FAA
Commercial Space
Transportation

Guide to Reusable Launch and Reentry Vehicle Software and Computing System Safety

Version 1.0

July 2006

Guide to Reusable Launch and Reentry Vehicle Software and Computing System Safety

Version 1.0

July 2006

Federal Aviation Administration
Office of Commercial Space Transportation
800 Independence Avenue, Room 331
Washington, DC 20591

TABLE OF CONTENTS

FIGURES

TABLES

1.0 INTRODUCTION

1.1 Purpose

This guide is designed to aid reusable launch vehicle (RLV) and reentry vehicle (RV) operators in producing safe, reliable launch vehicles through the application of a systematic and logical process for identification, analysis, and control of software and computing system safety hazards and risks.

1.2 Background

The FAA, Office of Commercial Space Transportation (AST), regulates commercial space transportation activities only to the extent necessary to ensure public health and safety and the safety of property. In fulfilling its responsibilities, AST issues licenses for expendable launch vehicle (ELV), RLV, and RV launch and reentry activities and experimental permits for reusable suborbital rocket operations. Software and computing systems are becoming increasingly important in assuring safe operations of launch and reentry vehicles. Software and its associated computing systems (computer system hardware and firmware) are used in on-board and ground systems to support safety-critical functions, such as guidance, navigation, and health monitoring. Software is also used to produce safety-critical data and to assist in mitigating system risks. Therefore, analyses are required to identify, characterize, and evaluate the hazards and mitigate the risks associated with the use of software and computing systems on commercial space launch vehicles.

1.3 Scope

This guide provides assistance to launch vehicle operators in developing software and computing system safety analyses to improve the safety of their operations. The guide is not intended to cover all analysis methods to identify software and computing system hazards and risks or all aspects of the methods identified here.

Reusable launch vehicles typically include ascent and descent phases of flight while RVs include only a descent phase. Although RLVs and RVs could technically be different types of vehicles, the software and computing system safety approaches described here are the same for both types of vehicles. For the purposes of this document, the terms "launch vehicle" and "RLV" are assumed to encompass both RLVs and RVs.

1.4 Authority

49 USC Title IX chapter 701, Commercial Space Launch Activities, section 70105

14 CFR part 431, subpart C, Safety Review and Approval for Launch and Reentry of a Reusable Launch Vehicle

14 CFR part 435, Reentry of a Reentry Vehicle Other Than a Reusable Launch Vehicle (RLV)

2.0 DEFINITIONS AND ACRONYMS

2.1 Definitions

Anomaly	An apparent problem or failure that occurs during verification or operation and affects a system, a subsystem, a process, support equipment, or facilities.
Commercial off-the-shelf software	Operating systems, libraries, applications, and other software purchased from a commercial vendor and not custom built for the operator's project.
Failure Modes and Effects Analysis	System analysis by which each potential failure in a system is analyzed to determine the effects on the system and to classify each potential failure according to its severity and likelihood.
Failure Modes, Effects, and Criticality Analysis	Failure Modes and Effects Analysis that includes the relative mission significance or criticality of all potential failure modes.
Fault Tree Analysis	Deductive system reliability analysis that provides qualitative and quantitative measures of the probability of failure of a system, subsystem, or event.
Firmware	Software that resides in the central processing unit's read-only memory and manages the hardware functions.
Flight Safety System	System designed to limit or restrict the hazards to public health and safety and the safety of property presented by a launch vehicle or reentry vehicle while in flight by initiating and accomplishing a controlled ending to vehicle flight.
Functional Hazard Analysis	Systematic, comprehensive examination of vehicle and system functions to identify potentially hazardous conditions that may arise as a result of a malfunction or failure.
Hazard	Equipment, system, operation, or condition

	with an existing or potential condition that may result in loss or harm.
Memory	Parts of an electronic digital computer that retain instructions and data for some interval of time. Memory is the electronic holding place for instructions and data that the microprocessor of a computer can access quickly.
Preliminary Hazard Analysis	System analysis conducted to classify each potential hazard in a system according to its severity and likelihood of occurrence and to develop mitigation measures to those hazards.
Preliminary Hazard List	Initial list of potential system hazards, compiled without regard to risk or possible mitigation measures.
Risk	Measure that takes into consideration the probability of occurrence and the consequence of a hazard to a population or installation.
Risk mitigation	Process of reducing either the likelihood or the severity of a risk.
Safety critical	Essential to safe performance or operation. A safety-critical system, subsystem, condition, event, operation, process, or item is one whose proper recognition, control, performance, or tolerance is essential to system operation such that it does not jeopardize public safety.
Safety-critical computer system function	Any computer system function that, if not performed, if performed out of sequence, or if performed incorrectly, may directly or indirectly cause a public safety hazard.
Software	Digitally coded instructions that manage computer system hardware. Software is a set of instructions or parameters that controls the operation of the computer and related hardware. Operating system software that controls the basic functions of the computer system and application software that enables the computer to perform tasks are included.
Validation	An evaluation to determine that each safety measure derived from a system safety process

	is correct, complete, consistent, unambiguous, verifiable, and technically feasible. Validation is the process that ensures that the right safety measure is implemented.
Verification	An evaluation to determine that safety measures derived from a system safety process are effective and have been properly implemented. Verification provides measurable evidence that a safety measure reduces risk to acceptable levels.

2.2 Acronyms

AC	Advisory Circular
AIAA	American Institute of Aeronautics and Astronautics
AST	Office of Commercial Space Transportation
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
E_c	Expected Average Number of Casualties
ELV	Expendable Launch Vehicle
FAA	Federal Aviation Administration
GOTS	Government Off-The-Shelf
GPS	Global Positioning System
FADEC	Full Authority Digital Electronic Control
FHA	Functional Hazard Analysis
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes, Effects, and Criticality Analysis
FSS	Flight Safety System
FTA	Fault Tree Analysis
FTS	Flight Termination System
IEEE	Institute of Electrical and Electronics Engineers
IIP	Instantaneous Impact Point
JSSSC	Joint Services Software Safety Committee
MCO	Mars Climate Orbiter
MPL	Mars Polar Lander
MSAW	Minimum Safe Altitude Warning

NASA	National Aeronautics and Space Administration
NTSB	National Transportation Safety Board
PHA	Preliminary Hazard Analysis
PHL	Preliminary Hazard List
RLV	Reusable Launch Vehicle
RPM	Revolutions Per Minute
RV	Reentry Vehicle
SDP	Software Development Plan
SFMEA	Software Failure Modes and Effects Analysis
SFTA	Software Fault Tree Analysis
SRM	Solid Rocket Motor
SSPP	System Safety Program Plan

3.0 SOFTWARE AND COMPUTING SYSTEMS IN RLV SAFETY

A launch operator uses a three-pronged approach to ensure that public health and safety and the safety of property would not be jeopardized by the conduct of an RLV mission. The three safety-related elements reflected in this strategy for RLV mission and vehicle operations are as follows:

- Using a logical, disciplined system safety process to identify hazards and to mitigate or eliminate risk.
- Establishing limitations of acceptable public risk as determined through a calculation of the individual and collective risk, including the expected number of casualties (E_c).
- Imposing mandatory and derived operating requirements.

A launch vehicle is a complex and integrated system comprised of hardware, software, human interactions, environmental interactions, and so on. Therefore, a software and computing system safety process should be considered as one part of the integrated system safety process.

A system safety process consists of the structured application of system safety engineering and management principles, criteria, and techniques to address safety within the constraints of operational effectiveness, time, and resources throughout all phases of the life cycle of a system or program. This process identifies and analyzes hazards and risks, then reduces or controls such risks to acceptable levels, as described in FAA Advisory Circular (AC) 431.35-2A, *Reusable Launch and Reentry Vehicle System Safety Process*, July 2005.

Although software safety is part of the launch vehicle system safety effort that includes hardware, recognizing some key differences between hardware and software is important. Hardware, including computer system hardware and associated equipment, fails most often because of such factors as deficiencies and variability in design, production, and maintenance. However, software does not fail in the conventional sense – software does not break, wear out, or fall out of tolerance like hardware. Software faults are primarily systematic, not random, and are primarily caused by design faults, particularly in defining and interpreting requirements. Randomness can be introduced into software operation by actions that interrupt the operation or by computer memory faults. For example, a user performing an action to stop the software from processing normally may introduce randomness. However, the majority of software problems can be traced to improper design or improper implementation of that design. Therefore, the software and computing system safety effort should focus on the fault avoidance, removal, detection, and tolerance. The launch vehicle operator should

- prevent faults from entering the system through the development of valid safety requirements (fault avoidance);
- find and correct faults within a system before it enters service, through the use of a thorough verification process (fault removal);
- use techniques to detect problems in the operational system so that their effects can be minimized (fault detection); and
- design the system to operate correctly even if faults are present (fault tolerance).

For an effective software and computing system safety effort, the operator should consider using a combination of these approaches.

Software and computing system safety analyses should consider safety aspects of the following items:

- Computer system hardware, which includes physical devices that assist in the transfer of data and perform logic operations. Examples include central processing units (CPU), busses, display screens, memory cards, and peripherals.
- Computer system firmware, which is resident software that controls the CPU's basic functioning.
- Computer system software, including operating system software and applications programs.

In addition, because software safety is a systems issue, software and computing systems must be considered with respect to other aspects of the system, such as the following:

- Physical entities whose function and operation are being monitored or controlled, often called the application.
- Sensors (thermocouples, pressure transducers).
- Effectors that take an instruction from the computing system and impart an action on the system (valves, actuators).

- Data communication to other computers.
- Humans who will interact with the system.

Safety is enhanced through the use of layers of protection that include both software- and hardware-specific safety measures.

