addition of new functionality to a system that adds safety-critical functions to the software. For example, if the software developed for a system did not contain safety-critical functions in the original design yet the modifications add new functionality that is safety-critical, the software safety effort will have to revisit a great deal of the original software design to assess its safety risk potential. The software safety analysts will have to revisit both the generic safety design requirements and the functionally derived safety requirements to determine their applicability in light of the proposed software change. Where the tailoring process determined that certain generic requirements were not applicable, the rationale will have to be examined and the applicability re-determined. It is very easy for the PA to try to argue that the legacy software is safe and the new functionality requires that it be the only portion examined. Unless very high-quality, software-engineering standards were followed during the original development, it will be very difficult for the safety analyst to ensure that the legacy software cannot adversely impact the new functionality. Again, the process used is much the same as it was for the original software development.

D. COTS AND NDI SOFTWARE

D.1 INTRODUCTION

The safety assessment of systems requires detailed knowledge of the system components and their interactions. This applies to hardware and software. Only through the analysis of these components and their interactions can the safety analyst identify and evaluate hazard causal factors. Hazard causal factors include errors (e.g., operator errors, design errors, implementation errors, etc.) or failures that are elements of a hazardous condition. By themselves, they generally do not cause hazards, rather, they require other conditions to result in a hazard. Unique military hardware and software allows the safety analyst the insight to the design specifics permitting him/her to determine these hazard causal factors. By recommending design requirements and implementations that reduced the occurrence of the hazard causal factors, the safety analyst reduces the probability of occurrence of hazards that he/she cannot eliminate through design selection recommendations. However, without detailed knowledge, the analyst must work at a higher level of abstraction minimizing his/her ability to control the hazard causal factors. In many instances, the analyst will not be able to provide any direct control of the causal factors and thus must rely on other techniques for hazard risk reduction. Generally, as the level of abstraction increases, the availability of practical and economic control decreases.

Risk reduction in systems employing Non-Developmental Item (NDI) varies with the degree of abstraction of the non-development components. For legacy components with adequate documentation, the level of abstraction may be sufficient to allow the analyst to develop causal factor controls. He/she can incorporate features that mitigate specific causal factors in the legacy components. Similarly, Government Off-The-Shelf (GOTS) components (hardware or software) may provide the necessary detailed documentation to allow the safety analyst to develop causal factor controls. Even if the developer cannot modify these components, the insight allows the analyst to determine the causal factors and develop recommendations for controls in the application software design to preclude their causing or influencing a hazard. COTS and CDI generally pose the real problem; the lack of documentation precludes the analyst from performing the detailed analyses necessary to determine the causal factors related to these items. In addition, the complexity of interactions between some NDI software (e.g., OSs) and the application software (i.e., software unique to the system) makes it extremely difficult to determine the causal factors to the required level of detail. This prevents the analyst from developing specific design recommendations to mitigate the causal factors.

A note of caution to the reader; do not assume that it is only the NDI that is related to the fielded system of interest or concern. Computers, workstations, and software development tools used as part of the process are also a concern and can have significant safety implications. Similarly, the use of simulators (including stubs) and emulators used during development or as adjuncts to the fielded system may also have safety implication. For example, software for a Weapon Control System (WCS) proceeds through development using a specific set of CASE tools including a compiler. The software safety analysis and testing demonstrates that the software executes in the system context with an acceptable level of risk. However, prior to deployment, the developers make minor changes to correct non-safety-related problems and recompile the software using a

new compiler. Although the developer made no changes to the safety-related portions of the software, the recompiled software executes differently due to differences in the compilers. These differences can impact the safety of the system in an operating environment. The validation testing performed on the system actually validates the object code executing on the system: not the design or the source code. System safety must therefore perform regression testing on the recompiled software to ensure that changes to the object code do not adversely affect the safety performance of the software in the system context.

D.2 RELATED ISSUES

D.2.1 MANAGING CHANGE

Technology refreshments and upgrades may introduce hazard causal factors and thus require special attention by system safety. Many developers provide upgrades, enhancements, or error corrections to commercial software products on a routine basis. In addition, developers often offer patch updates via the Internet. Microprocessors and single board computers change less frequently, however they both change much faster than their earlier military-unique counterparts. The pace of change is increasing as manufacturing techniques improve. One commercial software developer reportedly can introduce revisions in "shrink-wrap" commercial products within four hours of approval. Developers often do not change version numbers or provide other indications of minor upgrades or error corrections. Although a packaged product may indicate a certain version number, it may include patches and updates that earlier versions, perhaps even those sitting on the same shelf, did not include.

The products used in the system are not the only items subject to change. Compilers, software development tools, and other items used during the development process also change frequently. System developers often assume that they may recompile code using a new compiler with few, if any problems. However, experience demonstrates otherwise, especially for systems upgrading to new processors and OSs simultaneously. System safety must ensure that the effort includes sufficient safety testing and analysis to verify that the changes do not influence known hazards or adversely affect the residual risk of the system. Processor or OS changes may invalidate interlocks, firewalls, timing constraints, or other hazard mitigation implemented in the design. Therefore, system safety must address these issues as part of a delta analysis to the system that may include revisiting code-level analyses of the application software.

D.2.2 CONFIGURATION MANAGEMENT

Managing change introduces the subject of CM, a significant issue in safety-critical applications of COTS software and hardware. If the developer performs sufficient analysis and testing to verify that a specific version of a COTS program will not adversely affect the safety of the system, that certification applies only to the specific version of the COTS program tested. Unfortunately, as noted above, the system developer is often not aware of changes inserted into the COTS software. Therefore, unless he/she releases only the tested version to the user as part of the system and uses only that version for future releases, the safety certification for that software is invalid. System developers must obtain agreements from COTS software to alert

them of any changes made to the packaged software used. This rule applies to software that is part of the system as well as software used to develop the system.

D.2.3 REUSABLE AND LEGACY SOFTWARE

Reusable software, as its name implies, is software from a previous program or system used in place of uniquely developed software to reduce costs. The term generally applies to software developed for a broad range of similar systems or applications, such as avionics software. The software may or may not meet all of the needs of the system under development or it may include functionality not required by the system, therefore requiring modification. Generally, the system developers do not modify the software to remove the unnecessary functionality. The software safety analyst must analyze the software proposed for reuse in the system context to determine what its role and functionality will be in the final system. He/she must perform this analysis regardless of whether or not the software was safety "qualified" in its previous application since the safety-criticality of the software is application dependent. Thus, the reuse of software does not ease the software safety program burden.

D.3 APPLICATIONS OF NON-DEVELOPMENTAL ITEMS

Applications of NDI software include OSs and environments, communications handlers, interface handlers, network software, database managers, data reduction and analysis tools, and a variety of other software components that are functionally part of the system. Indirect applications of NDI include programming languages, compilers, software development (e.g., CASE) tools, and testing tools that indirectly affect the applications software in the fielded system.

D.3.1 COMMERCIAL-OFF-THE-SHELF SOFTWARE

The safety assessment of COTS software poses one of the greatest challenges to the safety certification of systems. COTS software is generally developed for a wide range of applications in the commercial market. Developers use an internal company or industry standard, such as IEEE, ANSI, or National Institute for Standards and Technology for software development. In general, the company or the project team determines the language used. Since the vendor releases only compiled versions of the product, there is often no way to determine that language. Because the COTS developers can only deduce at the applications for the product, they cannot address specific issues related to a particular application. However, the developer attempts to ensure that the product is compatible with many system and software configurations. This often results in additional functionality that, in it self may be hazardous.

An excellent example of a COTS related issue in a safety-critical system occurred with a major blood databank. The databank used COTS database management and network software tailored to the company's needs. The databank stores information regarding individual units of blood including blood type and whether or not the blood has any infectious diseases. Two separate laboratories tested samples for blood type and for diseases. However, each laboratory had testing responsibility for different diseases. The databank operated safely for several years when a problem occurred: both laboratories accessed the same record simultaneously to enter their test

results. The simultaneous access resulted from reduced blood donations meaning the laboratories had fewer blood samples to test. Laboratory A tested for several diseases, including HIV, in the blood samples, noted the results in the database record, and saved the record. Meanwhile, Laboratory B discovered infectious hepatitis in a blood sample, noted the fact in the same record, and saved it, overwriting laboratory A's findings. Freezing methods can destroy infectious hepatitis therefore, the company believed the blood safe for use after processing. Thus, whether or not the blood had the HIV was unknown. Fortunately, the company discovered the error prior to distribution of the blood. Note the various causal factors in this example. Tainted blood is an obvious hazard causal factor outside the company's direct control. Therefore, they required the blood screening process (hazard control). Accessing the same record simultaneously is a second causal factor. The probability of this occurring was very low when blood donations were high. Laboratory B overwriting the record from laboratory A is a third causal factor. Finally, various aspects of blood collection and distribution are conditions that, in combination with the causal factors, would lead to a hazard if a patient received tainted blood.

The blood databank example illustrates how the application of a NDI program affects safety issues. The database management program designer could not anticipate all applications of the program and design it to preclude the event described above. Originally, he/she developed the database manager for a single computer application without considering networked applications and therefore could not anticipate that two users accessing a record simultaneously would lead to a hazard. Likewise, the network program designer could not anticipate such a hazard either. He/she would rely on the database management program to provide the necessary records security.

In the conduct of a SSP, the safety team analyzes the system to identify the undesired events (mishaps), the hazards that provide the preconditions for the mishap, and the potential causal factors that may lead to those hazards. By eliminating or reducing the probability of occurrence of the causal factors, the analyst reduces the probability of the hazard and the overall mishap risk associated with the system. As noted earlier, the safety analyst requires detailed knowledge of the system design and its intended use to eliminate or reduce the risk of hazards. COTS software poses several problems in this regard. Generally, vendors provide only user documentation from the commercial software developers. The type of documentation necessary to conduct detailed analyses is usually not available. The developer may not even generate high-level specifications, FFDs, DFDs, or detailed design documents for a given commercial software package. The lack of detailed documentation limits the SwSE to identifying system hazard causal factors related the COTS software at a high level. He/she has no ability to develop and implement design requirements or analyze the design implementation within the COTS software to eliminate or reduce the probability of occurrence of these causal factors.

Occasionally, system developers may purchase the required documentation from the COTS developer who often charge premium prices for it. However, its availability provides the safety team the detailed knowledge of the design to identify the hazard causal factors and design specific interlocks in the applications software to preclude their occurrence. The detailed analysis and testing of the COTS product with the application software poses both cost and schedule risks. The project team must determine if these costs and risks are greater than

implementing the other recommendations for reducing the risk of safety-critical applications of COTS.

Testing of the COTS software in this application is very limited in its ability to provide evidence that the software cannot influence system hazards. Testing in a laboratory cannot duplicate every nuance of the operational environment nor can it duplicate every possible combination of events. Based on their knowledge of failures and operational errors of the software design, test engineers can develop procedures to test software paths. Even when the developer knows the design and implementation of the software in detail, many constraints still apply. The testing organization, like the safety organization, must still treat COTS software as a "black box," developing tests to measure the response of the software to input stimulus under (presumably) known system states. Hazards identified through "black box" testing are sometimes happenstance and difficult to duplicate. Timing issues and data senescence issues also are difficult to fully test in the laboratory environment even for software of a known design. Without the ability to analyze the code, determining potential timing problems in the code is difficult at best. Without detailed knowledge of the design of the software, the system safety and test groups can only develop limited testing to verify the safety and fail-safe features of the system.

Like commercially available software, commercially available hardware also possesses limitations that make safety certification of software applications difficult in most cases. Companies design commercial systems, such as workstations, to industry standards that are much less stringent than the military standards. Any number of vendors may supply parts or subsystems with little control over the quality or conformance to requirements other than those imposed by the developer. As with COTS software, the vendor generally does not document the firmware embedded in the COTS hardware in a manner that can be analyzed by the user. In addition, high-level, detailed documentation on the hardware design may not be available for review and analysis. Therefore, it is easy to draw corollaries between system safety issues related to COTS software and COTS hardware. Open architecture systems employing COTS software and NDI hardware pose safety concerns requiring different solutions than existing system architectures.

D.4 REDUCING RISKS

The system developer has a number of options for addressing the risk associated with the application of NDI software in safety-critical systems. The first, and unfortunately most obvious is to ignore it, i.e., treat it as “trusted” software. This option virtually guarantees that a hazard will occur at some point in the system’s lifecycle. The system development group must employ techniques to reduce the risk of NDI software in safety-critical applications.

D.4.1 APPLICATIONS SOFTWARE DESIGN

The straightforward approach is to design the application software for any eventuality. However, this is often difficult, particularly if the developers are not aware of the full range of functionality of the NDI software. It requires the safety analyst to identify all of the potential causal factors and ensure that the applications software design will respond in a safe manner. Often this adds significant complexity to the software and will likely reduce its availability. However, the safety

analyst must thoroughly evaluate any change to the NDI software and ensure that the applications software changes correspondingly to mitigate any risks. This process can be extremely expensive.

One technique related to this issue is under investigation by the system safety organization at Lockheed-Martin in Syracuse, New York, is the use of discriminators. Discriminators are characteristics of the NDI product that are potentially safety-critical in relation to the applications software. The developer uses discriminators to determine when changes to the NDI software may impact the applications software. This process appears viable on paper, however in practical application it has proven much more difficult than expected. One reason for this difficulty is the determination of changes that indirectly impact the safety-critical functionality of the software. To a large degree, the ability of unrelated software to impact the functionality that influences the safety-critical aspects of the applications software depends directly on the quality of the software development process used by the NDI developer. If the developer maintained high quality software engineering practices, including information hiding, modularization, and weak coupling, the likelihood that unrelated software can impact the functionality that influences safety-critical portions of the applications software is greatly reduced. This same paradigm applies to any safety-critical software.

D.4.2 MIDDLEWARE OR WRAPPERS

An effective method of reducing the risk associated with NDI software applications is to reduce its influence on the safety-critical functions in the system. This requires isolation (e.g., firewalls) of the NDI software from the safety-critical functions. Isolation from an OS requires a layer of software, often called middleware, between the applications software and the OS. The middleware interfaces to both the OS and the applications software. All interactions between the applications software and the OS take place through the middleware. The middleware implementation simplifies the design of the applications software by eliminating the need to provide robustness to change in the OS interface: that robustness becomes part of the middleware. The developer can specify and maintain a detailed interface design between the middleware and the applications software. The developer must then provide the necessary robustness for the OS in the middleware. The middleware also contains the exception and interrupt handlers necessary for safe system operation. The middleware may contain macros and other programs used by numerous modules in the applications software rather than relying on those supplied by the NDI products.

Isolation of safety-critical functions from the NDI software may require application-specific software “wrappers” on either side of the NDI software effectively isolating two software packages. This technique is similar in nature to the middleware discussed above but is much more specific to a particular function within the system. The wrapper will perform the necessary sanity checks and isolation to preclude hazards’ occurrence. The safety analyst must consider the risk associated with NDI during the PHA phase of the program. The identification of potential causal factors associated with NDI (these need not be too specific) helps identify the need for such wrappers. By identifying these causal factors, the analyst can develop design requirements for the wrapper. Again, these need not be specific. The safety analyst knows what the safety-

critical software needs and expects to see. The wrapper simply prevents anything else from "getting through".

The use of isolation techniques also has limitations. One limitation is when interfaces to external components pass through NDI software. For example, a system may use a hard disk to store and retrieve safety-critical data. The NDI OS provides the handler and data transfer between the disk and the applications software. For devices such as data storage and retrieval, the developer must ensure that the devices selected for the system provide data integrity checks as part of that interface. Another limitation is the amount of software required to achieve adequate isolation. If the applications software has interactions with the NDI software in numerous functional areas, the complexity of the middleware and/or the number of wrappers required may make them impractical or cost prohibitive.

Wrappers and middleware are effective techniques for isolating the safety-critical functionality. If identified early in the system development process they are also cost effective although they do invoke costs for the development and testing of this specialized software. Another benefit is that when properly designed, the wrappers and middleware reduce the impact of changes in the NDI products from both the system safety and system maintainability perspective. Wrappers and middleware can be relatively simple programs that the developer can readily modify to accommodate changes in the NDI software. However, as noted above, when the interactions are complex in nature, the middleware and wrappers can become complex as well.

D.4.3 MESSAGE PROTOCOL

A technique for isolating safety-critical data from NDI software (OSs, network handlers, etc.) is to package all communications in a robust manner. Specifying a communications protocol that provides unique identification of the message type and validation of the correct receipt of the data transfer will ensure that the NDI products to not adversely affect the safety-critical data. The degree of robustness required depends on the criticality of the data. For highly critical data, a message protocol using Cyclic Redundancy Checks (CRC), Fletcher Checksums, or bit-by-bit comparisons provides a very high degree of assurance that safety-critical data passed between system components is correct. Less robust data checks include arithmetic and linear check sums and parity checks coupled with well-defined message structures. The middleware may incorporate the message handler, including the CRC or checksum software, thus offloading that functionality from the applications software. The benefits of this approach are that it is relatively easy and cost effective if implemented early in the system design. However, like all other aspects of system design, late identification of these requirements results in a significant cost impact.

D.4.4 DESIGNING AROUND IT

A technique often used is to design around the CDI software. Embedding exception and interrupt handlers in the applications software ensures that the application software maintains control of the system. However, it is generally not possible to wrestle control for all exceptions or interrupts from the OS and environment or the compiler. Micro-code interrupt handlers embedded in the microprocessor BIOS often cannot be supplanted. Attempting to supplant these handlers is likely to be more hazardous than relying on them directly. However, careful attention

to the design of the applications software and its interaction with the system when these interrupts occur can mitigate any related hazards.

The system developer can design the applications software to ensure robustness in its interface to the OS and environment. The robustness of the design depends on the ability of the analyst to identify potential failure modes and changes to the OS or environment. For complex systems, the time and resources required to identify all of the failure modes and potential changes is very high. The additional complexity in the applications software may also introduce hazard causal factors into the design that were not present in the simpler design. Many other aspects of designing around CDI software are beyond the scope of this Appendix. The need for design features depends directly on the CDI software in the system and the interactions between the CDI software and the applications software. Message protocol and watchdog timers are other examples of designing around CDI software.

D.4.5 ANALYSIS AND TESTING OF NDI SOFTWARE

The system developer may have access to detailed design documentation on the NDI products. The availability varies from product to product but is generally very expensive. However, the detailed analysis of the NDI software and its interactions with the applications software provides the greatest degree of assurance that the entire software package will execute safety in the system context. Analysis of the NDI products allows the development of directed testing of the application software in the NDI environment to determine if identified causal factors will indeed result in an undesired condition. However, it may be difficult to generate failure modes in NDI software without actually inserting modifications. Earlier paragraphs discussed this option and noted that the project team must evaluate the cost of procuring this documentation against the cost of other options. A portion of the decision must consider the potential consequences of a safety-critical failure in the system.

D.4.6 ELIMINATING FUNCTIONALITY

Eliminating unnecessary functionality from OSs and environments reduces the risk that these functions will corrupt safety-critical functions in the applications software. Some functionality, such as file editors, is undesirable in safety-critical applications. One US Navy program retained an OS's file editor to allow maintenance personnel to insert and test patches and perform software updates. Unfortunately, the users discovered this capability and used it to resolve problems they were having with the system. One of the problems resolutions discovered by the sailors also overrode a safety-interlock in the system that could have inadvertently launched a weapon. Although an inadvertent launch did not occur, the potential for its occurrence was very high. It may not be possible, and occasionally even risky, to eliminate functions from OSs or environments. Generally, one eliminates the functionality by preventing certain modules from loading. However, there may be interactions with other software modules in the system not obvious to the user. This interdependency, particularly between apparent unrelated system modules, may cause the software to execute unpredictably or to halt. If the NDI developer designed the product well, these interdependencies will be minimal.

D.4.7 RUN-TIME VERSIONS

Some OSs and environments have different development and run-time versions. The run-time versions already have unnecessary functionality removed. This allows the use of the full version on development workstations. The execution and testing use only those functions available in the run-time version thus allowing the developers more flexibility during the design. Some of these systems, such as VxWorks® are rapidly becoming the OS of choice for command and control, fire control, and weapon control systems. System developers should consider the availability of run-time versions when selecting an OS or environment.

D.4.8 WATCHDOG TIMERS

The purpose of a watchdog timer is to prevent processors from entering loops that, for whatever reason, go on indefinitely, or to exit processing that takes longer than expected. Applications software can also use similar timers for safety-critical timing constraints. Watchdog timers issue an interrupt to the processor after a pre-determined time. A command from the applications software resets the timer to its pre-determined value each cycle. Software designers must not embed the reset command within a loop, other than the main execution loop in the program. In the design and implementation of the watchdog timer, the safety engineer addresses several issues. The watchdog timer processing should return safety-critical outputs and external system components to a safe state. Often, when a processor enters an infinite loop, the processor state and hence the system state is non-deterministic. Therefore, the safety engineer has no assurance that the system is in a safe state. The watchdog timer must be independent of the processing, i.e., not be an imbedded function within the processor software unless that processing is completely independent of the timing loop it is monitoring. The safety engineer should determine those processes that may adversely affect the safety of the system should their execution time go beyond a pre-determined time. An example of such a process is a call to another subroutine that obtains a safety-critical value from an external source. However, the external source data is not available and the subroutine waits for the external source to provide it. If a delay in this data causes a data senescence problem in the safety-critical process, the program should interrupt the subroutine waiting on the data and return to a safe state. That safe state may simply be to refresh stale data or it may require the software actively change the state of external system components. The watchdog timer may perform this function for a small system or the design may implement secondary watchdog timers specifically for this type of function.

D.4.9 CONFIGURATION MANAGEMENT

Once the system developer determines the NDI software for the system, they should attempt to maintain configuration control over that software just as they do over the applications software. For commercially obtained items, this may require an agreement between the vendor and the developer. Even if a NDI supplier is unwilling to provide notification of changes to the product, the system developer can establish procedures to detect changes and determine their potential impact. In the simplest case, detection may use file comparison programs to compare two or more copies of a product. However, these generally only detect a difference without providing any indication of the changes made.

D.4.10 PROTOTYPING

Although prototyping is not directly related to the application of NDI software in safety-critical systems, some of the benefits derived from this development methodology apply to the safety assessment of the application software in the NDI environment. Rapid prototyping allows the safety analyst to participate in the "build-a-little, test a little" process by analyzing and testing relatively small portions of the system functionality each time. They can then identify safety issues and incrementally incorporate them into the design. The safety analyst also begins to build an analytical picture of the interactions between the NDI software and the applications software. This allows the development and integration of risk mitigation techniques on a real-time basis.

D.4.11 TESTING

Testing at the functional and system levels also helps evaluate the risk associated with NDI in safety-critical systems. However, there are numerous limitations associated with testing. Some of these are that it is impossible to examine all possible paths, conditions, timing, and data senescence problems, create all possible failure conditions and modes, and cause the system to enter every state machine configuration that may occur in the operational environment. Many other limitations to testing for safety that apply to any developmental system apply to the NDI environment as well. Earlier paragraphs in this Appendix addressed even more limitations associated with testing. As with safety testing in general, the analyst can develop directed tests that focus on the safety-critical aspects of the interaction of the NDI software with the applications software. However, the ability to introduce failure modes is very limited in this environment. Testers must have a set of benchmark (regression) tests that evaluate the full spectrum of the application software's safety-critical functionality when evaluating changes to the NDI software.

D.5 SUMMARY

The techniques discussed in this Appendix will not reduce the residual mishap risk associated with systems employing NDI unless they are part of a comprehensive SSP. A single technique may not be adequate for highly critical applications and the system developer may have to use several approaches to reduce the risk to an acceptable level. Early identification of hazards and the development and incorporation of safety design requirements into the system design are essential elements to a cost-effective risk reduction process. The analysis of their implementation in the design ensures that the software design meets the intent of the requirements provided. Early identification of the requirements for isolation software will provide a cost-effective approach to addressing many of the potential safety issues associated with NDI applications.

Unfortunately, there are no silver bullets to resolve the safety issues with safety-critical applications of NDI just as there are none for the software safety process in general. The techniques all require time and effort and involve some level of programmatic risk.

This Handbook discusses the various techniques for reducing the risk of NDI applications in the context of safety-critical functions in the applications software. However, these same techniques

are equally applicable to other desirable characteristics of the system, such as mission effectiveness, maintainability and testability.

E. GENERIC REQUIREMENTS AND GUIDELINES

E.1 INTRODUCTION

The goal of this Appendix is to provide generic safety design requirements and guidelines for the design and development of systems that have or potentially have safety-critical applications. These requirements and guidelines are designed that, if properly implemented, they will reduce the risk of the computing system causing an unsafe condition, malfunction of a fail safe system, or non-operation of a safety function. These requirements and guidelines are not intended to be used as a checklist but in conjunction with safety analyses performed in accordance with applicable standards and directives and must be tailored to the system or system type under development. These requirements and guidelines must also be used in conjunction with accepted high quality software engineering practices including configuration control, reviews and audits, structured design, and related systems engineering practices.

E.1.1 DETERMINATION OF SAFETY-CRITICAL COMPUTING SYSTEM FUNCTIONS

The guidelines of this Appendix are to be used in determining which computing system functions are safety-critical. Specific identification of safety-critical computing systems should be done using the safety assessment requirements or similar techniques.

E.1.1.1 SPECIFICATIONS

The required safety functions of the computing system shall be determined from the analysis of the system and its specifications. These computing system safety functions are to be designated Safety-Critical Computing System Functions (SCCSFs).

E.1.1.2 SAFETY-CRITICALITY

A hazard analysis of the risks associated with the specified functions of the computing system shall be made to reduce the potentially harmful effects of both random and systemic (design) failures in such systems. The functions associated with a potentially unacceptable level of risk to the system user; other personnel; third parties; and, if appropriate, other facilities, equipment, and the environment in general are to be designated SCCSFs.

E.1.1.3 LIKELY SCCSFS

- Any function which controls or directly influences the pre-arming, arming, enabling, release, launch, firing, or detonation of a weapon system, including target identification, selection and designation,
- Any function that determines, controls, or directly influences the flight path of a weapon system,

- Any function that controls or directly influences the movement of gun mounts, launchers, and other equipment, especially with respect to the pointing and firing safety of that equipment,
- Any function which controls or directly influences the movement of munitions and/or hazardous materials,
- Any function which monitors the state of the system for purposes of ensuring its safety,
- Any function that senses hazards and/or displays information concerning the protection of the system,
- Any function that controls or regulates energy sources in the system,
- Fault detection priority. The priority structure of fault detection and restoration of safety or correcting logic shall be considered safety-critical. Software units or modules handling or responding to these faults,
- Interrupt processing software. Interrupt processing software, interrupt priority schemes and routines that disable or enable interrupts,
- Autonomous control. Software components that have autonomous control over safety-critical hardware,
- Software controlled movement. Software that generates signals which have been shown through analysis to directly influence or control the movement of potentially hazardous hardware components or initiate safety-critical actions,
- Safety-critical displays. Software that generates outputs that displays the status of safety-critical hardware systems. Where possible, these outputs shall be duplicated by non-software generated output, and
- Critical data computation. Software used to compute safety-critical data. This includes applications software that may not be connected to or directly control a safety-critical hardware system (e.g., stress analysis programs).

E.2 DESIGN AND DEVELOPMENT PROCESS REQUIREMENTS AND GUIDELINES

The requirements and guidelines of this section apply to the design and development phases.

E.2.1 CONFIGURATION CONTROL

Configuration control shall be established as soon as practical in the system development process. The Software CCB prior to their implementation must approve all software changes occurring after an initial baseline has been established. A member of the Board shall be tasked with the responsibility for evaluation of all software changes for their potential safety impact. This member should be a member of the system safety engineering team. A member of the hardware CCB shall be a member of the software CCB and vice versa to keep members apprised

of hardware changes and to ensure that software changes do not conflict with or introduce potential safety hazards due to hardware incompatibilities.

E.2.2 SOFTWARE QUALITY ASSURANCE PROGRAM

A SQA program shall be established for systems having safety-critical functions. SQA is defined as follows: "Assurance is the confidence based upon objective evidence, that the risk associated with using a system conforms with our expectation of, or willingness to tolerate, risk." There are several issues that should be addressed by the program office to calibrate confidence in the software. There is consensus in the software development community that no one assurance approach (i.e., one "confidence building measure") is adequate for critical software assurance, and that some integration of the evidence provided by these various approaches must be used to make decisions about confidence.

E.2.3 TWO PERSON RULE

At least two people shall be thoroughly familiar with the design, code, testing, and operation of each software module in the system.

E.2.4 PROGRAM PATCH PROHIBITION

Patches shall be prohibited throughout the development process. All software changes shall be coded in the source language and compiled prior to entry into operational or test equipment.

E.2.5 SOFTWARE DESIGN VERIFICATION AND VALIDATION

The software shall be analyzed throughout the design, development, and maintenance process by a system safety engineering team to verify and validate that the safety design requirements have been correctly and completely implemented. Test results shall be analyzed to identify potential safety anomalies that may occur.

E.2.5.1 CORRELATION OF ARTIFACTS ANALYZED TO ARTIFACTS DEPLOYED

A great deal of the confidence placed in a critical software system is based upon the results of tests and analyses performed on specific Artifacts produced during system development (e.g., source code modules, executable programs produced from the source code). The results of such tests and analyses contribute to confidence in the deployed system only to the extent that we can be sure that the tested and analyzed components, and only them, are actually in the deployed system. There have been many instances where the "wrong version" of a component has accidentally been introduced into a deployed system and as a result caused unexpected failures or at least presented a potential hazard.

- Does the SDP describe a thorough CM process that includes version identification, access control, change audits, and the ability to restore previous revisions of the system?
- Does the CM process rely entirely on manual compliance, or is it supported and enforced by tools?

- Does the CM process include the ability to audit the version of specific components (e.g., through the introduction of version identifiers in the source code that are carried through into the executable object code)? If not, how is process enforcement audited (i.e., for a given executable image, how can the versions of the components be determined)?

- Is there evidence in the design and source code that the CM process is being adhered to (e.g., Are version identifiers present in the source code if this is part of the CM process described)?

- During formal testing, do any problems with inconsistent or unexpected versions happen?

A second issue that affects confidence of correlation between the artifacts analyzed and those deployed are "tool integrity." Software tools (i.e., computer programs used to analyze, transform, or otherwise measure or manipulate products of a software development effort) clearly can have an impact on the level of confidence placed in critical software. All of the analysis of source code performed can easily be undermined if we discover that the compiler used on the project is very buggy, for example. In many situations where this is a potential issue (e.g., the certification of digital avionics), a distinction is drawn between two classes of tools:

- Those that transform the programs or data used in the operational system; (and can therefore actively introduce unexpected behavior into the system), and

- Those used to evaluate the system (and therefore can at worst contribute to not detecting a defect).

Clearly there is a limit to how many resources should be applied to, for example, validating the correctness of an Ada compiler; in fact, a developer may reasonably argue that until there is evidence of a problem, it is sufficient mitigation to use widely-used commercially available tools, for example. Still, the program office should have some confidence that the developer is addressing these issues.

- Does the SDP and/or risk reduction plan include a description of tool qualification criteria and plans? Does the plan include a description of what the critical tools are (e.g., compiler, linker loader) and what the risk mitigation approach is (e.g., use widely available commercial compilers, establish good support relationship with vendor, canvass other users of the tools for any know problems)?

- Is there any evidence in the design documentation or source code of "work-arounds" being introduced to accommodate problems encountered in critical tools? If so, what steps are being taken to ensure that the problems are fixed and that these problems should not indicate a general reduced confidence in the tools?

E.2.5.2 CORRELATION OF PROCESS REVIEWED TO PROCESS EMPLOYED

- Confidence in the process used to develop critical software is a key part of overall confidence in the final system. However, that confidence is of course justified only if there is reason to believe that the process described is the process applied. The program office can use milestone reviews as a way to deliberately audit process enforcement and

adherence to the processes described. The use of static analysis and inspection of artifacts (e.g., design documentation, source code, test plans) can provide increased confidence that the process is being adhered to (or expose violations of the described process, which should be given immediate attention - are they an indication that the process is not being enforced?).

- Are the processes as described in the SDP enforceable and auditable? Specific coding standards or testing strategies can be enforced and they can be independently audited by a review of the products; vague or incomplete process descriptions can be very hard to enforce and to determine if they are being adhered to (which reduces the confidence they provide with respect to critical software assurance arguments).
- As the development progresses, what is the overall "track record" of compliance with the processes described in the SDP (as determined by audits of compliance during milestone reviews)? If there is reason for concern, this should become a separate topic for resolution between the program office and the developer.
- How does the DA monitor and enforce process compliance by the subcontractors? Is there evidence that this is being done?