4.0 SOFTWARE AND COMPUTING SYSTEM SAFETY PROCESS

Figure 1 shows the software and computing system safety process. Each of these steps will be described below. Note that although this process is presented in a linear, one-pass fashion for ease of discussion, the software and computing system safety process is in fact iterative. Over the life of the project, analyses and processes are updated. Additional information is obtained as the launch operator discovers new hazards, finds that certain hazards no longer apply, makes changes to the system, and continues to improve methods for defining and refining the system.

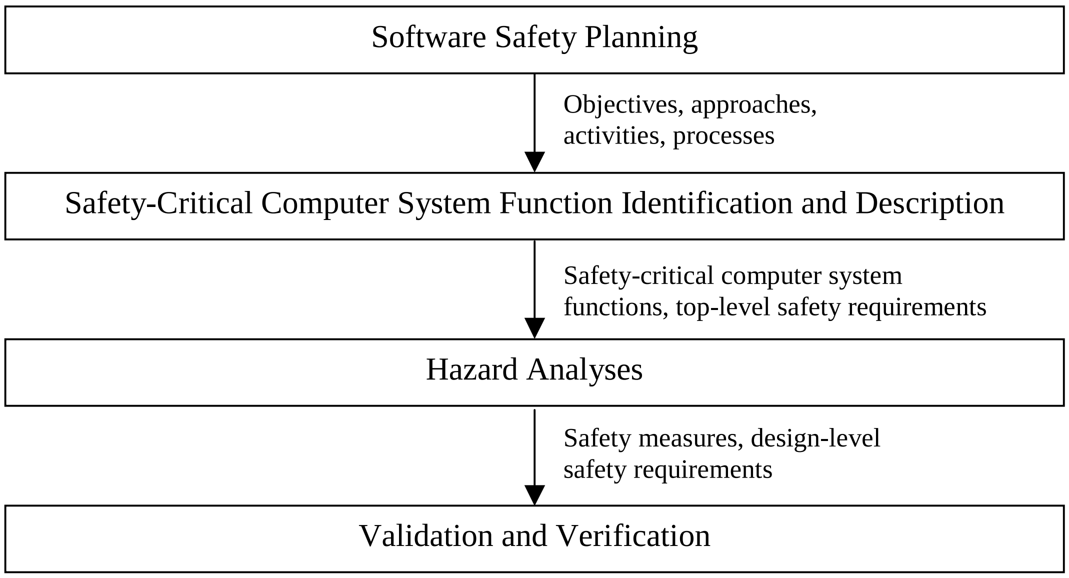

Software Safety Planning
Objectives, approaches, activities, processes
Safety-Critical Computer System Function Identification and Description
Safety-critical computer system functions, top-level safety requirements
Hazard Analyses
Safety measures, design-level safety requirements
Validation and Verification

Figure 1. Software and computing system safety process

4.1 Software Safety Planning

Software safety planning defines the approach that will aid in producing software that will satisfy system safety requirements. Planning helps ensure that safety is designed in and incorporated from the beginning of the life cycle. Early hazard identification and risk reduction will typically provide the most effective and lowest cost approach to addressing safety concerns. Because software is one subsystem in the vehicle, an operator should discuss the software and associated computing system safety process in the System Safety Program Plan (SSPP). The SSPP describes the tasks and activities required to identify, evaluate, and control hazards and reduce risk. Often, the description of software-specific tasks includes analyses, such as a Software Failure Modes and Effects Analysis (SFMEA) or Software Fault Tree Analysis (SFTA). The SSPP should also identify whether software considerations would be included as part of other system-wide analyses, such as Preliminary Hazard Lists (PHL) or Preliminary Hazard Analyses (PHA). Examples of software considerations in the SSPP are provided in AC 431.35-2A.

In addition to the SSPP, an operator should prepare a Software Development Plan (SDP). Software safety is based upon (1) developing valid requirements as a result of efforts to identify, characterize, and reduce the hazards and risks and (2) assuring the integrity of the software and proper implementation of the safety requirements. Integrity here means the likelihood of a safety-related system satisfactorily performing the required safety functions under all stated conditions. The SDP describes the activities, methods, and standards for the development of safety-critical software to reduce the software risks and assure software integrity with respect to safety. The SDP should include both management and engineering of the software safety effort. The following elements should be included in the plan:

Software System Safety Management

- Purpose, scope, and objectives of the software safety program and its tasks.
- Organization and responsibilities.
- Schedule and critical milestones.
- Staff training requirements specific to the software design, development, testing, implementation, and maintenance.
- Policy and procedures on the use of previously developed software, including reused and Commercial Off-The-Shelf (COTS) software.
- Contract management.
- Tools approved for use on the project, such as compilers, computer-aided software engineering products, editors, path analyzers, simulators, and automated test equipment.
- Design, coding, and safety standards and guidance documents.
- Generic safety requirements and approaches to managing requirements.

Software System Safety Engineering

- Methods for identifying safety-critical computer system functions.
- Methods for performing software and computing system hazard analyses that generate software safety requirements.
- Approaches to validation and verification of safety-critical software and computing systems, including testing, analyses, and inspections.
- Software configuration management.

- Software quality assurance.
- Installation processes and procedures.
- Maintenance activities.
- Anomaly reporting, tracking, root cause analysis, and corrective action processes.
- Training requirements.

Additional information about software safety planning can be found in IEEE 1228-1994, *IEEE Standard for Software Safety Plans*, 1994.

4.2 Safety-Critical Computer System Function Identification and Description

Once the planning is completed, an operator should identify and describe safety-critical computer system functions. The operator would normally do this in three steps:

1. Identify safety-critical vehicle functions.
2. Identify safety-critical computer system functions.
3. Describe safety-critical computer system functions.

When software is integrated as part of a system to command, control, or monitor safety-critical functions, special measures are required to understand and mitigate safety risks. Identifying those functions that are essential to safe performance or operation is the first imperative. Isolating these safety-critical vehicle functions helps the operator determine priorities within the safety effort, focus use of resources, and tailor activities based on the most important safety concerns. Examples of safety-critical vehicle functions include, but are not limited to, the following:

- Dynamic and static air data processing.
- Attitude, altitude, and heading sensing and display.
- Instantaneous Impact Point (IIP) tracking and display.
- Propulsion system control, including rocket engine start or shutdown operations.
- Propulsion system health monitoring and display, such as engine pressures and temperatures.
- Propellant dumping.
- Propellant tank and fuel management system maintenance and sensing.
- Flight control actuation health monitoring and display.
- Fire detection and suppression.
- Environmental controls, such as oxygen supply, defogging, carbon dioxide removal, and maintenance of cabin temperatures and pressures.
- Landing gear position and control.

- Ground braking, steering, and deceleration.
- Electrical power distribution and control.
- Active and passive thermal control health monitoring and display.
- Structures health monitoring and display.
- Guidance.
- Navigation.
- Communications.
- Payload health monitoring and management.
- Preflight safety parameter determinations, such as trajectory and winds.
- Recovery system deployment, such as parachutes and emergency egress systems.
- Flight safety system monitoring and operation.

A Functional Hazard Analysis (FHA) is one common approach used to identify safety-critical vehicle functions. An operator could also develop a Preliminary Hazard List (PHL) to assist in defining safety-critical vehicle functions. Other approaches for determining safety-critical functions include industry guidelines, such as the AIAA *Guide to the Identification of Safety-Critical Hardware Items for Reusable Launch Vehicle (RLV) Developers*, May 1, 2005; government guidance documents, such as FAA AC 23.1309, *Equipment, Systems, and Installations in Part 23 Airplanes,* March 1999; mishap data; and experience with similar systems.

After identifying the safety-critical functions of the vehicle, the operator should identify the safety-critical computer system functions. Safety-critical computer system functions monitor, control, or provide data for the safety-critical vehicle functions. Safety-critical computer system functions include the following:

- Transmission of safety-critical data, including time-critical data and data about hazardous conditions through a computing system.
- Software used to detect faults in safety-critical computer hardware and software.
- Software that responds to the detection of a safety-critical fault.
- Software and computing systems used in a flight safety system.
- Processor-interrupt software associated with previously designated safety-critical computer system functions.
- Software that computes safety-critical data.
- Software that accesses safety-critical data.

After identifying the safety-critical computer system functions, the operator should describe each software function and associated computing system. Examples include, but are not limited to, the following:

- Interfaces between the software and other systems.
- Flow charts or diagrams that show data busses, hardware interfaces, data flow, power systems, and operations of each safety-critical software and computing system function.
- Logic diagrams and software designs.
- Operator consoles or displays.
- Operator user manuals and documentation.

The output from this step in the software and computing system safety process includes identification of safety-critical computer system functions and description of the software and computing system. Note that safety-critical computer system functions may not be safety-critical during all phases of flight. In such cases, the operator may also identify the phase of flight where the function is safety critical.

At this stage, the operator should define the top-level, or generic, requirements. In general, these requirements are not tied to a specific hazard but rather are derived from knowledge of the safety-critical functions, design standards, safety standards, mishap reports, experience on similar software, and lessons learned from other programs. Appendix A provides examples of generic software safety requirements assembled from various sources that an operator can use in developing its own top-level safety requirements.