E.2.5.3 REVIEWS AND AUDITS

Desk audits, peer reviews, static and dynamic analysis tools and techniques, and debugging tools shall be used to verify implementation of design requirements in the source code with particular attention paid to the implementation of identified safety-critical computing system functions and the requirements and guidelines provided in this document. Reviews of the software source code shall ensure that the code and comments within the code agree.

E.3 SYSTEM DESIGN REQUIREMENTS AND GUIDELINES

The requirements and guidelines of this section apply to the general system design.

E.3.1 DESIGNED SAFE STATES

The system shall have at least one safe state identified for each logistic and operational phase.

E.3.2 STANDALONE COMPUTER

Where practical, safety-critical functions should be performed on a standalone computer. If this is not practical, safety-critical functions shall be isolated to the maximum extent practical from non-critical functions.

E.3.3 EASE OF MAINTENANCE

The system and its software shall be designed for ease of maintenance by future personnel that are not associated with the original design team. Documentation specified for the computing system shall be developed to facilitate maintenance of the software. Strict configuration control

of the software during development and after deployment is required. The use of techniques for the decomposition of the software system for ease of maintenance is recommended.

E.3.4 SAFE STATE RETURN

The software shall return hardware subsystems terms under the control of software to a designed safe state when unsafe conditions are detected. Conditions that can be safely overridden by the battle short shall be identified and analyses performed to verify their safe incorporation.

E.3.5 RESTORATION OF INTERLOCKS

Upon completion of tests and/or training wherein safety interlocks are removed, disabled or bypassed, restoration of those interlocks shall be verified by the software prior to being able to resume normal operation. While overridden, a display shall be made on the operator's or test conductor's console of the status of the interlocks, if applicable.

E.3.6 INPUT/OUTPUT REGISTERS

Input/output registers and ports shall not be used for both safety-critical and non-critical functions unless the same safety design criteria are applied to the non-critical functions.

E.3.7 EXTERNAL HARDWARE FAILURES

The software shall be designed to detect failures in external hardware input or output hardware devices and revert to a safe state upon their occurrence. The design shall consider potential failure modes of the hardware involved.

E.3.8 SAFETY KERNEL FAILURE

The system shall be designed such that a failure of the safety kernel (when implemented) will be detected and the system returned to a designated safe state.

E.3.9 CIRCUMVENT UNSAFE CONDITIONS

The system design shall not permit detected unsafe conditions to be circumvented. If a "battleshort" or "safety arc" condition is required in the system, it shall be designed such that it cannot be activated either inadvertently or without authorization.

E.3.10 FALLBACK AND RECOVERY

The system shall be designed to include fallback and recovery to a designed safe state of reduced system functional capability in the event of a failure of system components.

E.3.11 SIMULATORS

If simulated items, simulators, and test sets are required, the system shall be designed such that the identification of the devices is fail safe and that operational hardware cannot be inadvertently identified as a simulated item, simulator or test set.

E.3.12 SYSTEM ERRORS LOG

The software shall make provisions for logging all system errors detected. The operator shall have the capability to review logged system errors. Errors in safety-critical routines shall be highlighted and shall be brought to the operator's attention as soon as practical after their occurrence.

E.3.13 POSITIVE FEEDBACK MECHANISMS

Software control of critical functions shall have feedback mechanisms that give positive indications of the function's occurrence.

E.3.14 PEAK LOAD CONDITIONS

The system and software shall be designed to ensure that design safety requirements are not violated under peak load conditions.

E.3.15 ENDURANCE ISSUES

Although software does not "wear out," the context in which a program executes can degrade with time. Systems that are expected to operate continuously are subjected to demands for endurance - the ability to execute for the required period of time without failure. As an example of this, the failure of a Patriot missile battery in Dhahran during the Persian Gulf War was traced to the continuous execution of tracking and guidance software for over 100 hours; the system was designed and tested against a 24-hour upper limit for continuous operation. Long-duration programs are exposed to a number of performance and reliability problems that are not always obvious and that are difficult to expose through testing. This makes a careful analysis of potential endurance-related defects an important risk-reduction activity for software to be used in continuous operation.

- Has the developer explicitly identified the duration requirements for the system? Has the developer analyzed the behavior of the design and implementation if these duration assumptions are violated? Are any of these violations a potential hazard?
- Has the developer identified potential exposure to the exhaustion of finite resources over time, and are adequate detection and recovery mechanisms in place to handle these? Examples are as follows:
  - ✓ Memory (e.g., heap leaks from incomplete software storage reclamation),
  - ✓ File handles, Transmission Control Protocol ports, etc. (e.g., if not closed under error conditions), and
  - ✓ Counter overflow (e.g., 8-bit counter and > 255 events was a factor in the failure of Theriac-25 radiation treatment machines).
- Has the developer identified potential exposure to performance degradation over time, and are adequate deduction and recovery mechanisms in place to handle these? Examples are memory and disk fragmentation that can result in increased latency.

- Has the developer analyzed increased exposure to cumulative effects over time, and are adequate detection and recovery mechanisms in place to handle these so that they do not present any hazards? Examples include cumulative drift in clocks, cumulative jitter in scheduling operations, and cumulative rounding error in floating point and fixed-point operations.

E.3.16 ERROR HANDLING

Causal analyses of software defects frequently identify error handling as a problem area. For example, one industry study observed that a common defect encountered was "failure to consider all error conditions or error paths." A published case study of a fault tolerant switching system indicated that approximately two thirds of the system failures that were traceable to design faults were due to faults in the portion of the system that was responsible for detecting and responding to error conditions. The results of a Missile Test and Readiness Equipment (MlTRE) internal research project on Error Handling in Large Software Systems also indicate that error handling is a problematic issue for many software systems. In many cases, the problems exposed were the result of oversight or simple logic errors. A key point is that these kinds of errors have been encountered in some software that is far along in the development process and/or under careful scrutiny because it is mission critical software. The presence of simple logic errors such as these illustrates the fact that error handling is typically not as carefully inspected and tested as other aspects of system design. It is important that the program office gain adequate insight into the developer's treatment of error handling in critical systems.

- Has the developer clearly identified an overall policy for error handling? Have the specific error detection and recovery situations been adequately analyzed? Has the developer defined their relationship between exceptions, faults, and "unexpected" results?

- Are different mechanisms used to convey this status of computations? What are they? [e.g., Ada exceptions, OS signals, return codes, messages]. If return codes and exceptions are both used, are there guidelines for when each is to be used? What are these guidelines and the rationale for them, and how are they enforced? Are return codes and exceptions used in distinct "layers of abstraction" (e.g., return codes only in calls to COTS OS services) or freely intermixed throughout the application? How are return codes and exceptions mapped to each other? In this mapping, what is done if an unexpected return code is returned, or an unexpected exception is encountered?

- Has the developer determined the costs of using exceptions for their compiler(s)? What is the space and runtime overhead of having one or more exception handlers in a subprogram and a block statement, and is the overhead fixed or a function of the number of handlers? How expensive is propagation, both explicit and implicit?

- Are derived types used? If so, are there any guidelines regarding the exceptions that can be raised by the derived operations associated with the derived types? How are they enforced?

- Are there guidelines regarding exceptions that can be propagated during task rendezvous? How are they reinforced and tested?

- Is program suppression ever used? If so, what are the restrictions on its use, and how are they enforced? What is the rationale for using/not-using program suppression? If it is used, are there any guidelines for explicit checking that must be in the code for critical constraints in lieu of the implicit constraint checks? If not, how is the reliability of the code ensured?

- Are there any restrictions on the use of tasks in declarative regions of subprograms (i.e., subprograms with dependent tasks)? If so, how are they enforced? How are dependent tasks terminated when the master subprogram is terminating with an exception, and how is the suspense of exception propagation until dependent task termination handled?

- What process enforcement mechanisms are used to ensure global consistency among error handling components? (e.g., we have seen examples of systems where various subcontractors were under constrained; they each make locally plausible design decisions regarding error handling policy, but when these components were integrated they were discovered to be globally inconsistent.)

- Are there guidelines on when exceptions are masked (i.e., a handler for an exception does not in turn propagate an exception), mapped (i.e., a handler for an exception propagates a different exception), or propagated? If so, how are they enforced? Are there any restrictions on the use of the "others" handlers? If so, how are they enforced?

- How does the developer ensure that return codes or status parameters are checked after every subroutine call, or ensure that failure to check them does not present a hazard?

- Are there any restrictions on the use of exceptions during elaboration? (e.g., checking data passed to a generic package during installation). Is exception handling during elaboration a possibility due to initialization functions in declarative regions? If so, how is this handling tested, and are there design guidelines for exception handling during elaboration? If not, how are they assured that this does not present a hazard?

E.3.17 REDUNDANCY MANAGEMENT

In order to reduce the vulnerability of a software system to a single mechanical or logic failure, redundancy is frequently employed. However, the added complexity of managing the redundancy in fault-tolerant systems may make them vulnerable to additional failure modes that must be accounted for by the developer. For example, the first shuttle flight and the 44th flight of NASA's Advanced Fighter Technology Integration (AFTI)-F16 software both exhibited problems associated with redundancy management. The first shuttle flight was stopped 20 minutes before scheduled launch because of a race condition between the two versions of the software. The AFTI-F16 had problems related to sensor skew and control law gain causing the system to fail when each channel declared the others had failed; the analog backup was not selected, because the simultaneous failure of two channels was not anticipated.

If the developer's design includes redundancy (e.g., duplicate independent hardware, or "N version programming"), have the additional potential failure modes from the redundancy scheme

been identified and mitigated? Examples include sensor skew, multiple inconsistent states, and common mode failures.

E.3.18 SAFE MODES AND RECOVERY

A common design idiom for critical software systems is that they are "self checking and self protecting." This means that software components "protect" themselves from invalid requests or invalid input data by frequently checking for violations of assumptions or constraints. In addition, they check the results of service requests to other system components to make sure that they are behaving as expected. Finally, such systems typically provide for the checking of internal intermediate states to determine if the routine is itself working as expected. Violations of any of these kinds of checks can require transition to a safe state if the failure is serious or if the confidence in further correct execution has been seriously reduced. Failure to address this "defensive" approach can allow a wide variety of failures to propagate throughout the system in unexpected and unpredictable ways, potentially resulting in a hazard.

- Does the developer identify a distinct safe mode or set of safe modes? Has the analysis of these safe modes adequately considered the transition to these safe modes from potentially hazardous states (e.g., internal inconsistency)?
- Does the design include acceptable safety provisions upon detection of an unsafe state?
- Does the design include assertion checks or other mechanisms for the regular run-time calibration of internal logic consistency?
- Does the developer provide for an orderly system shutdown as a result of operator shutdown instructions, power failure, etc.?
- Does the developer explicitly define the protocols for any interactions between the system and the operational environment? If anything other than the expected sequences or interlocks is encountered, does the system design detect this and transition to a safe state?
- Does the developer account for all power-up self-test and handshaking with other components in the operational environment in order to ensure execution begins in a predicted and safe state?

E.3.19 ISOLATION AND MODULARITY

The practical limits on resources for critical software assurance are consistent with the consensus in the software development community that a major design goal for critical software is to keep the critical portions small and isolated from the rest of the system. The program office can evaluate evidence provided by the developer that indicates the extent to which this isolation has been a design goal and the extent to which the implementation has successfully realized this goal. Confidence that unanticipated events or latent defects in the rest of the software will not introduce an operational hazard is in part correlated with the confidence that such isolation has been achieved.

- Does the developer's design provide explicit evidence of an analysis of the criticality of the components and functions (i.e., does the design reflect an analysis of which functions can introduce a hazard)?
- Does the developer's design and implementation provide evidence that in critical portions of the software, coupling has been kept to a minimum (e.g., are there restrictions on shared variables and side-effects for procedures and functions)?
- Does the developer's design include attention to the implementation of "firewalls" in the software - boundaries where propagation of erroneous values is explicitly checked and contained? Do critical portions of code perform consistency checking of data values provided to them both by "clients" (i.e., software using the critical software as a service) and by the software services the critical software calls (e.g., OS services)?
- Does the critical software design and implementation include explicit checks of intermediate states during computation, in order to detect possible corruption of the computing environment (e.g., range checking for an intermediate product in an algorithm)?
- Does the developer provide the criteria for determining what software is critical, and is there evidence that these criteria were applied to the entire software system? How does the developer provide evidence that the portions considered non-critical in fact will not introduce a hazard?

E.4 POWER-UP SYSTEM INITIALIZATION REQUIREMENTS

The following requirements apply to the design of the power subsystem, power control, and power-on initialization for safety-critical applications of computing systems.

E.4.1 POWER-UP INITIALIZATION

The system shall be designed to power up in a safe state. An initialization test shall be incorporated in the design that verifies that the system is in a safe state and that safety-critical circuits and components are tested to ensure their safe operation. The test shall also verify memory integrity and program load.

E.4.2 POWER FAULTS

The system and computing system shall be designed to ensure that the system is in a safe state during power up, intermittent faults or fluctuations in the power that could adversely affect the system, or in the event of power loss. The system and/or software shall be designed to provide for a safe, orderly shutdown of the system due to either a fault or power down, such that potentially unsafe states are not created.

E.4.3 PRIMARY COMPUTER FAILURE

The system that shall be designed such that a failure of the primary control computer will be detected and the system returned to a safe state.

E.4.4 MAINTENANCE INTERLOCKS

Maintenance interlocks, safety interlocks, safety handles, and/or safety pins shall be provided to preclude hazards to personnel maintaining the computing system and its associated equipment. Where interlocks, etc. must be overridden to perform tests or maintenance, they shall be designed such that they cannot be inadvertently overridden, or left in the overridden state once the system is restored to operational use. The override of the interlocks shall not be controlled by a computing system.

E.4.5 SYSTEM-LEVEL CHECK

The software shall be designed to perform a system-level check at power up to verify that the system is safe and functioning properly prior to application of power to safety-critical functions including hardware controlled by the software. Periodic tests shall be performed by the software to monitor the safe state of the system.

E.4.6 CONTROL FLOW DEFECTS

Control flow refers to the sequencing of instructions executed while a program is running. The consequence of defects in control flow may be program termination (e.g., when an Ada exception propagates to the outermost scope, or the program attempts to execute an illegal instruction or to branch to an invalid region of memory). What is more difficult to detect, and more problematic, is that the consequence of a control flow defect may simply be continued execution in an invalid or unpredictable state. For example, a "computed go-to" (e.g., using base and displacement registers in an assembly language program) may branch to a legitimate instruction sequence that is simply not the correct sequence given the current state of the system. Therefore, for a critical system, evidence must be presented that these kinds of defects are avoided or mitigated.

- If the developer is using assembly language, are there any computed control-flow statements? That is, are there any branches or jumps to an address that is computed (e.g., base and displacement registers) rather than a static symbolic label? If so, how does the developer ensure that these address computations never result in a "wild jump," or that such wild jumps do not represent a hazard?

- In Ada functions, there may be paths where control can "fall through." (i.e., the function terminates in a statement other than a return or an exception propagation.) This is an invalid control flow and will result in the propagation of the pre-defined Ada exception Program_Error. How does the developer ensure that either this will not happen, or that the propagation of Program_Error from a function will not represent a hazard?

- If the developer is using the C programming language, is the C facility of passing addresses of functions as arguments ("funargs") used? If so, how does the developer

ensure that all calls to a function pointed to by a funarg are valid, or that no hazards result from invalid funargs?

- Since Ada is used, how does the developer ensure that no exception can propagate to the outermost scope and terminate the program (or how is this dealt with so that such termination is not a hazard)? Are restrictions on exceptions that can be propagated from specific routines present? (e.g., are only a restricted set of exceptions allowed to propagate to a caller?) If there are such restrictions, how are they enforced? If not, how does the developer provide assurance that all potential exceptions are handled?

- A second timing-related failure mode for software is the existence of race conditions: activities that execute concurrently, and for which the result depends on the sequencing of activity. For example, if Ada tasking is used and two tasks concurrently access and change a device, the final result of the computations may depend on which task is the first to access the device (which task "won the race"). In Ada, the scheduling of eligible concurrent tasks is non-deterministic, which means that two consecutive executions with the same data may run differently. Note that the race need not be between tasks; in the fatal software failures of the Theriac-25 radiation treatment devices, one failure mode was a race condition between internal tasks and the operator's timing for screen update and entry. These kinds of failure modes are often difficult to isolate, repeat, and correct once a system has been deployed; they are equally difficult to detect during testing and so have caused some extremely subtle latent defects in deployed software (e.g., the first attempt to launch the Columbia Space Shuttle was aborted 20 minutes before launch due to a latent race condition that had a 1-in-67 chance of being exposed at each system power-up. The program office should look for evidence that all potential hazards resulting from timing and sequencing have been systematically considered, and any hazards that are identified are mitigated.