4.3 Hazard Analyses

Once the safety-critical computer system functions have been identified, an operator should perform analyses to identify the hazards, assess the risks, and identify risk mitigation approaches associated with those functions. In software intensive systems, mishaps often occur because of a combination of factors, including component failure and faults, human error, environmental conditions, procedural deficiencies, design inadequacies, and software and computing system errors. In such systems, software often cannot be divorced from the system where it resides. The operator should, therefore, first perform a preliminary analysis that considers software hazards on a system or subsystem level. An example of such a system would be a flight display, which might include both hardware and software components. An operator can perform these system-level hazard analysis and risk assessments in a manner similar to that used for systems consisting only of hardware. Typical approaches include PHA and Failure Modes, Effects, and Criticality Analysis (FMECA). Often an operator will use the PHL or FHA as a starting point. In developing this analysis, an operator should classify each hazard according to its consequences and likelihood of occurrence. Advisory Circular 431.35-2A and MIL-STD-882D, *Standard Practice for System Safety*, provide definitions that an operator can use in the qualitative evaluation of severity and likelihood for the assessment of hazards. The analysis will result in mitigation measures to reduce risk and system-level requirements to implement those mitigation measures. For example, mitigation measures for the loss of a flight control display might involve using a redundant display or aborting the mission and shutting down the propulsion system. A resulting safety requirement may entail developing procedures that specify the abort and shutdown conditions.

In addition to the system or subsystem hazard analysis, the operator should perform software specific hazard analyses. Software specific hazard analyses identify what can go wrong, what are the potential effects, and what mitigation measures can be used to reduce the risk. Note, however, because of the difficulties in assigning probabilities to newly developed software, the software specific hazard analysis does not usually include an assessment of the likelihood of a software fault. Typical software specific hazard exploration techniques include SFMEA and SFTA. Appendix B provides examples of SFMEA and SFTA. The analytical method and level of detail in the analysis should be made based on the complexity of the system, intricacy of the operations, and scope of the program.

An operator's software specific hazard analyses should consider multiple error conditions (see subparagraph 4.3.1). Subparagraph 4.3.2 describes potential risk mitigation measures for those error conditions.

4.3.1 Error Conditions

Error Condition	Examples
Calculation or computation errors	
Incorrect algorithms	The software may perform calculations incorrectly because of mistaken requirements or inaccurate coding of requirements.
Calculation overflow or underflow	The algorithm may result in a divide by zero condition.
Data errors	
Improper data	The software may receive out of range or incorrect input data, no data because of transducer drop out, wrong data type or size, or untimely data; produce incorrect or no output data; or both.
Input data stuck at some value	A sensor or actuator could always read zero, one, or some other value.
Large data rates	The software may be unable to handle large amounts of data or many user inputs simultaneously.
Logic Errors	
Improper or unexpected commands	The software may receive bad data but continue to run a process, thereby doing the right thing under the wrong circumstances.
Failure to issue a command	The software may not invoke a routine.
Command out of sequence	A module may be exercised in the wrong sequence, or a system operator may

Error Condition	Examples
	interrupt a process leading to an out of sequence command.
Incorrect timing	System operators can interrupt processes causing a problem in timing sequences, or processes may run at the wrong times.
Interface errors	
Incorrect, unclear, or missing messaging	A message may be incorrect or unclear, leading to the system operator making a wrong decision.
Poor interface design and layout	An unclear graphical user interface can lead to an operator making a poor decision.
Inability to start or exit processing safely	A system operator may be unable to start or stop a test of a flight safety system once the automated routines have started.
Multiple events occurring simultaneously	A system operator may provide input in addition to expected automated inputs during software processing.
Software development environment errors	
Improper use of tools	Turning on the compiler option to optimize or debug the code in production software may lead to a software fault.
Changes in the operating system or commercial software module	Upgrades to an operating system may lead to a software fault.
Hardware-related errors	
Unexpected shutdown of the computing system	Loss of power to the CPU or a power transient may damage circuits.
Memory overwriting	Improper memory management may cause overwriting of memory space and unexpected results.

Appendix B, table 1, provides an example of a classification scheme for software and computing system errors that an operator can use to develop its hazard analysis.

4.3.2 Risk Mitigation Measures

The recommended order of precedence for eliminating or reducing risk in the use of safety-critical software and computing system follows:

1. Design for minimum risk.
2. Incorporate safety devices.
3. Provide warning devices.
4. Develop and implement procedures and training.

Software specific analysis should provide mitigation approaches for each potential hazard identified. Dunn (2002) and Storey (1996) describe potential mitigation measures. Mitigation measures include, but are not limited to, the following approaches:

Mitigation Measures	**Examples**
Software fault detection	Built-in tests, incremental auditing, and checks for infinite loops.
Software fault isolation	Isolating safety-critical functions from non-safety-critical functions, implementing checks on data input to a safety-critical function, and assuring that a malfunctioning component does not accidentally call a safety-critical module.
Software fault tolerance	Recovery blocks that use multiple software versions of progressively more reliable construction should faults occur, N-version programming in which multiple versions of the software execute simultaneously, and majority voting between different software versions and architectures.
Real-time hardware and software fault recovery	Incremental reboots or exception handling.
Memory management approaches	Placing limits on the dynamic memory usage or reverting to a safe system state if memory limits are reached.
Task prioritization schemes	Assuring that commands are issued in the proper sequence and detecting if commands are conducted out of sequence.
Defensive programming techniques	Verifying pre- and post-conditions, such as the ranges of certain variables; initiating a safe response if a variable is out of range; preventing branching into a safety-critical loop; and using bounded time requirements.
Verified watchdog timers	Using a device that perform a specific operation after a certain period if something goes wrong with an electronic system (and the system does not respond on its own) to detect the loss of a processor.
Checksums	Using numbers representing the sum of the digits in digital data, used to test whether data transmission errors have occurred.
Parity bit checks	Adding a binary digit to a group of bits (and counting of those bits) to detect proper communication between systems.
Interrupt scheduling	Making sure that all interrupt priorities and responses are defined.
Redundant functionality using	Adding a manual shutdown capability in addition

Mitigation Measures	Examples
hardware devices	to a software-driven shutdown system or redundant input devices.
Redundant mission abort capability	Implementing manual and automated mission abort procedures in the event of a specified software fault or computing system loss.
Redundant software reviews	Implementing manual and automated review of software written to memory.

The output from the software specific hazard analysis process includes design-level safety requirements based on safety measures developed to mitigate hazards. These design requirements could include specific hardware mitigation measures, such as redundant functionality using hardware, or coding requirements that must be implemented. Design requirements are statements that can be translated into code without interpretation, or specific mitigations that must be implemented. Examples of design requirements include the following:

- Time must not be less than zero.
- Total Mass = Mass Stage 1 + Mass Stage 2 + Miscellaneous Mass
- A software function must be developed and used to detect out of range temperature and pressure conditions.

The launch vehicle operator should obtain input to the software requirements from environmental requirements, program specifications, facility requirements, tailored generic requirements, and system functionality.

Note that the software specific subsystem hazard analyses may also yield additional hazards and mitigation measures. An operator should update system hazard analyses to account for new hazards and mitigation measures identified in the software specific hazard analysis process.

4.4 Validation and Verification

Software safety analyses generate top- and design-level safety requirements that are used to meet the operator’s system safety goals. These requirements typically result from implementation of mitigation measures, operational controls, or software coding specifications to reduce risk. Other sources of requirements may include operating practices, standard industry practices, and regulations. Regardless of the source, effective management of the complete set of safety requirements is an essential component of system safety engineering.

Deficient requirements are the single largest factor in software and computing system project failure, and deficient requirements have led to a number of software-related aerospace failures and accidents (see appendix C). The applicant should implement a process for managing requirements throughout the life cycle. The FAA/AST *Guide to*

Reusable Launch Vehicle Safety Validation & Verification Planning and IEEE 1012-1998 provide examples of approaches that can assist in managing requirements. The IEEE 1228-1994 and NASA GB 8719.13 also provide methods for managing and analyzing software safety requirements.

The validation and verification process is used to manage the set of safety requirements. Validation determines that the correct requirements are implemented. To do this, the validation effort ensures that each requirement is unambiguous, correct, complete, consistent, testable, and operationally and technically feasible. In addition, the validation process demonstrates that designers, programmers, and others implementing the requirements understand them. Verification determines that safety requirements are effective and have been properly implemented. Acceptable methods of verification include analyses, formal inspections, and testing. These methods are often used in combination. The acceptability of one method over another depends on the feasibility of the method and the maturity of the vehicle and operations. Subparagraphs 4.4.1 through 4.4.4 describe specific areas of consideration in verification.

4.4.1 Analysis

Analyses to verify that the safety-critical software requirements are correctly implemented may include, but are not limited to, the following:

Analysis	Comments
Logic	Evaluates the sequence of operations represented by the coded program and detects programming errors that might create hazards.
Data	Evaluates the data structure and usage in the code to ensure each is defined and used properly by the program. Analysis of the data items used by the program is usually performed in conjunction with logic analysis.
Interface	Ensures compatibility of program modules with each other and with external hardware and software.
Constraint	Ensures that the program operates within the constraints imposed upon it by requirements, design, and target computer. Constraint analysis is designed to identify these limitations, ensure that the program operates within them, and make sure that all interfaces have been considered for out-of-sequence and erroneous inputs.
Programming style	Ensures that all portions of the program follow approved programming guidelines.
Noncritical code	Examines portions of the code that are not considered safety-critical code to ensure that they do not cause hazards. As a general rule, safety-critical code should be isolated from non-safety-critical code. The intent of this analysis is to prove that this isolation is complete and that interfaces between safety-critical code and non-safety-critical code do not create hazards.

Analysis	Comments
Timing and sizing	Evaluates safety implications of safety-critical requirements that relate to execution time, clock time, and memory allocation.

Additional information about software analysis methods is available in IEEE 1228-1994, the *Joint Services Software Safety Committee (JSSSC) Software System Safety Handbook*, and appendix J of the *FAA System Safety Handbook*.