- Has the developer clearly presented the concurrent requirements (explicit or derived) for the system? Have the timing and sequencing consequences been given significant attention with respect to repeatable behavior and hazard identification and mitigation for the concurrent?

- Has the developer identified all real-time requirements (e.g., for reading sensor data, for interacting with devices, for constraints imposed by other systems the application interacts with)? Would the consequences of failing to meet any of those requirements represent a hazard? If so, what hazard mitigation has the developer used? Note that real-time requirements not only include deadlines, but upper and lower bounds on event timing (e.g., the minimum interval between consecutive packets on a communications channel, or time-out triggers).

- If there are any real-time requirements that are critical (i.e., failing to meet them would present a hazard), how has the developer identified and controlled all sources of unpredictable timing, including upper and lower bounds on device latency (e.g., secondary storage), upper and lower bounds on the timing characteristics of Ada language features that may vary widely (e.g., exception propagation, use of dynamic memory for access types, use of the delay statement, task rendezvous and termination), and upper and

lower bounds on the software' s interaction with other subsystems (e.g., burst mode or failed communications, data rates exceeding or below expected values, time-outs for failed hardware).

- Have all control and data flows where there is potential interference or shared data among multiple threads of control (e.g., Ada tasks, OS processes) or multiple interrupt handlers been identified by the developer? If so, has the developer identified all potential race conditions? How has the developer ensured that either there are no race conditions that could present a hazard or that such hazards are mitigated? Note that in Ada, the interleaved update of shared variables by multiple Ada tasks is erroneous; the results are, therefore, unpredictable.

- Has the developer identified potential hazards resulting from sequencing errors? Even for single threads of control there are potential failure modes related to sequencing errors. For example, in Ada, calling a function before the function body has been elaborated is an example of access before elaboration, and it results in Program_Error being raised. The initial elaboration sequence is an important aspect of program correctness for non-trivial Ada programs, and is an example of the kinds of sequencing failures that the developer should review for identification of possible hazards and the mitigation of any such hazards discovered. Another sequencing failure example is calling an operation before performing any required initialization.

E.5 COMPUTING SYSTEM ENVIRONMENT REQUIREMENTS AND GUIDELINES

The requirements and guidelines of this section apply to the design and selection of computers, microprocessors, programming languages, and memories for safety-critical applications in computing systems.

E.5.1 HARDWARE AND HARDWARE/SOFTWARE INTERFACE REQUIREMENTS

- CPU,
- Memory,
- Failure in the computing environment,
- Hardware and software interfaces,
- Self-test Features,
- Watchdog timers, periodic memory checks, operational checks,
- System utilities,
- Compilers, assemblers, translators, and OSs,
- Diagnostic and maintenance features, and

- Memory diagnosis.

E.5.1.1 FAILURE IN THE COMPUTING ENVIRONMENT

An application program exists in the context of a computing environment - the software and hardware that collectively support the execution of the program. Failures in this environment can result in a variety of failures or unexpected behavior in the application program and, therefore, must be considered in a hazard analysis. For some of these failure modes (e.g., program overwrite of storage), it is particularly difficult to completely predict the consequences (e.g., because it depends on what region is overwritten and what pattern is written there); the burden of proof is, therefore, on the developer to provide evidence either that there is no exposure to these kinds of failure or that such failures do not represent a potential hazard.

- Has the developer identified the situations in which the application can corrupt the underlying computing environment? Examples include the erroneous writing of data to the wrong locations in storage (by writing to the 11th element of a 10 element array, for example, or through pointer manipulation in "C" or unchecked conversion or use of pragma Interface in Ada). Has Ada's pragma Suppress been used? If so, how does the developer ensure that such storage corruption is not being missed by removing the runtime checks? Note that if pragma Suppress is used and the detection of a constraint violation is masked, the results are unpredictable (the program is "erroneous"). Has the developer provided evidence that the software's interaction with the hardware does not corrupt the computing environment in a way that introduces a hazard (e.g., setting a program status word to an invalid state, or sending invalid control sequences to a device controller)?

- Has the developer analyzed potential failure modes of the Ada Runtime Environment (ARTE), the host OS or executive, and any other software components (e.g., Data Base Management System) used in conjunction with the application for any hazards that they could introduce? What evidence does the developer provide that either there are no failure modes that present a hazard or that the identified hazards have been mitigated [e.g., what evidence does the developer provide for the required level of confidence in the ARTE, OS, etc.? (e.g., for commercial avionics certification and other safety-critical domains, high assurance or even "certified" subset ARTEs have been used)].

- Has the developer provided evidence that data consistency management has been addressed adequately where it can affect critical functions? For example, is file system integrity checked at startup? Are file system transactions atomic, or is there a mechanism for backing out from corrupted transactions?

E.5.2 CPU SELECTION

The following guidelines apply to the selection of CPUs:

- CPUs that process entire instructions or data words are preferred to those that multiplex instructions or data (e.g., an 8-bit CPU is preferred to a 4-bit CPU emulating an 8-bit machine).

- CPUs with separate instructions and data memories and busses are preferred to those using a common data/instruction buss. Alternatively, memory protection hardware, either segment or page protection, separating program memory and data memory is acceptable.
- CPUs, microprocessors and computers that can be fully represented mathematically are preferred to those that cannot.

E.5.3 MINIMUM CLOCK CYCLES

For CPUs that do not comply with the guidelines above, or those used at the limits of their design criteria (e.g., at or above maximum clock frequency), analyses and measurements shall be conducted to determine the minimum number of clock cycles that must occur between functions on the buss to ensure that invalid information is not picked up by the CPU. Analyses shall also be performed to ensure that interfacing devices are capable of providing valid data within the required time frame for CPU access.

E.5.4 READ ONLY MEMORIES

Where Read Only Memories (ROM) are used, positive measures shall be taken to ensure that the data cannot be corrupted or destroyed.

E.6 SELF-CHECK DESIGN REQUIREMENTS AND GUIDELINES

The design requirements of this section provide for self-checking of the programs and computing system execution.

E.6.1 WATCHDOG TIMERS

Watchdog timers or similar devices shall be provided to ensure that the microprocessor or computer is operating properly. The timer reset shall be designed such that the software cannot enter an inner loop and reset the timer as part of that loop sequence. The design of the timer shall ensure that failure of the primary CPU clock cannot compromise its function. The timer reset shall be designed such that the system is returned to a known safe state and the operator alerted (as applicable).

E.6.2 MEMORY CHECKS

Periodic checks of memory, instruction, and data buss(es) shall be performed. The design of the test sequence shall ensure that single point or likely multiple failures are detected. Checksum of data transfers and Program Load Verification checks shall be performed at load time and periodically thereafter to ensure the integrity of safety-critical code.

E.6.3 FAULT DETECTION

Fault detection and isolation programs shall be written for safety-critical subsystems of the computing system. The fault detection program shall be designed to detect potential safety-critical failures prior to the execution of the related safety-critical function. Fault isolation

programs shall be designed to isolate the fault to the lowest level practical and provide this information to the operator or maintainer.

E.6.4 OPERATIONAL CHECKS

Operational checks of testable safety-critical system elements shall be made immediately prior to performance of a related safety-critical operation.

E.7 SAFETY-CRITICAL COMPUTING SYSTEM FUNCTIONS PROTECTION REQUIREMENTS AND GUIDELINES

The design requirements and guidelines of this section provide for protection of safety-critical computing system functions and data.

E.7.1 SAFETY DEGRADATION

Other interfacing automata and software shall design the system such that automata and software shall prevent degradation of safety.

E.7.2 UNAUTHORIZED INTERACTION

The software shall be designed to prevent unauthorized system or subsystem interaction from initiating or sustaining a safety-critical function sequence.

E.7.3 UNAUTHORIZED ACCESS

The system design shall prevent unauthorized or inadvertent access to or modification of the software (source or assembly) and object code. This includes preventing self-modification of the code.

E.7.4 SAFETY KERNEL ROM

Safety kernels should be resident in non-volatile ROM or in protected memory that cannot be overwritten by the computing system.

E.7.5 SAFETY KERNEL INDEPENDENCE

A safety kernel, if implemented, shall be designed and implemented in such a manner that it cannot be corrupted, misdirected, delayed, or inhibited by any other program in the system.

E.7.6 INADVERTENT JUMPS

The system shall detect inadvertent jumps within or into SCCSFs; return the system to a safe state, and, if practical, perform diagnostics and fault isolation to determine the cause of the inadvertent jump.

E.7.7 LOAD DATA INTEGRITY

The executive program or OS shall ensure the integrity of data or programs loaded into memory prior to their execution.

E.7.8 OPERATIONAL RECONFIGURATION INTEGRITY

The executive program or OS shall ensure the integrity of the data and programs during operational reconfiguration.

E.8 INTERFACE DESIGN REQUIREMENTS

The design requirements of this section apply to the design of input/output interfaces.

E.8.1 FEEDBACK LOOPS

Feedback loops from the system hardware shall be designed such that the software cannot cause a runaway condition due to the failure of a feedback sensor. Known component failure modes shall be considered in the design of the software and checks designed into the software to detect failures.

E.8.2 INTERFACE CONTROL

SCCSFs and their interfaces to safety-critical hardware shall be controlled at all times, i.e., the interface shall be monitored to ensure that erroneous or spurious data does not adversely affect the system, that interface failures are detected, and that the state of the interface is safe during power-up, power fluctuations and interruptions, and in the event of system errors or hardware failures.

E.8.3 DECISION STATEMENTS

Decision statements in safety-critical computing system functions shall not rely on inputs of all ones or all zeros, particularly when this information is obtained from external sensors.

E.8.4 INTER-CPU COMMUNICATIONS

Inter-CPU communications shall successfully pass verification checks in both CPUs prior to the transfer of safety-critical data. Periodic checks shall be performed to ensure the integrity of the interface. Detected errors shall be logged. If the interface fails several consecutive transfers, the operator shall be alerted and the transfer of safety-critical data terminated until diagnostic checks can be performed.

E.8.5 DATA TRANSFER MESSAGES

Data transfer messages shall be of a predetermined format and content. Each transfer shall contain a word or character string indicating the message length (if variable), the type of data and content of the message. As a minimum, parity checks and checksums shall used for verification

of correct data transfer. CRCs shall be used where practical. No information from data transfer messages shall be used prior to verification of correct data transfer.

E.8.6 EXTERNAL FUNCTIONS

External functions requiring two or more safety-critical signals from the software (e.g., arming of an ignition safety device or arm fire device and release of an air launched weapon) shall not receive all of the necessary signals from a single input/output register or buffer.

E.8.7 INPUT REASONABLENESS CHECKS

Limit and reasonableness checks, including time limits, dependencies, and reasonableness checks, shall be performed on all analog and digital inputs and outputs prior to safety-critical functions' execution based on those values. No safety-critical functions shall be executable based on safety-critical analog or digital inputs that cannot be verified.

E.8.8 FULL SCALE REPRESENTATIONS

The software shall be designed such that the full scale and zero representations of the software are fully compatible with the scales of any digital-to-analog, analog-to-digital, digital-to-synchro, and/or synchro-to-digital converters.

E.9 HUMAN INTERFACE

The design requirements of this section apply to the design of the human interface to safety-critical computing systems.

E.9.1 OPERATOR/COMPUTING SYSTEM INTERFACE

- Computer/Human Interface (CHI) Issues,
- Displays,
- Duplicated where possible, SCCSF displays to be duplicated by non-software generated output, designed to reduce human errors, quality of display, clear and concise,
- Hazardous condition alarms/warnings,
- Easily distinguished between types of alerts/warning, corrective action required to clear,
- Process cancellation,
- Multiple operator actions to initiate hazardous function, and
- Detection of improper operator entries.

E.9.1.1 COMPUTER/HUMAN INTERFACE ISSUES

CHI issues are not software issues per se - they are really a distinct specification and design issue for the system. However, many of the CHI functions will be implemented in software, and CHI issues frequently are treated at the same time as software in milestone reviews.

- Has the developer explicitly addressed the safety-critical aspects of the design of the CHI? Has this included analysis of anticipated single and multiple operator failures? What kind of human factors, ergonomic, and cognitive science analyses were done (e.g., of cognitive overload, ambiguity of display information)?
- Does the design ensure that invalid operator requests are flagged and identified as such to the operator (vs. simply ignoring them or mapping them silently to "correct" values)?
- Does the developer ensure that the system always requires a minimum of two independent commands to perform safety-critical function? Before initiating any critical sequence, does the design require an operator response or authorization?
- Does the developer ensure that there are no "silent mode changes" that can put the system in a different safety-related state without operator awareness (i.e., does the design not allow critical mode transitions to happen with notification)?
- Does the developer ensure that there is a positive reporting of changes of safety-critical states?
- Does the system design provide for notification that a safety function has been executed, and is the operator notified of the cause?
- Are all critical inputs clearly distinguished? Are all such inputs checked for range and consistency validity?

E.9.2 PROCESSING CANCELLATION

The software shall be designed such that the operator may cancel current processing with a single action and have the system revert to a designed safe state. The system shall be designed such that the operator may exit potentially unsafe states with a single action. This action shall revert the system to a known safe state. (e.g., the operator shall be able to terminate missile launch processing with a single action which shall safe the missile.) The action may consist of pressing two keys, buttons, or switches at the same time. Where operator reaction time is not sufficient to prevent a mishap, the software shall revert the system to a known safe state, report the failure, and report the system status to the operator.

E.9.3 HAZARDOUS FUNCTION INITIATION

Two or more unique operator actions shall be required to initiate any potentially hazardous function or sequence of functions. The actions required shall be designed to minimize the potential for inadvertent actuation, and shall be checked for proper sequence.

E.9.4 SAFETY-CRITICAL DISPLAYS

Safety-critical operator displays, legends and other interface functions shall be clear, concise, and unambiguous, and where possible, be duplicated using separate display devices.

E.9.5 OPERATOR ENTRY ERRORS

The software shall be capable of detecting improper operator entries or sequences of entries or operations and prevent execution of safety-critical functions as a result. It shall alert the operator to the erroneous entry or operation. Alerts shall indicate the error and corrective action. The software shall also provide positive confirmation of valid data entry or actions taken (i.e., the system shall provide visual and/or aural feedback to the operator such that the operator knows that the system has accepted the action and is processing it). The system shall also provide a real-time indication that it is functioning. Processing functions requiring several seconds or longer shall provide a status indicator to the operator during processing.

E.9.6 SAFETY-CRITICAL ALERTS

Alerts shall be designed such that routine alerts are readily distinguished from safety-critical alerts. The operator shall not be able to clear a safety-critical alert without taking corrective action or performing subsequent actions required to complete the ongoing operation.

E.9.7 UNSAFE SITUATION ALERTS

Signals alerting the operator to unsafe situations shall be directed as straightforward as practical to the operator interface.

E.9.8 UNSAFE STATE ALERTS

If an operator interface is provided and a potentially unsafe state has been detected, the system shall alert the operator to the anomaly detected, the action taken, and the resulting system configuration and status.

E.10 CRITICAL TIMING AND INTERRUPT FUNCTIONS

The following design requirements and guidelines apply to safety-critical timing functions and interrupts.

E.10.1 SAFETY-CRITICAL TIMING

Safety-critical timing functions shall be controlled by the computer and shall not rely on human input. Safety-critical timing values shall not be modifiable by the operator from system consoles, unless specifically required by the system design. In these instances, the computer shall determine the reasonableness timing values.

E.10.2 VALID INTERRUPTS

The software shall be capable of discriminating between valid and invalid (i.e., spurious) external and/or internal interrupts. Invalid interrupts shall not be capable of creating hazardous conditions. Valid external and internal interrupts shall be defined in system specifications. Internal software interrupts are not a preferred design as they reduce the analyzability of the system.

E.10.3 RECURSIVE LOOPS

Recursive and iterative loop shall have a maximum documented execution time. Reasonableness checks will be performed to prevent loops from exceeding the maximum execution time.

E.10.4 TIME DEPENDENCY

The results of a program should not be dependent on the time taken to execute the program or the time at which execution is initiated. Safety-critical routines in real-time programs shall ensure that the data used is still valid (e.g., by using senescence checks).

E.11 SOFTWARE DESIGN AND DEVELOPMENT REQUIREMENTS AND GUIDELINES

The requirements and guidelines of this section apply to the design and coding of the software.

E.11.1 CODING REQUIREMENTS/ISSUES

The following applies to the software-coding phase.

- Language issues,
  - ✓ Ada, C++,
- Logic errors,
- Cumulative data errors,
- Drift in clocks, round-off errors,
- Specific features/requirements,
- No unused executable code, no unreferenced variables, variable names and declaration for SCFs, loop entry/exits, use of annotation within code, assignment statements, conditional statements, strong data typing, ban of global variables for SCFs, safety-critical files
- All safety-critical software to occupy same amount of memory,
- Single execution path for safety-critical functions,
- No unnecessary/undocumented features,

- No bypass of required system functions, and
- Prevention of runaway feedback loops.

E.11.1.1 ADA LANGUAGE ISSUES

The Ada programming language provides considerable support for preventing many causes of unpredictable behavior allowed in other languages. For example, unless pragma Suppress or unchecked conversion (and certain situations with pragma Interface) are used, implicit constraint checks prevent the classic "C" programming bug of writing a value into the 11th element of a 10-element array (thus overwriting and corrupting an undetermined region of memory, with unknown results that can be catastrophic). However, the Ada language definition identifies specific rules to be obeyed by Ada programs but which no compile-time or run-time check is required to enforce. If a program violates one of these rules, the program is said to be erroneous. According to the language definition, the results of executing an erroneous program are undefined and unpredictable. For example, there is no requirement for a compiler to detect the reading of uninitialized variables or for this error to be detected at run-time. If a program does execute such a use of uninitialized variables, the effects are undefined: the program might raise an exception (e.g., Program_Error, Constraint_Error), or simply halt, or some random value may be found in the variable, or the compiler may have a pre-defined value for references to uninitialized variables (e.g., 0). For obvious reasons, the overall confidence that the program office has in the predictable behavior of the software will be seriously undermined if there are shown to be instances of "erroneous" Ada programs for which no evidence is provided that they do not present a hazard. There are several other aspects of the use of Ada that can introduce unpredictable behavior, timing, or resource usage, while not strictly erroneous.

- Are all constraints static? If not, how are the following sources of unpredictable behavior shown to prevent a hazard: Constraint_Error raised?
- Use of unpredictable memory due to elaboration of non-static declarative items,
- For Ada floating point values, are the relational operators "<", '>", ""=, and "/=" precluded? Because of the way floating point comparisons are defined in Ada the values of the listed operators depend on the implementation. "<=" and ">=" do not depend on the implementation, however. Note that for Ada floating point it is not guaranteed that, for example, "X <= Y" is the same as "not (X>Y)". How are floating point, operations ensured to be predictable or how is the lack of predictability shown to not represent a hazard by the developer?
- Does the developer use address clauses? If so, what restrictions are enforced on the address clauses to prevent attempts to the overlay of data, which results in an erroneous program?
- If Ada access types are used, has the developer identified all potential problems that can result with access types (unpredictable memory use, erroneous programs if Unchecked_Deallocation is used and there are references to a deallocated object, aliasing,

unpredictable timing for allocation, constraint checks) and provided evidence that these do not represent hazards?

- If pragma Interface is used, does the developer ensure that no assumptions about data values are violated in the foreign language code that might not be detected upon returning to the Ada code (e.g., passing a variable address to a C routine that violates a range constraint - this may not be detected upon return to Ada code, enabling the error to propagate before detection)?
- Does the developer ensure that all out and in out mode parameters are set before returning from a procedure or entry call unless an exception is propagated, or provide evidence that there is no case where returning with an unset parameter (and therefore creating an erroneous program) could introduce a hazard?
- Since Ada supports recursion, has the developer identified restrictions on the use of recursion or otherwise presented evidence that recursion will not introduce a hazard (e.g., through exhaustion of the stack, or unpredictable storage timing behavior)?
- Are any steps taken to prevent the accidental reading of an uninitialized variable in the program [through coding standards (defect prevention) and code review or static analysis (defect removal)]? Does the developer know what the selected compiler's behavior is when uninitialized variables are referenced? Has the developer provided evidence that there are no instances of reading uninitialized variables that introduce a hazard, as such a reference results in an erroneous program?
- If the pre-defined Ada generic function Unchecked_Conversion is used, does the developer ensure that such conversions do not violate constraints of objects of the result type, as such a conversion results in an erroneous program?
- In Ada, certain record types and private types have discriminants whose values distinguish alternative forms of values of one of these types. Certain assignments and parameter bindings for discriminants result in an erroneous program. If the developer uses discriminants, how does he ensure that such erroneous uses do not present a hazard?

E.11.2 MODULAR CODE

Software design and code shall be modular. Modules shall have one entry and one exit point.

E.11.3 NUMBER OF MODULES

The number of program modules containing safety-critical functions shall be minimized where possible within the constraints of operational effectiveness, computer resources, and good software design practices.

E.11.4 EXECUTION PATH

SCCSFs shall have one and only one possible path leading to their execution.

E.11.5 HALT INSTRUCTIONS

Halt, stop or wait instructions shall not be used in code for safety-critical functions. Wait instructions may be used where necessary to synchronize input/output, etc. and when appropriate handshake signals are not available.

E.11.6 SINGLE PURPOSE FILES

Files used to store safety-critical data shall be unique and shall have a single purpose. Scratch files, those used for temporary storage of data during or between processes, shall not be used for storing or transferring safety-critical information, data, or control functions.

E.11.7 UNNECESSARY FEATURES

The operational and support software shall contain only those features and capabilities required by the system. The programs shall not contain undocumented or unnecessary features.

E.11.8 INDIRECT ADDRESSING METHODS

Indirect addressing methods shall be used only in well-controlled applications. When used, the address shall be verified as being within acceptable limits prior to execution of safety-critical operations. Data written to arrays in safety-critical applications shall have the address boundary checked by the compiled code.

E.11.9 UNINTERRUPTABLE CODE

If interrupts are used, sections of the code which have been defined as uninterruptable shall have defined execution times monitored by an external timer.

E.11.10 SAFETY-CRITICAL FILES

Files used to store or transfer safety-critical information shall be initialized to a known state before and after use. Data transfers and data stores shall be audited where practical to allow traceability of system functioning.

E.11.11 UNUSED MEMORY

All processor memory not used for or by the operational program shall be initialized to a pattern that will cause the system to revert to a safe state if executed. It shall not be filled with random numbers, halt, stop, wait, or no-operation instructions. Data or code from previous overlays or loads shall not be allowed to remain. (Examples: If the processor architecture halts upon receipt of non-executable code, a watchdog timer shall be provided with an interrupt routine to revert the system to a safe state. If the processor flags non-executable code as an error, an error handling routine shall be developed to revert the system to a safe state and terminate processing.) Information shall be provided to the operator to alert him to the failure or fault observed and to inform him of the resultant safe state to which the system was reverted.

E.11.12 OVERLAYS OF SAFETY-CRITICAL SOFTWARE SHALL ALL OCCUPY THE SAME AMOUNT OF MEMORY

Where less memory is required for a particular function, the remainder shall be filled with a pattern that will cause the system to revert to a safe state if executed. It shall not be filled with random numbers, halt, stop, no-op, or wait instructions or data or code from previous overlays.

E.11.13 OPERATING SYSTEM FUNCTIONS

If an OS function is provided to accomplish a specific task, operational programs shall use that function and not bypass it or implement it in another fashion.

E.11.14 COMPILERS

The implementation of software compilers shall be validated to ensure that the compiled code is fully compatible with the target computing system and application (may be done once for a target computing system).

E.11.15 FLAGS AND VARIABLES

Flags and variable names shall be unique. Flags and variables shall have a single purpose and shall be defined and initialized prior to use.

E.11.16 LOOP ENTRY POINT

Loops shall have one and only one entry point. Branches into loops shall not be used. Branches out of loops shall lead to a single exit point placed after the loop within the same module.

E.11.17 SOFTWARE MAINTENANCE DESIGN

The software shall be annotated, designed, and documented for ease of analysis, maintenance, and testing of future changes to the software.

E.11.18 VARIABLE DECLARATION

Variables or constants used by a safety-critical function will be declared/initialized at the lowest possible level

E.11.19 UNUSED EXECUTABLE CODE

Operational program loads shall not contain unused executable code.

E.11.20 UNREFERENCED VARIABLES

Operational program loads shall not contain unreferenced or unused variables or constants.

E.11.21 ASSIGNMENT STATEMENTS

SCCSFs and other safety-critical software items shall not be used in one-to-one assignment statements unless the other variable is also designated as safety-critical (e.g., shall not be redefined as another non-safety-critical variable).

E.11.22 CONDITIONAL STATEMENTS

Conditional statements shall have all possible conditions satisfied and under full software control (i.e. there shall be no potential unresolved input to the conditional statement). Conditional statements shall be analyzed to ensure that the conditions are reasonable for the task and that all potential conditions are satisfied and not left to a default condition. All condition statements shall be annotated with their purpose and expected outcome for given conditions

E.11.23 STRONG DATA TYPING

Safety-critical functions shall exhibit strong data typing. Safety-critical functions shall not employ a logic "1" and "0" to denote the safe and armed (potentially hazardous) states. The armed and safe state for munitions shall be represented by at least a unique, four-bit pattern. The safe state shall be a pattern that cannot, as a result of a one-, two-, or three-bit error, represent the armed pattern. The armed pattern shall also not be the inverse of the safe pattern. If a pattern other than these two unique codes is detected, the software shall flag the error, revert to a safe state, and notify the operator, if appropriate.

E.11.24 TIMER VALUES ANNOTATED

Values for timers shall be annotated in the code. Comments shall include a description of the timer function, its value and the rationale or a reference to the documentation explaining the rationale for the timer value. These values shall be verified and shall be examined for reasonableness for the intended function.

E.11.25 CRITICAL VARIABLE IDENTIFICATION

Safety-critical variables shall be identified in such a manner that they can be readily distinguished from non-safety-critical variables (e.g., all safety-critical variables begin with a letter S).

E.11.26 GLOBAL VARIABLES

Global variables shall not be used for safety-critical functions.

E.12 SOFTWARE MAINTENANCE REQUIREMENTS AND GUIDELINES

The requirements and guidelines of this section are applicable to the maintenance of the software in safety-critical computing system applications. The requirement applicable to the design and development phase as well as the software design and coding phase are also applicable to the maintenance of the computing system and software

E.12.1 CRITICAL FUNCTION CHANGES

Changes to SCCSFs on deployed or fielded systems shall be issued as a complete package for the modified unit or module and shall not be patched.

E.12.2 CRITICAL FIRMWARE CHANGES

When not implemented at the depot level or in manufacturers' facilities under appropriate QC, firmware changes shall be issued as a fully functional and tested circuit card. Design of the card and the installation procedures should minimize the potential for damage to the circuits due to mishandling, electrostatic discharge, or normal or abnormal storage environments, and shall be accompanied with the proper installation procedure.

E.12.3 SOFTWARE CHANGE MEDIUM

When not implemented at the depot level or in manufacturers' facilities under appropriate QC, software changes shall be issued as a fully functional copy on the appropriate medium. The medium, its packaging, and the procedures for loading the program should minimize the potential damage to the medium due to mishandling, electrostatic discharge, potential magnetic fields, or normal or abnormal storage environments, and shall be accompanied with the proper installation procedure.

E.12.4 MODIFICATION CONFIGURATION CONTROL

All modifications and updates shall be subject to strict configuration control. The use of automated CM tools is encouraged.

E.12.5 VERSION IDENTIFICATION

Modified software or firmware shall be clearly identified with the version of the modification, including configuration control information. Both physical (e.g., external label) and electronic (i.e., internal digital identification) "fingerprinting" of the version shall be used.

E.13 SOFTWARE ANALYSIS AND TESTING

The requirements and guidelines of this section are applicable to the software-testing phase.

E.13.1 GENERAL TESTING GUIDELINES

Systematic and thorough testing is clearly required as evidence for critical software assurance; however, testing is "necessary but not sufficient." Testing is the chief way that evidence is provided about the actual behavior of the software produced, but the evidence it provides is always incomplete since testing for non-trivial systems is always a sampling of input states and not an exhaustive exercise of all possible system states. In addition, many of the testing and reliability estimation techniques developed for hardware components are not directly applicable to software; and care must, therefore, be taken when interpreting the implications of test results for operational reliability.

Testing to provide evidence for critical software assurance differs in emphasis from general software testing to demonstrate correct behavior. There should be a great deal of emphasis placed on demonstrating that even under stressful conditions, the software does not present a hazard; this means a considerable amount of testing for critical software will be fault injection, boundary condition and out-of-range testing, and exercising those portions of the input space that are related to potential hazards (e.g., critical operator functions, or interactions with safety-critical devices). Confidence in the results of testing is also increased when there is evidence that the assumptions made in designing and coding the system are not shared by the test developers (i.e., that some degree of independence between testers and developers has been maintained).

- Does the developer provide evidence that for critical software testing has addressed not only nominal correctness (e.g., stimulus/response pairs to demonstrate satisfaction of functional requirements) but robustness in the face of stress? This includes a systematic plan for fault injection, testing boundary and out-of-range conditions, testing the behavior when capacities and rates are extreme (e.g., no input signals from a device for longer than operationally expected, more frequent input signals from a device than operationally expected), testing error handling (for internal faults), and the identification and demonstration of critical software's behavior in the face of the failure of various other components.

- Does the developer provide evidence of the independence of test planning, execution, and review for critical software? Are unit tests developed, reviewed, executed, and/or interpreted by someone other than the individual developer? Has some amount of independent test planning and execution been demonstrated at the integration test level?

- Has some amount of independent Navy 'free play" testing been provided? If so, during this testing is there evidence that the critical software is robust in the face of "unexpected" scenarios and input behavior, or does this independent testing provide evidence that the critical software is "fragile"? (Navy free play testing should place a high priority on exercising the critical aspects of the software and in presenting the system with the kinds of operational errors and stresses that the system will face in the field.)

- Does the developer's software problem tracking system provide evidence that the rate and severity of errors exposed in testing is diminishing as the system approaches operational testing, or is there evidence of "thrashing" and increasing fragility in the critical software? Does the problem tracking system severity classification scheme reflect the potential hazard severity of an error, so that evidence of the hazard implications of current Problems can be reviewed?

- Has the developer provided evidence that the tests that exercise the system represent a realistic sampling of expected operational inputs? Has some portion of testing been dedicated to randomly selected inputs reflecting the expected operational scenarios? (This is another way to provide evidence that implicit assumptions in the design do not represent hazards in critical software, since the random inputs will be not selectively "screened" by implicit assumptions.)

E.13.2 TRAJECTORY TESTING FOR EMBEDDED SYSTEMS

There is a fundamental challenge to the amount of confidence that software testing can provide for certain classes of programs. Unlike "memory-less" batch programs that can be completely defined by a set of simple stimulus/response pairs, these programs "appear to run continuously...One cannot identify discrete runs, and the behavior at any point may depend on events arbitrarily far in the past." In many systems where there are major modes or distinct partitioning of the program behavior depending on state, there is mode-remembered data that is retained across mode-changes. The key issue for assurance is the extent to which these characteristics have been reflected in the design and especially in the testing of the system. If these characteristics are ignored and the test set is limited to a simplistic set of stateless stimulus/response pairs, the extrapolation to the operational behavior of the system is seriously weakened.

- Has the developer identified the sensitivities to persistent stale and the "input trajectory" the system has experienced? Is this reflected in the test plans and test descriptions?
- Are the developer's assumptions about prohibited or "impossible" trajectories and mode changes explicit with respect to critical functions? "There is always the danger that the model used to determine impossible trajectories over looks the same situation overlooked by the programmer who introduced a serious bug. It is important that any model used to eliminate impossible trajectories be developed independently of the program. Most safety experts would feel more comfortable if some tests were conducted with "crazy" trajectories."

E.13.3 FORMAL TEST COVERAGE

All software testing shall be controlled by a formal test coverage analysis and document. Computer-based tools shall be used to ensure that the coverage is as complete as possible.

E.13.4 GO/NO-GO PATH TESTING

Software testing shall include GO/NO-GO path testing.

E.13.5 INPUT FAILURE MODES

Software testing shall include hardware and software input failure mode testing.

E.13.6 BOUNDARY TEST CONDITIONS

Software testing shall include boundary, out-of-bounds, and boundary crossing test conditions.

E.13.7 INPUT RATA RATES

Software testing shall include minimum and maximum input data rates in worst case configurations to determine the system' capabilities and responses to these conditions.

E.13.8 ZERO VALUE TESTING

Software testing shall include input values of zero, zero crossing, and approaching zero from either direction and similar values for trigonometric functions.