4.4.2 Formal Inspections

Formal inspections are well thought-out technical reviews that provide a structured way to find and eliminate defects in software documentation products, ranging from a requirements document to the actual source code. These inspections differ from informal reviews or walkthroughs in that there are specified steps to be taken and roles assigned to individual reviewers. Further information regarding formal inspections can be found in NASA's *Software Formal Inspections Standard* (NASA-STD-2202-93) and NASA's *Software Formal Inspections Guidebook* (NASA-GB-A302).

4.4.3 Testing

Tests may include, but are not limited to, the following:

Test	Comments
Unit	Demonstrates correct functioning of critical software elements.
Interface	Shows that critical computer software units execute together as specified.
System	Demonstrates the performance of the software within the overall system.
Stress	Confirms the software will not cause hazards under abnormal circumstances, such as unexpected input values or overload conditions.
Regression	Demonstrates changes made to the software did not introduce conditions for new hazards.

Software Testing in the Real World (Kit 1995), *Testing Computer Software* (Kaner et al. 1999), and *Safety Critical Computer Systems* (Storey 1996) provide additional information about software testing.

Before verification testing begins, the operator should develop a test plan to show how the results of software testing will be used to meet all software safety requirements. A plan normally prescribes the scope, approach, resources, and schedule of the testing activities. The operator's plan should include a description of the test environments, including software tools and hardware test equipment.

The operator should also define specific test cases. Test cases describe the inputs, predicted results, test conditions, and procedures for conducting the test. The operator should design test cases to assure that all safety requirements are covered. These test cases should include scenarios that demonstrate the ability of the software to respond to both nominal and off-nominal inputs and conditions. Off-nominal and failure test scenarios often come from the hazard analysis.

The operator should record the results of the tests; this is often done in a test log. Anomalies discovered during testing should also be recorded (see paragraph 5.4). IEEE-STD 829-1983 provides additional information on test documentation.

Testing has traditionally been relied on to verify that software requirements have been met and have been implemented correctly. While testing should be used whenever possible for verification, exhaustive testing of all possible conditions in complex software systems is effectively impossible. The combinations of possible input conditions are prohibitively large in all but the most trivial software programs. Therefore, the operator should use a combination of verification approaches (analysis, inspection, and test). The operator should also use proven methods to verify the software which include, but are not limited to, the following types of tests:

Test	**Comments**
Equivalence partitioning	Identifying valid and invalid classes of input conditions. If, for example, an input field calls for values between 1 and 10, inclusive, then a valid equivalence class would be all values between and including 1 and 10. Invalid equivalent classes would be values less than 1 and values greater than 10.
Boundary value	Testing at the extremes of an input condition, values close to those extremes, and crossing those boundaries. If, for example, an input field calls for values between 0 and 100, then test inputs could include 0, 100, 0.0001, and 99.9999.
Error guessing	Using empty or null lists and strings, negative numbers, null characters in a string, and incorrect entry types.
Statement coverage	Assuring that each statement is executed at least once.
Decision coverage	Assuring that each decision takes on all possible outcomes at least once. For example, assuring that all “if” and “while” statements are evaluated to both true and false.
Function coverage	Determining whether each function or procedure was invoked.
Call coverage	Verifying that each function call has been completed.

Software and computing system testing is conducted as part of a larger system and vehicle verification program. Interlocks are often turned off, and inhibits are often implemented during testing and maintenance to reduce the risks to operations personnel and to facilitate testing. Interlocks are hardware or software safety functions that prevent succeeding operations when specific conditions exist. An example of an interlock would be a key that controls a safe/arm switch. Inhibits prevent a specific event function from occurring or a specific function from being available. For example, an operator may use a software function to inhibit an audible alarm during sensor calibration testing. Failure to properly manage interlocks and inhibits has contributed to accidents. In developing and using interlocks and inhibits, an operator should consider lessons learned from previous accidents and software development efforts (see appendix A).

4.4.4 Verification After Deployment

Changes to both the hardware and software after deployment can produce software and computing system anomalies. A launch operator should identify a process for verifying the integrity of the safety-critical software and computing system after deployment. Examples of such verification methods include the use of checksums and parity bit checks to assure proper data transfer, built-in or external measures for evaluating the software and its data, inspections to detect unauthorized modification of the software or its data, and regression testing.

4.5 Software Safety Evolution - Updating Analyses

Development of any system requires making changes throughout the life cycle of the system. This need is especially true of software because changes are made to solve problems encountered during verification or identified after deployment. Changes are also the result of upgrades and product enhancements. The operator should implement a process to update its analyses. In particular, as the life cycle progresses an operator should

- determine whether changes have created any new safety-critical computer system functions;
- determine whether the changes introduced new hazards;
- identify any hazards that no longer apply;
- identify new software safety requirements;
- determine whether any changes would circumvent safety measures;
- determine the need for new safety measures;
- determine whether previously identified mitigation measures require changes;
- determine whether verification approaches and test documentation need updates; and
- determine the need for additional verification, including regression testing.

5.0 ADDITIONAL CONSIDERATIONS

Several additional factors should be considered in the management of the software and computing system safety process. These factors include development standards, configuration management and control, quality assurance, anomaly reporting and tracking, previously developed software and computing systems, training, and maintenance (see paragraphs 5.1 through 5.7).

5.1 Development Standards

The launch vehicle operator should identify software development standards that define the rules and constraints for the software development process. These standards should enable uniformly designed and implemented software components and prevent the use of methods that are incompatible with safety requirements. Software development standards include requirements, design, coding, and safety standards, as follows:

- Requirements standards might include methods for developing requirements and a description of how the requirements flow down to coding.
- Design standards might include restrictions on the use of scheduling and interrupts or rules for conditional branches to reduce complexity.
- Coding standards might include requirements for the programming language; naming conventions for modules, variables, and constants; and constraints on the use of tools.
- Safety standards might include approaches for analyzing risk and classifying hazards, such as MIL-STD-882.

5.2 Configuration Management and Control

Changes to the software, especially on safety-critical systems, can have significant impacts on public safety. The launch vehicle operator should implement a software configuration management and control process that will at a minimum:

- Identify components, subsystems, and systems.
- Establish baselines and traceability.
- Track changes to the software configuration and system safety documentation.

This software configuration management and control process should be in force during the entire life cycle of the program, from initiation of development through software retirement, and should include control of project documentation, source code, object code, data, development tools, test tools, environments (both hardware and software), and test cases.

5.3 Quality Assurance

Quality assurance is used to verify that objectives and requirements of the software system safety program are being satisfied and to confirm that deficiencies are detected, evaluated, tracked, and resolved. This function is usually performed through audits and inspections of elements and processes, such as plans, standards, and problem tracking and configuration management systems. In addition, the software quality assurance function can evaluate the validity of system safety data. The launch vehicle operator should perform quality assurance activities suitable to the objectives of the program. NASA's

Software Assurance Guidebook (NASA-GB-A201), *Software Quality Assurance Audits Guidebook* (NASA-GB-A301), and *Software Assurance Standard* (NASA-STD-8739.8) provide detailed information about software quality assurance.

5.4 Anomaly Reporting and Tracking

Software anomaly reports (also known as problem reports) are a means to identify and record:

- Software product anomalous behavior and its resolution, including failure to respond properly to nominal and off-nominal conditions.
- Process non-compliance with software, requirements, plans, and standards, including improperly implemented safety measures.
- Deficiencies in documentation and safety data, including invalid requirements.

To help prevent recurrence of software safety-related anomalies, the launch vehicle operator should develop a standardized process to document anomalies, analyze the root cause, and determine corrective actions.

5.5 Previously Developed Software and Computing Systems

Using previously developed software can reduce development time because those components have already undergone design and testing. However, analysis of accidents where software was a contributing factor shows the risks in this approach. Previously developed software includes commercial off-the-shelf software (COTS), government off-the-shelf software (GOTS), and “reused” software. Although another vendor may have developed the software, reducing the risks of using such software remains the responsibility of the operator. These risk reduction efforts should include evaluating the differences between how the software will be used within the new system and how it was used in the previous system. Identifying the abilities and limitations of the previously developed software and computing systems with respect to any safety requirements is also necessary. In addition, an operator should identify the safety-critical computer system functions, perform hazard analyses, and provide validation and verification data in a manner similar to software developed specifically for its vehicle.

5.6 Training

Designing safety into the system requires that personnel involved in system development, production, and operation understand and practice operations and procedures that protect public safety. Training can help ensure that personnel can produce a safe system or operation. In addition, training can be included as a risk mitigation measure; therefore, training can be a critical element in helping to ensure the safety of the public. The launch vehicle operator should develop plans that describe essential training. This training should include, but is not limited to, development tools, software development approaches, software installation, hazard analysis approaches, safety-critical software use, and software maintenance.

5.7 Maintenance

Maintenance engineering ensures that systems and subsystems will remain at the design safety level by minimizing wear-out failures through replacement of failed items and surveillance over possible degraded environments. Maintenance engineering personnel also participate in analyzing the safety implications of proposed maintenance procedures on the ground and in flight. Therefore, the launch vehicle operator should perform activities to aid maintenance and repair of computing system hardware.

Software maintenance differs from hardware maintenance because software does not wear out or degrade in the same way that hardware does. However, software is maintained to correct defects; add or remove features and capabilities; compensate for or adapt to hardware changes, wear-out, or failure; and accommodate changes in other software components.

Because software changes can be expected over the life cycle of the product, a launch vehicle operator should build maintainable software to facilitate those changes and reduce the likelihood of introducing new hazards. The following are some considerations for building maintainable software:

- Planning early for expected changes.
- Using a strong configuration management program.
- Using modular design, where appropriate, to minimize the overall impact of changes.
- Implementing naming conventions for variables to improve code readability.
- Using comment and style coding standards to improve code readability.
- Implementing documentation standards to make important information easy to find.
- Using a standardized set of development tools to reduce the chance of introducing errors in code changes.
- Assuring that design and verification documentation, such as regression tests and test cases, are updated and maintained.

NASA-GB-8719.13 provides additional information on software maintainability.

6.0 LESSONS LEARNED

Appendix C provides a compilation of some accidents and failures where software and computing systems played a critical role. Researchers have studied many of the accidents described here (Greenwell and Knight 2003, Holloway 1999, JSSSC 1999, Leveson 2001, Leveson 2004). The following are some of the broad lessons learned from the study of those accidents:

- Risks associated with software and automated processes are often underestimated or misunderstood.
- Systems engineering efforts to identify problems related to interaction among components, including hardware and software, are often insufficient.
- System safety efforts to identify hazards and risks, including the intended and unintended operations of the software and computing systems, are often insufficient.
- Requirements are often flawed and specifications are often missing.
- Software is often needlessly complex.
- Software is often reused without appropriate safety analyses.
- Software is often not adequately reviewed.
- Test and simulation environments often do not match the operational environment.
- Software changes are often made without reassessing system safety.
- Configuration management processes are often inaccurate.
- Safety aspects of the graphical user interface are often not considered in the system safety analyses.
- Cost and schedule overruns often lead to reduced testing, reduced system understanding, and increased system risk.
- Software can be out of synch with its documentation, leading to misunderstanding and increased risk.

Chan (2001) identified recommendations based on additional lessons learned from U.S. military experience, including the following:

- Testing should take into account possible abuse or bypassing of expected procedures.
- Design and implementation of software and computing systems must be subject to the same safety analysis, review and quality control as other parts of the system.
- An effective mechanism should be in place for documenting and characterizing field problems involving software and computing systems.
- Programmer qualifications are as important as qualifications of any other member of the engineering team.
- The test, verification, and review processes must include end-to-end event review and test.
- Reuse of software components must include review and testing of the integrated components in the new operating environment.

- Specified equations describing physical world phenomenon must be thoroughly defined, with assumptions as to accuracies, ranges, uses, environments, and limitations of the computation.
- Boundary assumptions should be used to generate test cases because the more subtle failures caused by assumptions are usually not covered by ordinary test cases.
- Training must describe the safety-related software functions, such as the possibility of software overrides to operator commands.

Launch vehicle developers should take advantage of these broad lessons learned as well as more specific design-level lessons learned reflected in the generic requirements in appendix A to reduce the possibility of software and computing system failures and mishaps.

APPENDIX A: GENERIC SOFTWARE SAFETY REQUIREMENTS

This appendix provides generic software safety requirements that a launch vehicle operator may use in the design and development of safety-critical software and computing systems. These requirements represent best practices used in the aerospace community. Additional information and requirements can be found in the following references:

- *FAA System Safety Handbook* (2002)
- *Joint Services Software Safety Committee Software System Safety Handbook* (1999)
- *NASA Software Safety Guidebook* (2004)
- *Range Safety User Requirements Manual: Air Force Space Command Range Safety Policies and Procedures* (2004)
- *System Safety Analysis Handbook* (1997)

A.1 General Computer System Requirements

A.1.1 Computer systems should be validated for operation in the intended use and environment. Such validation should include testing under operational conditions and environments.

A.1.2 Under maximum system loads, CPU throughput should not exceed 80 percent of its design value.

A.1.3 Computer system architecture should be single fault tolerant. No single software fault or output should initiate a hazardous operation, cause a critical accident, or cause a catastrophic accident.

A.1.4 Safety-critical computer system flight architecture that will be exposed to cosmic radiation should protect against CPU single event upset and other single event effects. A single event upset occurs when an energetic particle travels through a transistor substrate and causes electrical signals within a component.

A.1.5 Sensitive components of computer systems should be protected against the harmful effects of electromagnetic radiation, electrostatic discharges, or both.

A.1.6 The computer system should periodically verify that safety-critical computer hardware and software safety-critical functions, including safety data transmission, operate correctly.

A.1.7 The computer system should periodically verify the validity of real-time software safety data, where applicable.

A.1.8 Software should process the necessary commands within the time-to-criticality of a hazardous event.

A.1.9 Memory allocation should occur only at initialization.

A.1.10 If the system begins to use areas of memory that are not part of the valid program code, the system should revert to a safe state.

A.1.11 Memory partitions, such as RAM, should be cleared before loading software.

A.1.12 Prerequisite conditions for the safe execution of an identified hazardous command should exist before starting the command. Examples of these conditions include correct mode, correct configuration, component availability, proper sequence, and parameters in range. If prerequisite conditions have not been met, the software should reject the command and alert the crew, ground operators, or controlling executive.

A.1.13 Provisions to protect the accessibility of memory region instructions, data dedicated to critical software functions, or both, should exist.

A.1.14 Software should provide proper sequencing, including timing, of safety-critical commands.

A.1.15 Where practical, software safety-critical functions should be performed on a standalone computer. If that is not practical, software safety-critical functions should be isolated to the maximum extent practical from non-critical functions.

A.1.16 Documentation describing the software and computing system should be developed and maintained to facilitate maintenance of the software.

A.1.17 The software should be annotated, designed, and documented for ease of analysis, maintenance, and testing of future changes to the software.

A.1.18 Interrupt priorities and responses should be specifically defined, documented, analyzed, and implemented for safety impact.

A.1.19 Critical software design and code should be structured to enhance comprehension of decision logic.

A.1.20 Software code should be modular in an effort to reduce logic errors and improve logic error detection and correction functions.

A.1.21 The software should be initiated and terminated in a safe state.

A.1.22 Critical hardware controlled by software should be initialized to a known safe state.

A.2 Computing System Power

A.2.1 Computer systems should be powered up, restarted, and shutdown in a safe state.

A.2.2 A computer system should not enter a hazardous state as a result of an intermittent power transient or fluctuation.

A.2.3 If a single failure of primary power to a computer system or computer system component occurs, then that system or some cooperating system should take action automatically to transition to a stable state.

A.2.4 Software used to power up safety-critical systems should power up the required systems in a safe state.

A.3 Anomaly and Failure Detection

A.3.1 Single event system failures should be protected against by employing mitigating approaches, such as redundancy, error-correcting memory, and voting between parallel CPUs.

A.3.2 Before initiating hazardous operations, computer systems should perform checks to ensure that they are in a safe state and functioning properly. Examples include checking safety-critical circuits, components, inhibits, interlocks, exception limits, safing logic, memory integrity, and program loads.

A.3.3 Failure of software safety-critical functions should be detected, isolated, and recovered from in a manner that prevents catastrophic and critical hazardous events from occurring.

A.3.4 Software should provide error handling to support software safety-critical functions. The following hazardous conditions and failures, including those from multiple sources, should be detected:

- Input errors. Data or sequences of data passed to software modules, either by human input, other software modules, or environmental sensors, that are outside a specified range for safe operation.
- Output errors. Data output from software modules that are outside a specified range for safe operation.
- Timing errors. The state when software-timed events do not happen according to specification.
- Data transmission errors.
- Memory integrity loss.
- Data rate errors. Greater than allowed safe input data rates.
- Software exceptions. “Divide by zero” or “file not found.”
- Message errors. Data transfer messages corrupted or not in the proper format.
- Logic errors. Inadvertent instruction jumps.

A.3.5 Watchdog timers or similar devices should be used to ensure that the microprocessor or computer operates properly. For example, a watchdog timer should be used to verify events within an expected time budget or to ensure that cyclic processing loops complete within acceptable time constraints.

A.3.6 Watchdog timers or similar devices should be designed, so the software cannot enter an inner loop and reset the timer or similar device as part of that loop sequence.

A.4 Anomaly and Failure Response

A.4.1 Software should provide fault containment mechanisms to prevent error propagation across replaceable unit interfaces.

A.4.2 All anomalies, software faults, hardware failures, and configuration inconsistencies should be reported to the appropriate system operator, safety official, or both, consoles in real time, prioritized as to severity, and logged to an audit file. The display should

- distinguish between read and unread anomaly alerts,
- support reporting multiple anomalies,
- distinguish between anomaly alerts for which corrective action has been taken and those that still require attention, and
- distinguish between routine and safety-critical alerts.

A.4.3 Upon detecting anomalies or failures, the software should

- remain in or revert to a safe state,
- provide provisions for safing the hardware subsystems under the control of the software,
- reject erroneous input, and
- ensure the logging of all detected software safety-critical function system errors.

A.4.4 Upon detecting a failure during launch vehicle processing, the software should maintain the Flight Safety System (FSS) in its current state in addition to meeting the requirements in paragraph 4.3 of this appendix. The software should maintain the FSS in the safe state. After the FSS is readied, the software should retain the FSS in the readied state. When the FSS receiver is on internal power, the software should maintain the FSS receiver on internal power. During flight, all detected FSS-related system errors should be transmitted to the safety official.

A.4.5 Details of each anomaly should be accessible with a single operator action.

A.4.6 Automatic recovery actions taken should be reported to the crew, operator, or controlling executive. There should be no necessary response from crew or ground operators to proceed with the recovery action.

A.4.7 Override commands should require multiple operator actions.

A.4.8 Software that executes hazardous commands should notify the initiating crew, ground operator, or controlling executive upon execution or provide the reason for failure to execute a hazardous command.

A.4.9 Hazardous processes and safing processes with a time-to-criticality such that timely human intervention may not be available should be automated. Additionally, such processes should not require human intervention to begin, accomplish interim tasks, or complete.

A.4.10 The software should notify the crew, ground operators, or controlling executive during or immediately after completing an automated hazardous or safing process.

A.4.11 After correction of erroneous entry, the software should provide positive confirmation of a valid data entry. The software should also provide an indication that the system is functioning properly.