E.13.9 REGRESSION TESTING

SCCSFs in which changes have been made shall be subjected to complete regression testing.

E.13.10 OPERATOR INTERFACE TESTING

Operator interface testing shall include operator errors during safety-critical operations to verify safe system response to these errors.

E.13.11 DURATION STRESS TESTING

Software testing shall include duration stress testing. The stress test time shall be continued for at least the maximum expected operating time for the system. Testing shall be conducted under simulated operational environments. Additional stress duration testing should be conducted to identify potential critical functions (e.g., timing, data senescence, resource exhaustion, etc.) that are adversely affected as a result of operational duration. Software testing shall include throughput stress testing (e.g., CPU, data bus, memory, input/output) under peak loading conditions.

F. LESSONS LEARNED

F.1 THERAC RADIATION THERAPY MACHINE FATALITIES

F.1.1 SUMMARY

Eleven Therac-25 therapy machines were installed, five in the US and six in Canada. The Canadian Crown (government owned) company Atomic Energy of Canada Limited (AECL) manufactured them. The -25 model was an advanced model over earlier models (-6 and -20 models, corresponding to energy delivery capacity) with more energy and automation features. Although all models had some software control, the -25 model had many new features and had replaced most of the hardware interlocks with software versions. There was no record of any malfunctions resulting in patient injury from any of the earlier model Theracs (earlier than the -25). The software control was implemented in a DEC model PDP 11 processor using a custom executive and assembly language. A single programmer implemented virtually all of the software. He had an uncertain level of formal education and produced very little, if any documentation on the software.

Between June 1985 and January 1987 there were six known accidents involving massive radiation overdoses by the Therac-25; three of the six resulted in fatalities. The company did not respond effectively to early reports citing the belief that the software could not be a source of failure. Records show that software was deliberately left out of an otherwise thorough safety analysis performed in 1983, which used fault-tree methods. Software was excluded because "software errors have been eliminated because of extensive simulation and field testing. (Also) software does not degrade due to wear, fatigue or reproduction process." Other types of software failures were assigned very low failure rates with no apparent justification. After a large number of lawsuits and extensive negative publicity, the company decided to withdraw from the medical instrument business and concentrate on its main business of nuclear reactor control systems.

The accidents were due to many design deficiencies involving a combination of software design defects and system operational interaction errors. There were no apparent review mechanisms for software design or QC. The continuing recurrence of the accidents before effective corrective action resulted was a result of management's view. This view had faith in the correctness of the software without any apparent evidence to support it. The errors were not discovered; because the policy was to fix the symptoms without investigating the underlying causes, of which there were many.

F.1.2 KEY FACTS

- The software was assumed to be fail-safe and was excluded from normal safety analysis review.
- The software design and implementation had no effective review or QC practices.
- The software testing at all levels were obviously insufficient, given the results.

- Hardware interlocks were replaced by software without supporting safety analysis.
- There was no effective reporting mechanism for field problems involving software.
- Software design practices (contributing to the accidents) did not include basic, shared-data, and contention management mechanisms normal in multi-tasking software. The necessary conclusion is that the programmer was not fully qualified for the task.
- The design was unnecessarily complex for the problem. For instance, there were more parallel tasks than necessary. This was a direct cause of some of the accidents.

F.1.3 LESSONS LEARNED

- Changeover from hardware to a software implementation must include a review of assumptions, physics and rules.
- Testing should include possible abuse or bypassing of expected procedures.
- Design and implementation of software must be subject to the same safety analysis, review and QC as other parts of the system.
- Hardware interlocks should not be completely eliminated when incorporating software interlocks.
- Programmer qualifications are as important as qualifications for any other member of the engineering team.

F.2 MISSILE LAUNCH TIMING CAUSES HANGFIRE

F.2.1 SUMMARY

An aircraft was modified from a hardware-controlled missile launcher to a software-controlled launcher. The aircraft was properly modified according to standards, and the software was fully tested at all levels before delivery to operational test. The normal weapons rack interface and safety overrides were fully tested and documented. The aircraft was loaded with a live missile (with an inert warhead) and sent out onto the range for a test firing.

The aircraft was commanded to fire the weapon, whereupon it did as designed. Unfortunately, the design did not specify the amount of time to unlock the holdback and was coded to the assumption of the programmer. In this case, the assumed time for unlock was insufficient and the holdback locked before the weapon left the rack. As the weapon was powered, the engine drove the weapon while attached to the aircraft. This resulted in a loss of altitude and a wild ride, but the aircraft landed safely with a burned out weapon.

F.2.2 KEY FACTS

- Proper process and procedures were followed as far as specified.

- The product specification was re-used without considering differences in the software implementation, i.e., the timing issues. Hence, the initiating event was a specification error.
- While the acquirer and user had experience in the weapons system, neither had experience in software. Also, the programmer did not have experience in the details of the weapons system. The result was that the interaction between the two parts of the system was not understood by any of the parties.

F.2.3 LESSONS LEARNED

- Because the software-controlled implementation was not fully understood, the result was flawed specifications and incomplete tests. Therefore, even though the software and subsystem were thoroughly tested against the specifications, the system design was in error, and a mishap occurred.
- Changeover from hardware to software requires a review of design assumptions by all relevant specialists acting jointly. This joint review must include all product specifications, interface documentation, and testing.
- The test, verification and review processes must each include end-to-end event review and test.

F.3 REUSED SOFTWARE CAUSES FLIGHT CONTROLS TO SHUT DOWN

F.3.1 SUMMARY

A research vehicle was designed with fly-by-wire digital control and, for research and weight considerations, had no hardware backup systems installed. The normal safety and testing practices were minimized or eliminated by citing many arguments. These arguments cited use of experienced test pilots, limited flight and exposure times, minimum number of flights, controlled airspace, use of monitors and telemetry, etc. Also, the argument justified the action as safer; because the system reused software from similar vehicles currently operational.

The aircraft flight controls went through every level of test, including "iron bird" laboratory tests that allow direct measurement of the response of the flight components. The failure occurred on the flight line the day before actual flight was to begin after the system had successfully completed all testing. The flight computer was operating for the first time unrestricted by test routines and controls. A reused portion of the software was inhibited during earlier testing as it conflicted with certain computer functions. This was part of the reused software taken from a proven and safe platform because of its functional similarity. This portion was now enabled and running in the background.

Unfortunately, the reused software shared computer data locations with certain safety-critical functions; and it was not partitioned nor checked for valid memory address ranges. The result was that as the flight computer functioned for the first time, it used data locations where this

reused software had stored out-of-range data on top of safety-critical parameters. The flight computer then performed according to its design when detecting invalid data and reset itself. This happened sequentially in each of the available flight control channels until there were no functioning flight controls. Since the system had no hardware backup system, the aircraft would have stopped flying if it were airborne. The software was quickly corrected and was fully operational in the following flights.

F.3.2 KEY FACTS

- Proper process and procedures were minimized for apparently valid reasons; i.e., the (offending) software was proven by its use in other similar systems.
- Reuse of the software components did not include review and testing of the integrated components in the new operating environment. In particular, memory addressing was not validated with the new programs that shared the computer resources.

F.3.3 LESSONS LEARNED

- Safety-critical, real-time flight controls must include full integration testing of end-to-end events. In this case, the reused software should have been functioning within the full software system.
- Arguments to bypass software safety, especially in software containing functions capable of a Kill/Catastrophic event, must be reviewed at each phase. Several of the arguments to minimize software safety provisions were compromised before the detection of the defect.

F.4 FLIGHT CONTROLS FAIL AT SUPERSONIC TRANSITION

F.4.1 SUMMARY

A front line aircraft was rigorously developed, thoroughly tested by the manufacturer, and again exhaustively tested by the Government and finally by the using service. Dozens of aircraft had been accepted and were operational worldwide when the service asked for an upgrade to the weapons systems. One particular weapon test required significant telemetry. The aircraft change was again developed and tested to the same high standards including nuclear weapons carriage clearance. This additional testing data uncovered a detail missed in all of the previous testing.

The telemetry showed that the aircraft computers all failed -- ceased to function and then restarted -- at specific airspeed (Mach 1). The aircraft had sufficient momentum and mechanical control of other systems so that it effectively "coasted" through this anomaly, and the pilot did not notice.

The cause of this failure originated in the complex equations from the aerodynamicist. His specialty assumes the knowledge that this particular equation will asymptotically approach infinity at Mach 1. The software engineer does not inherently understand the physical science involved in the transition to supersonic speed at Mach 1. The system engineer who interfaced

between these two engineering specialists was not aware of this assumption and, after receiving the aerodynamicist's equation for flight, forwarded the equation to software engineering for coding. The software engineer did not plot the equation and merely encoded it in the flight control program.

F.4.2 KEY FACTS

- Proper process and procedures were followed to the stated requirements.
- The software specification did not include the limitations of the equation describing a physical science event.
- The computer hardware accuracy was not considered in the limitations of the equation.
- The various levels of testing did not validate the computational results for the Mach 1 portion of the flight envelope.

F.4.3 LESSONS LEARNED

- Specified equations describing physical world phenomenon must be thoroughly defined, with assumptions as to accuracy, ranges, use, environment, and limitations of the computation.
- When dealing with requirements that interface between disciplines, it must be assumed that each discipline knows little or nothing about the other and, therefore, must include basic assumptions.
- Boundary assumptions should be used to generate test cases as the more subtle failures caused by assumptions are not usually covered by ordinary test cases (division by zero, boundary crossing, singularities, etc.).

F.5 INCORRECT MISSILE FIRING FROM INVALID SETUP SEQUENCE

F.5.1 SUMMARY

A battle command center with a network controlling several missile batteries was operating in a field game exercise. As the game advanced, an order to reposition the battery was issued to an active missile battery. This missile battery disconnected from the network, broke-down their equipment and repositioned to a new location in the grid.

The repositioned missile battery arrived at the new location and commenced setting up. A final step was connecting the battery into the network. This was allowed in any order. The battery personnel were still occupying the erector/launcher when the connection that attached the battery into the network, was made elsewhere on the site. This cable connection immediately allowed communication between the battery and the battle command center.

The battle command center, meanwhile, had prosecuted an incoming "hostile" and designated the battery to "fire," but targeted to use the old location of the battery. As the battery was off-line, the message was buffered. Once the battery crew connected the cabling, the battle command center computer sent the last valid commands from the buffer; and the command was immediately executed. Personnel on the erector/launcher were thrown clear as the erector/launcher activated on the old slew and acquire command. Personnel injury was slight as no one was pinned or impaled when the erector/launcher slewed.

F.5.2 KEY FACTS

- Proper process and procedures were followed as specified.
- Subsystems were developed separately with ICDs.
- Messages containing safety-critical commands were not "aged" and reassessed once buffered.
- Battery activation was not inhibited until personnel had completed the set-up procedure.

F.5.3 LESSONS LEARNED

- System engineering must define the sequencing of the various states (dismantling, reactivating, shutdown, etc.) of all subsystems with human confirmations and re-initialization of state variables (e.g., site location) at critical points.
- System integration testing should include buffering messages (particularly safety-critical) and demonstration of disconnect and restart of individual subsystems to verify that the system always transitions between states safely.
- Operating procedures must clearly describe (and require) a safe and comprehensive sequence in dismantling and reactivating the battery subsystems with particular attention to the interaction with the network.

F.6 OPERATOR'S CHOICE OF WEAPON RELEASE OVERRIDDEN BY SOFTWARE

F.6.1 SUMMARY

During field practice exercises, a missile weapon system was carrying both practice and live missiles to a remote site and was using the transit time for slewing practice. Practice and live missiles were located on opposite sides of the vehicle. The acquisition and tracking radar was located between the two sides causing a known obstruction to the missiles' field of view.

While correctly following command-approved procedures, the operator acquired the willing target, tracked it through various maneuvers, and pressed the weapons release button to simulate firing the practice missile. Without the knowledge of the operator, the software was programmed to override his missile selection in order to present the best target to the best weapon. The software noted that the current maneuver placed the radar obstruction in front of the practice

missile seeker while the live missile had acquired a positive lock on the target and was unobstructed. The software, therefore, optimized the problem and deselected the practice missile and selected the live missile. When the release command was sent, it went to the live missile; and "missile away" was observed from the active missile side of the vehicle when no launch was expected.

The "friendly" target had been observing the maneuvers of the incident vehicle and noted the unexpected live launch. Fortunately, the target pilot was experienced and began evasive maneuvers, but the missile tracked and still detonated in close proximity.

F.6.2 KEY FACTS

- Proper procedures were followed as specified, and all operations were authorized.
- All operators were thoroughly trained in the latest versions of software.
- The software had been given authority to select "best" weapon, but this characteristic was not communicated to the operator as part of the training.
- The indication that another weapon had been substituted (live vs. practice) by the software was displayed in a manner not easily noticed among other dynamic displays.

F.6.3 LESSONS LEARNED

- The versatility (and resulting complexity) demanded by the requirement was provided exactly as specified. This complexity, combined with the possibility that the vehicle would employ a mix of practice and live missiles was not considered. This mix of missiles is a common practice and system testing must include known scenarios such as this example to find operationally based hazards.
- Training must describe the safety-related software functions such as the possibility of software overrides to operator commands. This must also be included in operating procedures available to all users of the system.

G. PROCESS CHART WORKSHEETS

Proceeding Process
- Program Initiation

Software Safety Program Planning
Procuring Agency - Customer

Next Process
- Safety Program Management

Inputs (Suppliers)
- Acquisition Policy
- OCD/OR/MENS
- DOP
- Proposal
- Safety Policy
- Generic Requirements
- Lessons Learned
- PHL

Inputs (Later Milestones)
- Draft PHA
- Draft TEMP
- Draft SEMP
- Draft ILSP
- Draft CRLCMP
- Draft SSS
- Draft S/SDD
- Draft SOW/RFP

Entry Criteria
- Program Initiation

Purpose:

Identify and Establish System Safety Program Tasks and Requirements

Primary Sub-Processes
- Define Acceptable Level of Risk (HRI, Acceptance Authority, Risk Assessment Methodology)
- Establish System Safety Program:
  - Identify and Establish Interfaces to Other Program Disciplines
  - Identify Contract Requirements
  - Develop POA&M
  - Establish Safety Data Library
  - Establish Hazard Tracking Process
  - Resource Requirements Determination:
    - » Analyses Required
    - » Tests Required (U)
    - » Resources Required
  - Develop Safety Statement of Work: (U)
    - » Define Safety Program Requirements (U)
    - » Define Safety Reporting Requirements (U)
    - » Define Safety Review Requirements (U)
  - Proposal Evaluation
    - » Develop Evaluation Criteria
    - » Evaluate Proposals

Related Sub-Processes
- Program Planning and Management
- System Safety Management Plan development

Players

PM, Principal for Safety
SE, SWE, SWSE (EVAL)
SSWG (Later Milestones)

References in Addition to ¶ 4.1.1

Service Specific Guidance
IEEE 1228
FAR's

Outputs (Customers)
- Input to SOW/SOO
- Input to RFP
- Safety Proposal Evaluation Plan
- Safety POA&M
- Review Requirements
- System Safety Management Plan
- System Safety Program Plan with SwSPP Appendix
- SSWG Charter and Subgroups
- Input to SDP
- Input to TEMP
- Input to SEMP
- Input to ILSP
- Input to PHL
- Input to PHA
- Input to CRLCMP

Exit Objective
- System Safety Program Management Plan
- Proposal Evaluation and Acceptance
- Established System Safety Program

Comments:

Proceeding Process
- Program Initiation

Software Safety Program Planning
Developing Agency - Supplier

Next Process
- Safety Program Management

Inputs (Suppliers)
- SOW/RFP
- OCD/OR/MENS
- DOP
- Safety Policy
- PHL

Inputs (Later Milestones)
- Draft PHA
- Draft SEMP
- Draft TEMP
- Draft ILSP
- Draft CRLCMP
- Draft SSS
- Draft S/SDD

Entry Criteria
- RFP Receipt

Purpose:

Define and Implement an Effective System Safety Program

Primary Sub-Processes
- Interpretation of SOW Requirements
- Resource Requirements Determination:
  - Analyses Required
  - Tests Required
  - Resources Required
- Establish System Safety Program
  - Identify and Establish Interfaces to Other Program Disciplines
  - Identify Contract Requirements
  - Establish Safety Data Library
  - Establish Hazard Tracking Process (I)
  - Develop Proposal Input
- Develop Software Safety Program Plan
  - POA&M (U)
  - Allocation of Resources (U)
  - SPRA Schedule (U)

Related Sub-Processes
- Program Planning and Management
- System Safety Program Plan development

Players

PM, SSPM, SWSPM, SE, SWE
Contracts Personnel
SSWG (Later Milestones)

References in Addition to ¶ 4.1.1

IEEE 1228

Outputs (Customers)
- RFP Response/Proposal Input
- Safety POA&M
- Review Requirements
- BAFO Response
- System Safety Management Plan (If Required)
- System Safety Program Plan with SwSPP Appendix
- SSWG Charter and Subgroups
- Software Development Plan Input
- TEMP Input
- SEMP Input
- ILSP Input
- Input to SSS
- Input to S/SDD

Exit Objective
- System Safety Program Plan
- Established System Safety Program

Comments

Proceeding Process
- Software Safety Program Planning

Software Safety Program Management

Next Process
- Functional Hazard List Development
- Tailoring Generic Safety-Critical Software Requirements List

Inputs (Suppliers)
- OR/ORD/MENS
- SOW/SOO/RFP
- Safety Policy

Inputs (Later Milestones)
- SSMP
- SEMP
- TEMP
  - Software Safety Testing

Purpose:

Ensure Effective Execution of Software System Safety Program

Primary Sub-Processes
- Establish and Manage SwSWG
  - Charter SwSWG
    - » Define Membership
  - Chair Meetings (I)
  - Prepare Meeting Minutes (I)
  - Software Hazard Tracking
- Update Safety Program Plans
  - POA&M (U)
  - Allocation of Resources (U)
  - SPRA Schedule (U)
- Safety Program Monitoring
- Provide, Update and Develop Presentations (I)
- Provide Safety Management Inputs to Software Test Plans (I)

Outputs (Customers)
- Inputs to SOW/SOO/RFP
- SwSWG Membership Charter
- Update to SSMP
- Update to Safety Program Schedule
- Updates to SEMP/TEMP
- Input to SPRA Reviews

Entry Criteria
- Safety Program Initiation

Related Sub-Processes
- System level Hazard Tracking
- Initiate Sub-Contractor/Contract SOW/SOO
- Program Monitoring of Sub-Contractor/Contract

Exit Objective
- Program Completion

Players

PM/SSPM/
Customer/Supplier Personnel

References in Addition to ¶ 4.1.1

Service Specific Guidance
IEEE 1228

Comments

Proceeding Process
- Software Safety Program Planning
- Software Safety Program Management

Functional Hazard List Development

Next Process
- Tailoring the Generic Safety-Critical Software Requirements List

Inputs (Suppliers)
- OCD/OR
- DOP
- SOW/SOO/RFP
- Generic Requirements & Hazards List
- Lessons Learned :
  - Data Base
  - Previous Mishap Causes
- Draft PHL
- Functional Allocation
- Safety Function Inputs From Domain Experts
- Draft SSS
- Draft S/SDD
- Draft CRLCMP
- Life Cycle Environmental Profile

Purpose:

To Identify All Hazards and Safety-Critical Functions at the System Level

Primary Sub-Processes
- Tailor Generic and Domain Specific Hazards to System (I)
- Evaluate Hazards Against System Function Requirements (I)
- Evaluate Design Options (I)
  - Identify Technical Safety Risks (I)
  - Identify Potential Programmatic Risks (I)
- Evaluate All Safety-Critical Functions Identified by Each Domain Expert.

Outputs (Customers)
- TEMP Input
- SEMP Input
- PHL Input
- PHA Input
- SSHA Input
- Draft CRLCMP Input
- RHA/SRCA Input
- Initial SCFL
- Recommendations Regarding Design Options
- Trade Study Inputs
- General Test Requirements
- SRCA Input
- Domain Specific Functional Hazards List
- Input to SPRA Reviews

Entry Criteria
- Contract Award (MS 0) or Award of Safety Contract

Related Sub-Processes
- Systems Engineering
- Functional Allocations

Exit Objective
- Satisfactory completion of an initial safety-critical Functions List (SCFL)
- System Design Option Selection
- PDR
- Domain Specific Functional Hazards List

Players

SSWG, SwSWG, Domain Experts

References in Addition to ¶ 4.1.1

IEEE 1498

Comments

Tailoring The Generic Safety-Critical Software Requirements List

Proceeding Process
- Software Safety Program Planning & Management
- Functional Hazard List Development

Next Process
- Preliminary Hazard Analysis - (PHA)

Inputs (Suppliers)
- Design Standards
- General Requirements Doc.
- Generic Safety-Critical Software Requirements Lists
- Lessons Learned:
  - Analyses on Similar Systems
  - Previous Mishap Causes

Inputs (Later Milestones)
- Draft SEMP
- Draft TEMP
- Draft CRLCMP

Purpose:
Establish Domain Specific Generic Software Safety-Critical Requirements List

Primary Sub-Processes
- Obtain Existing Generic Requirements & Guidelines Lists
- Determine Applicability of Requirements to System Under Development (I)
  - Determine Allocation of Applicable Generic Requirements to Hardware, Software and Operator (I)
- Generate Additional Generic Requirements If Applicable
- Prioritize Requirements (I)
- Review Draft SDP/SEMP/TEMP and Incorporate Applicable Requirements (I)
- Obtain Evidence to Support Compliance Assessment
  - Perform Compliance Assessment (I)

Outputs (Customers)
- Tailored Generic Safety-Critical Software Requirements List
- RHA/SRCA Input
- PHL Input
- PHA Input
- SSHA Input
- SDP Input
- Input to Software Test Plans
  - Generic Test Requirements
- Input to SPRA Reviews
- TEMP Input
- SEMP Input
- CRLCMP Input

Entry Criteria
- Contract Award (MS 0) or Award of Safety Contract

Related Sub-Processes
- Brainstorming Session

Exit Objective
- Domain Specific Generic Safety-Critical Software Requirements List

Players
SSWG, SwSWG, SwQA, Domain Experts
Test and Evaluation Team

References in Addition to ¶ 4.1.1
STANAG 4404
Applicable Standards & Specifications

Comments

Preliminary Hazard Analysis - (PHA)

Proceeding Process
- Tailor The Generic Safety-Critical Software Requirements List
- Functional Hazard List Development
- Program Planning/Management

Next Process
- Derived System Specific Safety-Critical Software Requirements
- Preliminary Software Design Analysis

Inputs (Suppliers)
- SOW/SOO/RFP
- Risk Assessment Criteria, HRI
- Draft SSS, S/SDD
- Lessons Learned:
  - Analyses on Similar Systems
  - Incident Reports
  - Previous Mishap Causes
- Life Cycle Environment Profile
- PHL
- Tailored Generic Safety-Critical Software Requirements List

Inputs Later Milestones)
- HARs
- ECPs

Purpose:
Identification, Classification and Tracking of System Level Software Hazards

Primary Sub-Processes
- Identify System Level Causal Factors (I)
- Identify Software Level Causal Factors (I)
  - Apply Analysis Techniques (for Example SFTA'S) (I)
  - Develop Recommendations to Minimize Software Induced Hazards (I)
- Apply HRI and Prioritize Hazards (I)
- Apply Risk Assessment Criteria (Categorize) Hazards (I)
- Link Hazard Causal Factors to Requirements (I)
- Develop Design Recommendations to Mitigate Hazards (I)

Outputs (Customers)
- RHA/SRCA Inputs
- PHA Update
- Input to S/W Design
- Inputs to S/W Development Plan
- Input to Preliminary Software Design Analysis SSHA
- Input to Trade-Off Studies
- Design Implementation Recommendations
- HARs
- Inputs to Software Test Plans
  - Test Requirements
- Input to SPRA Reviews
- Prioritized Hazard List
- Input to SSS, SRS, S/SDD, IRS (I)
- Input to ECP, SENs, PIPs (I)

Entry Criteria
- Upon Completion of PHL

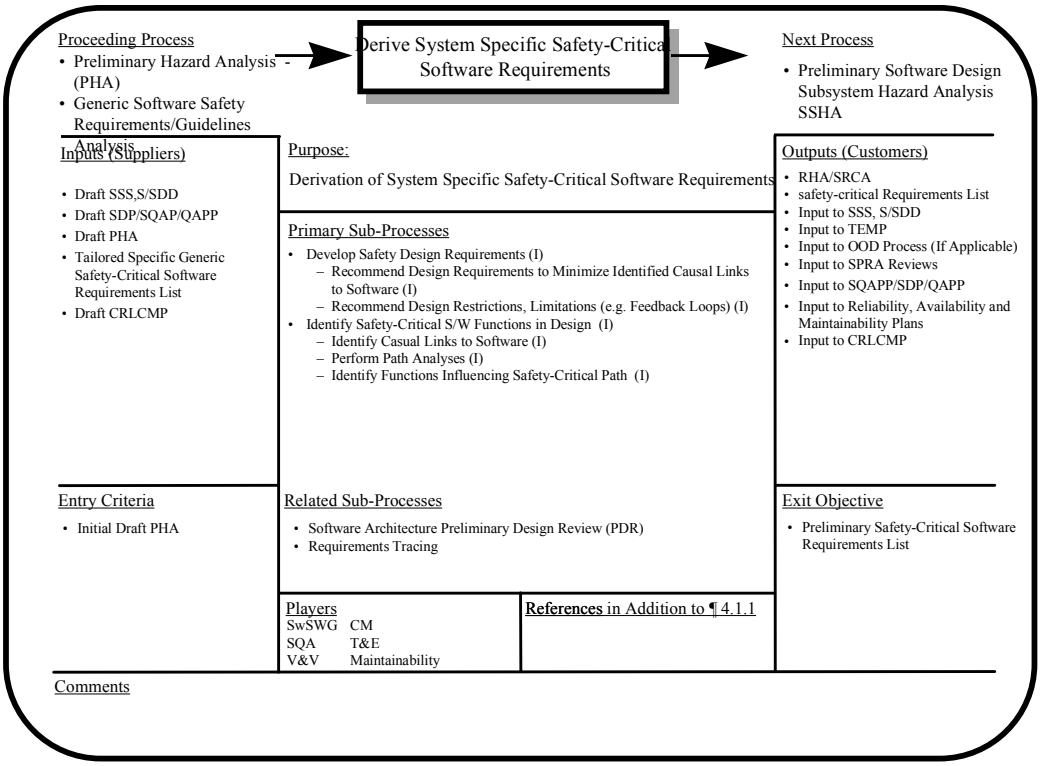
Related Sub-Processes
- Safety Data Library (SDL) Maintenance
- System Level Trade Studies
- Software Level Trade Studies

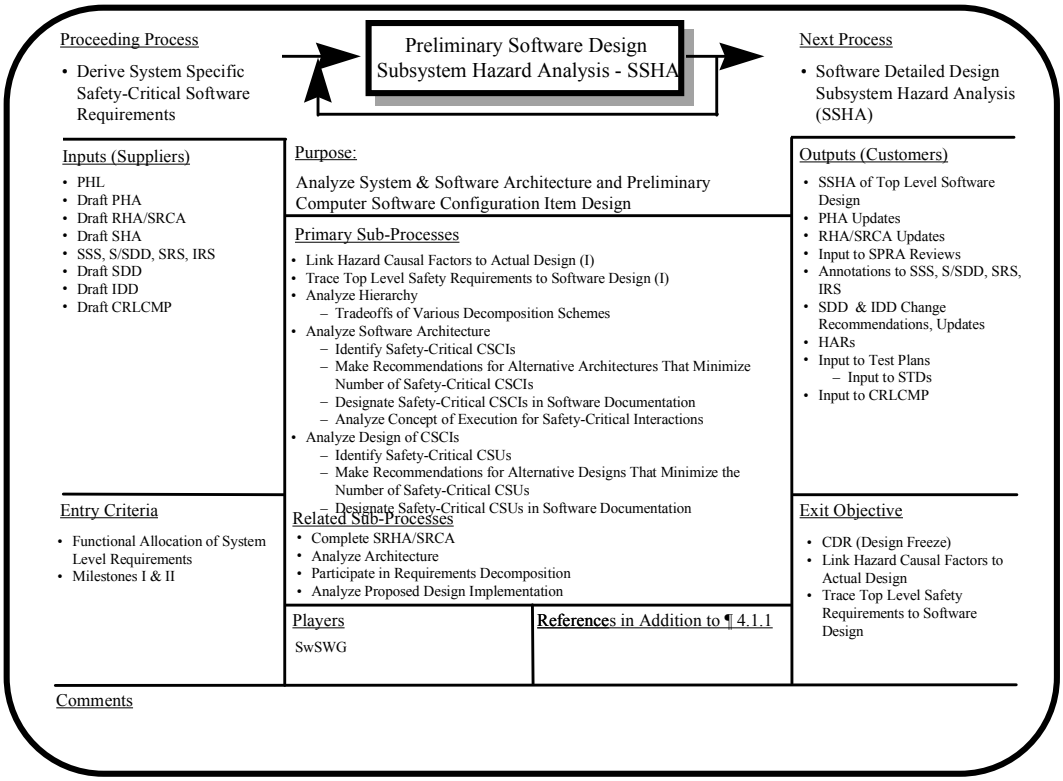
Exit Objective
- Completion of Hazard Categorization, Prioritization, and Determination of All Causal Factors (Initial Drafts)
- Resolution of Identified Hazards (Completion)

Players
SwSWG, Domain Experts

References in Addition to ¶ 4.1.1
IEEE 1498

Comments





Software Detailed Design Subsystem Hazard Analysis - SSHA

Proceeding Process
- Preliminary Software Design Subsystem Hazard Analysis - SSHA

Next Process
- System Hazard Analysis (SHA)
- Software Safety Testing and Analysis

Inputs (Suppliers)
- Draft SHA
- SSS, S/SDD, SRS, SDD, IDS, IRS
- Preliminary Software Design Subsystem Hazards Analysis (SSHA)

Inputs (later iterations)
- Schematics (for Firmware Analyses Only)
- Source Code
- Safety Test Results for Unit Test
- System Hazards Analysis

Purpose:
Analyze the CSCI Design Down to Code Level

Primary Sub-Processes
- Link Hazard Causal Factors to Actual Code (I)
- Analyze Final Implementation of Safety Interlocks, Checks and Flags (I)
  - Ensure Intent of Safety Requirement Is Met
- "What If" Type of Analysis (I)
- Safety-Critical Software Path Analysis (I)
- Identify Hazards Related to Interfacing Subsystems (I)
  - Determine Hazardous Interface Failure Modes
  - Determine Hazardous Interface Data Errors
  - Reflect Failure Modes and Data Errors to SHA

Related Sub-Processes
- Process Flow Chart Analysis
- Code Level Analysis
  - Global & Local Data Analysis
  - Data/Information Flow Analysis
  - Control Flow Analysis
  - Structure Analysis
  - Interface Analysis
  - Interrupt Analysis
  - Timing Analysis

Outputs (Customers)
- Inputs to SHA
- SDD & IDD Recommendations
- Development of Software Development Files (SDF) on a CSCI Basis:
  - Global & Local Variable Descriptions
  - Hierarchy Charts
  - Software Fault Trees
  - CSCI Descriptions
  - Annotated Code Listings
  - Process Flow Charts
- Input to Test Procedures
- Input to SPRA Reviews

Entry Criteria
- Draft Preliminary Software Design Analysis SSHA

Exit Objective
- Resolution of Identified Hazards

Players
SwSWG
Domain Experts
T&E.