A.5 Maintenance, Inhibits, and Interlocks

A.5.1 Systems should include hardware and software interlocks and software controllable inhibits, as necessary, to mitigate hazards when performing maintenance or testing.

A.5.2 Interlocks should be designed to prevent an inadvertent override.

A.5.3 Interlocks that are required to be overridden should not be autonomously controlled by a computer system, unless dictated by a timing requirement.

A.5.4 Interlocks that are required to be overridden and are autonomously controlled by a computer system should be designed to prevent an inadvertent override.

A.5.5 The status of all interlocks should be displayed on the appropriate operator consoles.

A.5.6 An interlock should not be left in an overridden state once the system is restored to operational use.

A.5.7 A positive indication of interlock restoration should be provided and verified on the appropriate operator consoles before restoring a system to its operational state.

A.5.8 Software should make available status of all software controllable inhibits to the crew, ground operators, or controlling executive.

A.5.9 Software should accept and process crew, ground operator, or controlling executive commands to activate and deactivate software controllable inhibits.

A.5.10 Software should provide an independent and unique command to control each software controllable inhibit.

A.5.11 Software should incorporate the ability to identify and display the status of each software inhibit associated with hazardous commands.

A.5.12 Software should make available current status on software inhibits associated with hazardous commands to the crew, ground operators, or controlling executive.

A.5.13 All software inhibits associated with a hazardous command should have a unique identifier.

A.5.14 If an automated sequence is already running when a software inhibit associated with a hazardous command is executed, the sequence should complete before the software inhibit is started.

A.5.15 Software should have the ability to resume control of an inhibited operation after deactivation of a software inhibit associated with a hazardous command.

A.5.16 The state of software inhibits should remain unchanged after the execution of an override.

A.6 Human-Computer Interface

A.6.1 The system should be designed such that the operator may exit current processing to a known stable state with a single action and have the system revert to a safe state.

A.6.2 Computer systems should minimize the potential for inadvertent actuation of hazardous operations.

A.6.3 Only one operator at a time should control safety-critical computer system functions.

A.6.4 Operator-initiated hazardous functions should require two or more independent operator actions.

A.6.5 Software should provide confirmation of valid command entries, data entries, or both, to the operator.

A.6.6 Software should provide feedback to the operator that indicates command receipt and status of the operation commanded.

A.6.7 Software should provide the operator with real-time status reports of operations and system elements.

A.6.8 Error messages should distinguish safety-critical states and errors from non-safety-critical states and errors.

A.6.9 Error messages should be unambiguous.

A.6.10 Unique error messages should exist for each type of error.

A.6.11 The system should ensure that a single failure or error cannot prevent the operator from taking safing actions.

A.6.12 The system should provide feedback for any software safety-critical function actions not initiated.

A.6.13 Safety-critical commands which require several seconds or longer to process should provide a status indicator to the operator indicating that processing is occurring.

A.6.14 Safety-critical operator displays and interface functions should be concise and unambiguous. Where possible, such displays should be duplicated using separate display devices.

A.7 Computing System Environment-Software Interface

A.7.1 The developer should identify the situations in which the application can corrupt the underlying computing environment.

A.7.2 The developer should check for system data integrity at startup.

A.7.3 The system should provide for self-checking of the programs and computing system execution.

A.7.4 Periodic checks of memory, instruction, and data busses should be performed.

A.7.5 Parity checks, checksums, or other techniques should be used to validate data transfer.

A.8 Operations

A.8.1 Operational checks of testable software safety-critical functions should be made immediately before performance of a related safety-critical operation.

A.8.2 Software should provide for flight or ground crew forced execution of any automatic safing, isolation, or switchover functions.

A.8.3 Software should provide for flight or ground crew forced termination of any automatic safing, isolation, or switchover functions.

A.8.4 Software should provide procession for flight or ground crew commands in return to the previous mode or configuration for any automatic safing, isolation, or switchover function.

A.8.5 Software should provide for flight or ground crew forced override of any automatic safing, isolation, or switchover functions.

A.8.6 Hazardous payloads should provide failure status and data to core software systems. Core software systems should process hazardous payload status and data to provide status monitoring and failure annunciation.

A.8.7 The system should have at least one safe state identified for each logistic and operational phase.

A.8.8 Software control of critical functions should have feedback mechanisms that give positive indications of the function's occurrence.

A.8.9 The system and software should ensure that design safety requirements are not violated under peak load conditions.

A.8.10 The system and software should ensure that performance degradation caused by factors, such as memory overload and counter overflow, does not occur over time.

A.9 Validation and Verification

A.9.1 A system safety engineering team should analyze the software throughout the design, development, and maintenance process to verify and validate that the safety design requirements have been correctly and completely implemented. Test results should be analyzed to identify potential safety anomalies that may occur.

A.9.2 If simulated items, simulators, and test sets are required, the system should be designed such that the identification of the devices is fail safe. The design should also assure that operational hardware can not be inadvertently identified as a simulated item, simulator, or test set.

A.9.3 The launch vehicle operator should use a problem-tracking system to identify, track, and disposition anomalies during the verification process.

A.9.4 The operator should have the ability to review logged system errors.

A.9.5 For software safety-critical functions, the developer should provide evidence that testing has addressed not only nominal correctness but also robustness in the face of stress. Such testing may involve stimulus and response pairs to demonstrate satisfaction of functional requirements. This approach should include a systematic plan for testing the behavior when capacities and rates are extreme. As a minimum, the plan would identify and demonstrate the behavior of safety-critical software in the face of the failure of various other components. Examples include having no or fewer input signals from a device for longer periods than operationally expected or, conversely, receiving more frequent input signals from a device than operationally expected.

A.9.6 The developer should provide evidence of the following:

- Independence of test planning, execution, and review for safety-critical software; to that end, someone other than the individual developer should develop, review, conduct, and interpret unit tests.
- Rate and severity of errors of software safety-critical functions exposed in testing diminishes as the system approaches operational testing.
- Tests of software safety-critical functions represent a realistic sampling of expected operational inputs.

A.9.7 Software testing should include the following:

- Hardware and software input failure modes.
- Boundary, out-of-bounds, and boundary crossing conditions.
- Minimum and maximum input data rates in worst-case configurations to determine the system's abilities and responses to these conditions.
- Input values of zero, zero crossing, and approaching zero from either direction and similar values for trigonometric functions.

A.9.8 Interface testing should include operator errors during safety-critical operations to verify safe system response to these errors. Issuing the wrong command, failing to issue a command, and issuing commands out of sequence should be among the conditions tested.

A.9.9 Software safety-critical functions in which changes have been made should be subjected to complete regression testing. The regression tests should be maintained and updated as necessary.

A.9.10 Where appropriate, software testing should include duration stress testing. The stress test periods should continue for at least the maximum expected operating time for the system. Operators should conduct testing under simulated operational environments. In addition, software testing should examine the following items:

- Inadvertent hardware shutdown and power transients.
- Error handling.
- Execution path coverage, with all statements completed and every branch tested at least once.

A.9.11 The launch vehicle operator should evaluate equations and algorithms to ensure that they are correct, complete, and satisfy safety requirements.

A.9.12 Non-operational hardware and software required for testing or maintenance should be clearly identified.

A.9.13 Existing code compiled with a new compiler or new release of a compiler should be regression tested.

A.9.14 Operators should not use beta test versions of language compilers or operating systems for safety-critical functions.

A.9.15 A launch operator should document and maintain test results in test reports.

A.10 Configuration Management

A.10.1 Software safety-critical functions and associated interfaces should be put under formal configuration control as soon as a software baseline is established.

A.10.2 A software configuration control board should be created to set up configuration control processes and pre-approve changes to configuration-controlled software.

A.10.3 The software configuration control board should include a member from the system safety engineering team, tasked with the responsibility of evaluating all proposed software changes for potential safety impacts.

A.10.4 Object code patches should not be performed without specific approval.

A.10.5 All software safety-critical functions should be identified as “safety-critical.”

A.10.6 The software configuration management process should include version identification, access control, and change audits. In addition, the ability to restore previous revisions of the system should be maintained throughout the entire life cycle of the software.

A.10.7 All software changes should be evaluated for potential safety impact, and the FAA should be advised of proposed changes that impact safety.

A.10.8 All software changes should be coded with a unique version identification number in the source code, then compiled and tested before introduction into operational equipment.

A.10.9 All software safety-critical functions and associated interfaces should be under configuration control.

A.10.10 Appropriate safeguards should be implemented to prevent non-operational hardware and software from being inadvertently identified as operational.

A.10.11 Test and simulation software should be positively identified as non-operational.

A.10.12 The run-time build should only include software that is built from contractor-developed software source modules, COTS software object modules that are traceable to a requirement, or derived requirement identified in the requirements or design documentation.

A.11 Quality Assurance

A quality assurance function should be implemented to verify that objectives are being satisfied and deficiencies are detected, evaluated, tracked, and resolved. This quality assurance function includes audits; code walk throughs; and inspections of elements and processes, such as plans, standards, problem-tracking systems, configuration management systems, and system life cycles.

A.12 Security

A.12.1 The software should be designed to prevent unauthorized system or subsystem interaction from initiating or sustaining a software safety-critical function sequence.

A.12.2 The system design should prevent unauthorized or inadvertent access to or modification of the software (source or assembly) and object code. This security measure includes preventing self-modification of the code.

A.13 Software Design, Development, and Test Standards

Software should be designed, developed, and tested in a manner that complies with IEEE 12207, *Standard for Information Technology*, or its equivalent.

A.14 Software Coding Practices

Software developers should apply the software coding practices described in appendixes D and E of the Joint Services Software System Safety Committee, *Software System Safety Handbook*, dated December 1999, or its equivalent.

A.15 Software Reuse

Reused software encompasses software developed for other projects by the developer as well as any open source or public domain software selected for the project. Such software should be evaluated to determine if it is a software safety-critical function. Reused safety-critical software should comply with all safety-critical provisions required of newly developed software. For example, a launch operator should analyze reused software that performs a safety-critical function for the following items:

- Correctness of new or existing system design assumptions and requirements.
- Impacts on the overall system as the reused software runs on or interfaces with replaced equipment, new hardware, or both.
- Changes in the environmental or operating conditions.
- Impacts to existing hazards.
- Correctness of the interfaces between system hardware; other software; and crew, ground operators, or controlling executive.
- Safety-critical computing system functions compiled with a different compiler.

A.16 Commercial Off-The-Shelf (COTS) Software

A.16.1 When employing commercial off-the-shelf software, a launch operator should ensure that every software safety-critical function that the software supports is identified and satisfies the requirements of this appendix.

A.16.2 Software hazard analyses should be performed on all COTS software used for software safety-critical functions.

A.16.3 Software safety-critical functions identified in COTS software should comply with all software safety-critical function requirements or be validated for intended use and environment. Compliance, validation method, and evidence are subject to FAA approval and should be documented.

APPENDIX B: SOFTWARE AND COMPUTING SYSTEM HAZARD ANALYSES

This appendix describes two methods for conducting software and computing system hazard analyses: Software Failure Modes and Effects Analysis (SFMEA) and Software Fault Tree Analyses (SFTA). Examples of the use of SFMEA and SFTA are provided. Other approaches may be acceptable to FAA. Note that the analysis method used and the level of detail in that analysis will be made based on the complexity of the system, difficulty of the operations, and scope of the program.

B.1 Software Failure Modes and Effects Analysis

As described in the FAA/AST *Guide to Reusable Launch and Reentry Vehicle Reliability Analyses*, a Failure Modes and Effects Analysis (FMEA) is a bottom-up, inductive, reliability and safety analysis method used to identify potential failure modes, effects on the system, risk reduction measures, and safety requirements. Although the steps to performing a SFMEA are similar to those of a hardware FMEA, an SFMEA differs in the following ways:

- Hardware failure modes generally include aging, wear-out, and stress, while software failure modes are functional failures resulting from software faults.
- Hardware FMEA analyzes both severity and likelihood of the failure, while an SFMEA usually analyzes only the severity of the failure mode.

Software Failure Modes and Effects Analysis allows for systematic evaluation of software and computing system failure modes and errors. In addition, this analysis helps to prioritize the verification effort to focus on those functions that have the most influence on the safety of the system. One procedure for performing an SFMEA is as follows:

1. Define the system to be analyzed. The system definition includes identification of modules. In addition, system definitions can include a description of interfaces between software and other systems, flow charts describing data flow or operations, logic diagrams, and user documentation.
2. Categorize the system into elements to be analyzed.
3. Identify the software and computing system failure modes or software error.
4. Identify the potential causes (specific faults leading to the error or failure). Identifying the specific causes helps to define mitigation measures and test cases.
5. Identify the local and system effects of each failure mode or software error.
6. Identify controls and requirements to mitigate the risks for each failure mode.
7. Document the analysis using an SFMEA worksheet.

In the majority of cases, failure modes for hardware components are understood and can be based on operational experience. A hardware FMEA can be based on the known hardware failures for a particular design or class of piece part, component, or subsystem. These hardware failures often result from such factors as wear-out, unanticipated stress, or operational variation. For software, such operational experience often does not exist. Software does not break or fall out of tolerance in the same way hardware does; therefore, software and computing system failure modes or software errors must be identified using generalized classifications. Table B.1-1 shows one example of a classification set derived from information in such standards as IEEE STD 1044-1993. This table does not list all possible faults and failures; therefore, an operator should consider these and others specific to its system when performing software hazard analyses.

Table 1. Example classification of software and computing system errors

Software and Computing System Failure Mode (Software Error) Class	**Specific Software and Computing System Faults and Failures**
Calculation	• Inappropriate equation for a calculation. • Incorrect use of parenthesis. • Inappropriate precision. • Round fault (or truncation fault). • Lack of convergence in calculation. • Operand incorrect in equation. • Operator incorrect in equation. • Sign fault. • Capacity overflow, underflow, or both. • Inappropriate accuracy. • Use of incorrect instruction.
Data	• Undefined data. • Non-initialized data. • Data defined several times. • Incorrect adapt protection. • Variable type incorrect. • Range incorrect. • Wrong use of data (bit alignment, global data). • Fault in the use of complex data (record, array, pointer). • No use of data. • Data stuck at some value.

Software and Computing System Failure Mode (Software Error) Class	**Specific Software and Computing System Faults and Failures**
Interface	• Data corruption. • Bad parameters in call between two procedures. • No or null parameters in the call between two procedures. • Non-existent call between two procedures. • Wrong call between two procedures. • Inappropriate end-to-end numerical resolution. • Wrong message communication (bad error handling). • Empty or no message communication (bad or no error handling). • Incorrect creation, deletion, or suspension of a task. • Software responds incorrectly to no data. • Wrong synchronization between tasks (task not invoked because of its low priority). • Incorrect task blocking. • Wrong commands or messages given by the user, operator, or both. • No commands given by the user, operator, or both. • Commands not given in time by the user, operator, or both. • Commands given at wrong time by the user, operator, or both.
Logic	• Wrong order of sequences (modules called at wrong time). • Wrong use of arithmetic or logical instruction. • Wrong or missing test condition. • Wrong use of branch instruction. • Timing overrun. • Missing sequence. • Wrong use of a macro. • Wrong or missing iterative sequence. • Wrong algorithm. • Shared data overwritten. • Unnecessary function. • Unreachable code. • Dead code.
Environment	• Compiler error. • Wrong use of tools options (optimize, debug). • Bad association of files during code link. • Change in operating system leads to software bug. • Change in third-party software leads to software bug.

Software and Computing System Failure Mode (Software Error) Class	**Specific Software and Computing System Faults and Failures**
Hardware	• CPU overload. • Memory overload. • Unexpected shutdown of the computer. • Wrong file writing. • Wrong interrupt activation. • Wrong data into register or memory. • No file writing. • No interrupt activation. • No data into register or memory. • Loss of operator visualization (loss of screen display). • Untimely file writing. • Untimely data into memory or register. • Untimely interrupt activation. • Untimely operator visualization.

Table 2 shows an SFMEA worksheet for functions and computing system hardware components in a hypothetical RLV. While the analyses in these examples are focused on software functions, an SFMEA can be performed at any level, for example, a software package or module. Analyses at lower levels, such as at the code, provide the most information but also require the most resources. The scope of the analysis will depend on the particular software and development program. Examples of SFMEA developed for other industries are provided in Czerny (2005), Dunn (2002), Feng and Lutz (2005), Ozarin (2006), and Wood (1999).

Performing an SFMEA as early as possible in the development process is desirable. Note, however, the software design is highly subject to change because designers continually make beneficial modifications during development. Therefore, updating the SFMEA throughout the development process to reflect these changes is important.

Table 2. Example software and computing system Failure Modes and Effects Analysis worksheet

Item No.	Software or Computing System Element	Failure Mode or Software Error	Error Cause (Specific Fault Type)	Local Effect	System Effect or Hazard	Risk Mitigation Measures
PS-1	Function: PROP_SENS Acquire temperature and pressure sensor inputs from propulsion system, and provide information to flight control modules and automated shutdown routines.	Function fails to work or performs incompletely because of logic, data, or interface errors.	• Wrong use of branch instruction. • Data out of range. • Missing data. • Non-existent or incorrect call between procedures. • Missing error-handling routine. • Function called at wrong time.	No sensor readings obtained from the propulsion system.	• Continuing to operate with last sensor inputs. • Failing to detect out-of-range condition. • Failing to issue proper abort and propulsion shutdown commands.	• Using a separate software execution monitoring function to detect whether the function was completed. • Verifying sensors before flight.
PS-2	Function: PROP_SENS Acquire temperature and pressure sensor input from propulsion system and provide information to flight control modules and automated shutdown routines.	Function works incorrectly because of calculation, logic, data, or interface errors.	• Incorrect conversion calculation. • Wrong use of branch instruction. • Wrong use of data. • Data out of range. • Missing data. • Missing error-handling routine. • Function called at wrong time.	Incorrect sensor signals received from the propulsion system.	• Using incorrect input; therefore, providing incorrect output. • Failing to issue proper abort and propulsion shutdown commands.	• Using a separate software function to detect out of range conditions for temperature and pressure. • Providing independent temperature and pressure readings to pilots to use for manual shutdown purposes. • Verifying sensors before flight.

Table 2. Example Software and Computing System Failure Modes and Effects Analysis worksheet (cont'd)

Item No.	Software or Computing System Element	Failure Mode or Software Error	Error Cause (Specific Fault Type)	Local Effect	System Effect or Hazard	Risk Mitigation Measures
CV-1	Function: CLOSE_VALVE When limits are exceeded command the main fuel and oxidizer valves to close.	Function fails to work or performs incompletely because of logic, data, or interface errors.	• Wrong use of branch instruction. • Data out of range or incorrect. • Non-existent or incorrect call between procedures. • Missing error-handling routine.	Signal is not sent to the valve actuators.	Failing to close valves, resulting in continued thrust, flight outside of operating area, or possible loss of vehicle.	• Using a separate software execution monitoring function to detect whether the function was completed. • Making manual shutdown procedures available.
GPS-1	Function: GPS_RECEIVE • Acquire GPS signal. • Send vehicle position to other functions. • Abort if location data out of range.	Function fails to execute or executes incompletely because of logic, data, or interface errors.	• Wrong use of branch instruction. • Data out of range or incorrect (loss of GPS signal). • Non-existent or incorrect call between procedures. • Missing error-handling routine.	Position information is not provided.	• Using incorrect input or having no GPS location data; therefore, providing incorrect output. • Failing to issue proper abort and propulsion shutdown commands.	• Using a separate software function to detect out of range conditions, including location values and signal strength. • Having the main computer initiate an abort if conditions are out of range. • Performing GPS checks before flight.