References in Addition to ¶ 4.1.1
Applicable Programmers
Reference & Processor Manuals

Comments

System Hazard Analysis (SHA)

Proceeding Process
- Software Detailed Design Subsystem Hazard Analysis - SSHA

Next Process
- Software Safety Test Planning

Inputs (Suppliers)
- PHA
- Draft SSHAs
- SSS
- S/SDD
- IRS
- IDD
- Tailored Generic Safety-Critical Software Design Requirements List
- Incident Reports
- Threat Hazard Assessment
- Life Cycle Environmental Profile
- HARs
- Lessons Learned

Purpose:
To Analyze Subsystem Interfaces & Interactions, Interface Design, and System Functional and Physical Requirements for Safety Hazards and to Assess and Classify System Risks

Primary Sub-Processes
- Analyze IRS, IDD to Ensure Correct Implementation of Safety Design Requirements (I)
- Integrate the Results of the SSHAs(U)
  - Identify Hazards That Cross Subsystem Boundaries (I)
  - Ensure That Hazards Are Mitigated in Interfacing Subsystems or External Systems (I)
  - Identify Unresolved Interface Safety Issues and Reflect Back to SSHAs (I)
- Examine Causal Relationship of Multiple Failure Modes (HW, SW) to Creating Software System Hazards (I)
- Determine Compliance With Safety Criteria and System and Subsystem Requirements Documents (I)
- Assess Hazard Impacts Related to Interfaces (I)
- Develop Recommendations to Minimize Hazard Effects (I)
- Develop Test Recommendations to Verify Hazard Mitigation (I)

Related Sub-Processes

Outputs (Customers)
- Input to SPRA Reviews
- Updates to PHA
- Updates to SSHAs
- HARs
- Inputs to Software Design
- Inputs to Interface Design
- Inputs to Test Requirements
- Inputs to Test Plan
- Prioritized Hazard List
- List of Causal Interrelationships to Hazards

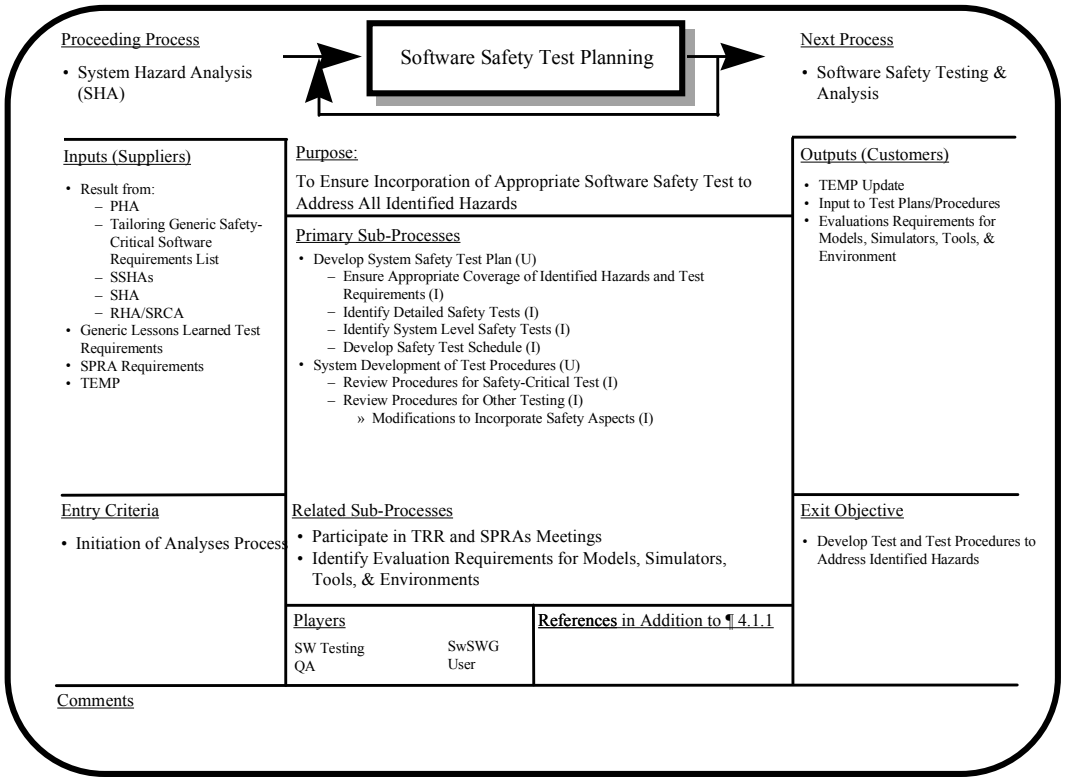
Entry Criteria
- System Design Review

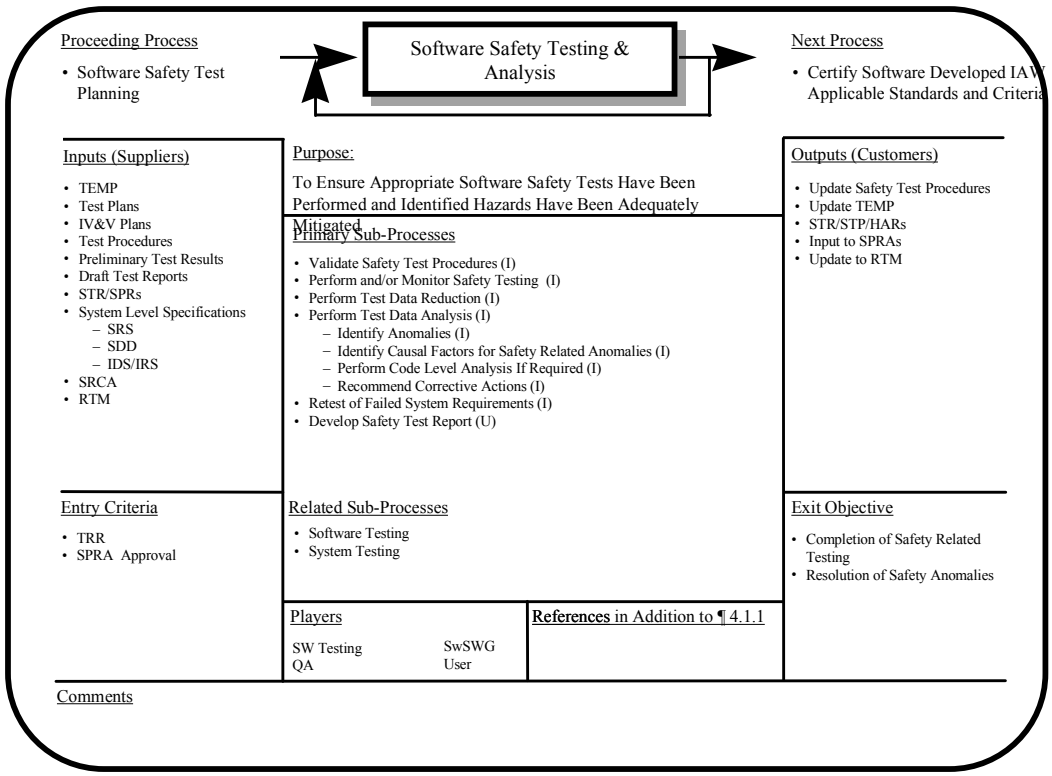
Exit Objective
- Milestone 3

Players
SSWG
SwSWG

References in Addition to ¶ 4.1.1

Comments





Proceeding Process

- Software Safety Testing and Analysis

Verify Software Developed IAW Applicable Standards & Criteria

Next Process

- Software Safety Assessment

Inputs (Suppliers)

- Safety requirements and criteria analysis/Requirements Hazards Analysis
- Software Development Plan
- SRS, SDD, IRS, IDD, VDD
- Source Code Listings
- Configuration Management Plan
- Software Quality Assurance Plan
- Lessons learned
- CRLCMP
- Generic Safety-Critical Software Requirements Lists
- Design Standards
- General Requirements Doc.

Purpose:

Ensure That Applicable Generic Safety Design and Implementation Requirements Have Been Adhered to in the Development of the Safety-Critical Software

Primary Sub-Processes

- Evaluate Safety-Critical Software Products Against Identified Safety Related Requirements (I)
- Evaluate Life Cycle Management Plans Against Safety Related Requirements (U)
- Prepare Compliance Assessment Report for Generic Safety-Critical Software Requirements (U)

Outputs (Customers)

- Compliance Assessment Report for Safety-Critical Software Requirements
- Provide Inputs to Software Safety Assessment Report
- Provide Input to SPRA Reviews

Entry Criteria

Related Sub-Processes

- Software Quality Assurance Assessment

Exit Objective

- Compliance Assessment Report

Players

SwSWG
SQA,, CM, V&V, R&M

References in Addition to ¶ 4.1.1

Comments

Proceeding Process

- Certify Software Developed IAW Applicable Standards and Criteria

Software Safety Assessment

Next Process

Inputs (Suppliers)

- PHA
- SSHAs
- SHA
- HARs
- Safety Test Results
- Software Test Results
- RTM
- IV&V Results
- Compliance Assessment for Safety-Critical Software Requirements
- OPEVAL Test Results
- Acceptance Test Results
- Earlier Software Safety Assessment Reports
- ECPs, STRs, SCNs, Etc.

Purpose:

To Provide Residual Risk Assessment of the Software As a Component of the System

Primary Sub-Processes

- Assess Results of Software Hazards Analyses (I)
- Assess Results of Safety and IV&V Tests (I)
- Review Safety-Critical Software Requirements Compliance Assessment (I)
- Update Hazard Risk Index for Individual Hazards As Appropriate (I)
- Generate Software Safety Assessment Report (U)
- Assess Residual Risk of System Modifications (I) (U)
- Update Software Safety Assessment Report (U)

Outputs (Customers)

- Input to Safety Assessment Report
- Input to Milestone Decision Memos
- Input to SPRA Reviews
- Updated PHA, SSHAs, SHA, HARs
- Software Safety Assessment Report
- Input to Training Manuals, Operating Manuals, and Other Logistics Manuals

Entry Criteria

- CDR Complete

Related Sub-Processes

- Hazard Action Report Closure
- Evaluate OPEVAL and Acceptance Test Results
- Reassess Hazard Controls (I)
- Determine Degree of Hazard Mitigation (I)
- Close Hazard Action Reports As Appropriate (I)
- Determine Residual Risk (U)

Exit Objective

- Successful Completion of OPEVAL and Acceptance Testing

Players

SSWG
SwSWG

References in Addition to ¶ 4.1.1

Comments

H. SAMPLE CONTRACTUAL DOCUMENTS

H.1 SAMPLE REQUEST FOR PROPOSAL

The following represents a sample System Safety paragraph within a RFP that should be considered as a starting point to launch from. As with any example, this sample paragraph must be carefully read and considered as to whether it meets the safety goals and objectives of the program it is being considered for. It must be tailored if tailoring is appropriate.

Suggested Language for Section L, Instructions to Offerors:

"System and Software Safety Requirements -

> Offerors shall describe the proposed System and Software Safety Engineering process, comparing it to the elements in MIL-STD-882 and NATO STANAG 4404. It will explain the associated tasks that will be accomplished to identify, track, assess, and eliminate hazards as an integral part of the design engineering function. It will also describe the proposed process to reduce the residual safety risk to a level acceptable to program management. It will specifically addresses (as a minimum) the role of the proposed System and Software Safety approach in design, development, management, manufacturing planning, and key program events (as applicable) throughout the system life cycle.

Suggested Language for Section M, Evaluation Factors for Award:

"The offeror's approach will be evaluated based on:

> The acceptability of the proposed System and Software Safety approach in comparison to the "System Safety Program" guidance described in the MIL-STD-882, and STANAG 4404 as applied to satisfy program objectives.
>
> The effectiveness of the proposed approach that either mitigate or reduce hazards to the extent to which:
>
> - The proposed approach reflects the integration of system and software safety engineering methodologies into the planning for this program.
> - The proposed approach evaluates the safety impacts of using COTS/GOTS/NDI hardware and software on the proposed system for this program.
> - The proposed approach demonstrates the ability to identify, document, track, analyze, and assess system and subsystem-level hazards and their associated causal factors through detailed analysis techniques. The detailed analysis must consider hardware, software and human interfaces as potential hazard causes.
> - The proposed approach communicates initial safety requirements to the design team including the activities necessary to functionally derive safety requirements from the detailed causal factor analysis.

- The proposed approach produces the engineering evidence of hazard elimination or risk abatement to acceptable levels of residual safety risk that balances hazard severity with probability of occurrence.
- The proposed approach considers all requirements to meet or obtain the necessary certifications or certificate criteria to field, test, and operate the system.

H.2 SAMPLE STATEMENT OF WORK

The following examples represent sample system safety and software safety related paragraphs that can be included in a SOW to ensure that the developer considers and proposes a SSP. As with the sample RFP paragraph above, the SOW system safety paragraphs must be considered as a starting point for consideration. All safety-related SOW paragraphs must be assessed and tailored to ensure they specify all necessary requirements to meet the safety goals and objectives of the acquiring agency.

H.2.1 SYSTEM SAFETY

The following paragraph represents a SOW example where a "full-blown" SSP is required which incorporates all functional components of the system including system-level, and subsystem-level hardware, software, and the human element. The suggested language for system safety engineering is as follows:

System Safety

The contractor shall conduct a system safety management and engineering program using MIL-STD-882 as guidance. The program shall include the necessary planning, coordinating, and engineering analysis to:

- Identify the safety-critical functions of the system and establish a protocol of analysis, design, test, and verification & validation for those functions.
- Tailor and communicate generic or initial safety-related requirements or constraints to the system and software designers as early in the life cycle phase as possible.
- Identify, document and track system and subsystem-level hazards.
- Identify the system-level effects of each identified hazard.
- Categorize each identified hazard in terms of severity and probability of occurrence (specify qualification or quantification of likelihood).
- Conduct in-depth analysis to identify each failure pathway and associated causal factors. This analysis will be to the functional depth necessary to identify logical, practical and cost-effective mitigation techniques and requirements for each failure pathway initiator (causal factor). This analysis shall consider all hardware, software, and human factor interfaces as potential contributors.

- Derive safety-specific hazard mitigation requirements to eliminate or reduce the likelihood of each causal factor.
- Provide engineering evidence (through appropriate inspection, analysis, and test) that each mitigation safety requirement is implemented within the design and the system functions as required to meet safety goals and objectives.
- Conduct a safety assessment of all residual safety risk after all design, implementation, and test activities are complete.
- Conduct a safety impact analysis on all Software Change Notices (SCN) or ECP for engineering baselines under configuration management.
- Submit for approval to the certifying authority, all waivers and/or deviations where the system does not meet the safety requirements or the certification criteria.
- Submit for approval to the acquiring authority an integrated system safety schedule that supports the programs' engineering and programmatic milestones.

The results of all safety engineering analysis performed shall be formally documented in a closed-loop hazard tracking database. The information shall be correlated in such a manner that it can be easily and systematically extracted from the database to produce the necessary deliverable documentation (i.e., PHA, SRCA, SSHA, SHA, O&SHA, FMEA, etc.) as required by the contract. The maturity of the safety analysis shall be commensurate with the maturity of system design in accordance with the acquisition life cycle phase.

H.2.2 SOFTWARE SAFETY

The following example represents a sample software safety program as a "stand-alone" task(s) where another contractor or agency possesses the responsibility of system safety engineering. The software safety program is required that incorporates all functional and supporting software of the system which has the potential to influence system-level, and subsystem-level hazards. The suggested language for software safety engineering program is as follows:

Software Safety

The contractor shall conduct a software safety engineering program using MIL-STD-882 and STANAG 4404 as guidance. This program shall fully support the existing system safety engineering program and functionally link software architecture to hazards and their failure pathways. The program shall include the necessary planning, coordinating, and engineering analysis to:

- Identify the safety-critical functions of the system and establish a protocol of analysis, design, test, and verification & validation for those functions within the software development activities.
- Tailor and communication generic or initial safety-related requirements or constraints to the system and software designers as early in the life cycle phase as possible.

- Analyze the existing documented hazards to determine software influence.
- Consider the system-level effects of each identified hazard.
- Provide input to system safety engineering as to the potential contributions or implications of the software that would affect probability of occurrence.
- Conduct in-depth analysis to identify each failure pathway and associated software causal factors. This analysis will be to the functional depth necessary to identify logical, practical and cost-effective mitigation techniques and requirements for each failure pathway initiator (causal factor). This analysis shall consider all hardware, software, and human factor interfaces as potential contributors.
- Derive safety-specific hazard mitigation requirements to eliminate or reduce the likelihood of each causal factor within the software functional architecture.
- Provide engineering evidence (through appropriate inspection, analysis, and test) that each mitigation SSR is implemented within the design and the system functions as required to meet safety goals and objectives.
- Conduct a safety assessment of all residual safety risk after all design, implementation, and test activities are complete.
- Conduct a safety impact analysis on all SCNs, PTRs, or ECPs for engineering baselines under configuration management.
- Submit for approval to the certifying authority, all waivers and/or deviations where the system does not meet the safety requirements or the certification criteria.
- Submit for approval to the acquiring authority an integrated system safety schedule that supports the programs' engineering and programmatic milestones.

The results of all software safety engineering analysis performed shall be formally documented in a closed-loop hazard tracking database. The information shall be correlated in such a manner that it can be easily and systematically extracted from the database to produce the necessary deliverable documentation (i.e., PHA, SRCA, SSHA, SHA, O&SHA, FMEA, etc.) as required by the contract. The maturity of the software safety analysis shall be commensurate with the maturity of system design in accordance with the acquisition life cycle phase. All software safety analysis shall be conducted and made available to support the goals, objectives, and schedule of the parent SSP.