Table 2. Example Software and Computing System Failure Modes and Effects Analysis worksheet (cont'd)

Item No.	Software or Computing System Element	Failure Mode or Software Error	Error Cause (Specific Fault Type)	Local Effect	System Effect or Hazard	Risk Mitigation Measures
CS-1	Main CPU.	Loss of main computer.	• Overload of CPU • Loss of power from on-board batteries. • Inadvertent shutdown.	Loss of all safety-critical computer and software functions.	• Continuing to operate with last sensor inputs. • Failing to detect out-of-range condition, causing failure to issue proper abort and propulsion shutdown commands.	• Using a watchdog timer to detect computing system functionality and trigger a reboot of the main CPU. • Implementing CPU self-tests and hardware diagnostics to detect failure. • Initiating abort sequences using a separate on-board CPU. • Making manual shutdown procedures available.
WD-1	Watchdog timer.	Watchdog timer failure.	• Loss of power. • Mechanical or electrical failure.	No system available to monitor CPU loss.	• Failing to detect loss of CPU. • Failing to issue proper abort commands if main CPU lost.	• Running watchdog computer and main CPU off separate power sources. • Verifying watchdog timer before flight. • Making manual abort procedures available.

B.2 Software Fault Tree Analysis

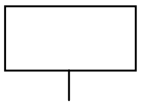
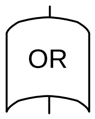
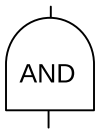
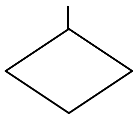
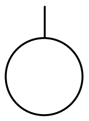
A top-down, deductive study of system reliability, Fault Tree Analysis (FTA) graphically depicts the sequence of events that can lead to an undesirable outcome. An FTA generates a fault tree, which is a symbolic logic model of the failures and faults. As an aid for system safety improvement, an FTA is often used to model complex processes. For example, an FTA may be used to estimate the probability that a top-level or causal event will occur, identify systematically possible causes leading to that event, and document the results of the analytic process to provide a baseline for future studies of alternate designs.

A Software Fault Tree Analysis (SFTA) is an extension of the system FTA in which software and computing system contributors to an undesirable event are identified and analyzed. While a hardware FTA can be quantitative or qualitative, an SFTA is rarely quantitative because of the difficulties in obtaining failure probabilities for software. An SFTA produces safety requirements that can then be implemented in the software life cycle.

Standard logic symbols are used in constructing an SFTA to describe events and logical connections. Table 3 shows the most common symbols. The FAA/AST *Guide to Reusable Launch and Reentry Vehicle Reliability Analyses* provides additional symbols and information on SFTA. The process for performing an SFTA is as follows:

1. Identify the undesirable events that require analysis. Usually these occurrences are called pivotal events – events that could ultimately lead to failure of the vehicle or system. Each pivotal event is a top event for the fault tree, and a new tree is required for each top event. The top event is often determined from other analyses, such as a hazard analysis, FMEA, or known undesirable event, such as a mishap.
2. Define the scope of the analysis to determine the level of depth of the analysis needed for each undesirable event. The level of depth may be determined based on the application of the analysis. In some cases, for example, analyzing broad functions may suffice. Other cases may require analyzing errors in specific modules.
3. Identify causes leading to the undesirable event, known as first-level contributors to the top event. Contributors must be independent of each other. For example, for a top event of "Incorrect navigation data on flight control display," the events "data not calculated correctly" and "inappropriate equation used for calculations" are not independent events. Use of an inappropriate equation may have led to calculating the data incorrectly. To determine events and contributors, data gathering may be required. Sources of this information include specifications, drawings, and block diagrams.
4. Link the first-level contributors to the top event by a logic gate.
5. Identify the second-level contributors to the first-level events.
6. Link the second-level contributors to the first-level contributors.

Table 3. Common fault tree logic and event symbols

Symbol	Description
	Top Event – Foreseeable, undesirable occurrence (also used for an intermediate event).
	"OR" Gate – Any of the events below gate will lead to an event above the gate.
	"AND" Gate – All events below gate must occur for event above gate to occur.
	Undeveloped Event – An event not further developed because of a lack of need, resources, or information.
	Initiator (Basic Event) – Initiating fault or failure, not developed further (marks limit of analysis).

7. Repeat until the analysis reaches a desired level. The bottom-most contributors are known as initiators or basic events.
8. Evaluate the tree to determine the validity of the input and failure paths.
9. Identify specific safety requirements.
10. Document the SFTA results.

An SFTA allows for systematic evaluation of the risks of complex software and computing systems. Using an SFTA helps to discover common cause failures and single-point failures, critical fault paths, and design weaknesses and to identify the best places to build in fault tolerance. In addition, an SFTA helps to prioritize the verification effort to focus on those functions with a large amount of influence on the safety of the system. Czerny (2005), Dunn (2002), Dehlinger and Lutz (2004), and Gowen (1996) provide examples of SFTA developed for other industries.

In developing an SFTA, a developer normally starts with a general FTA that describes the potential impacts of a safety-critical software function with respect to a large system.

Figure 2 shows an example of a fault tree for engine shutdown failure that includes hardware, software, and procedures. The contributing event, “Software or computing system error,” can be expanded further. Figure 3 shows a portion of a fault tree expanding this undesirable event. Note that the “logic error,” “data error,” and “data input error” basic events could be expanded further if necessary to identify specific areas of concern, such as out of range variables, logic sequences out of order, or other faults identified in table 1.

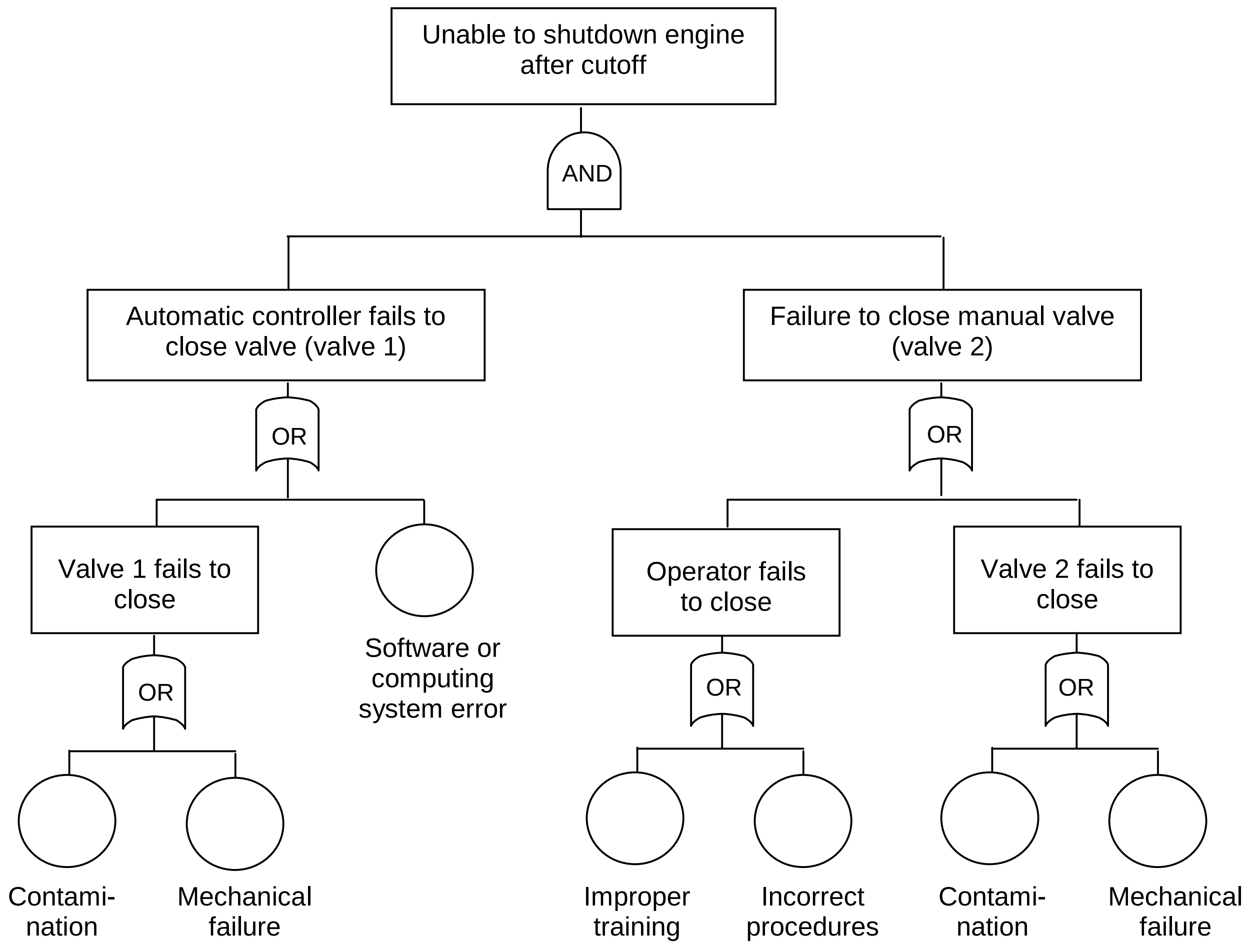


Figure 2. Fault tree for engine shutdown failure

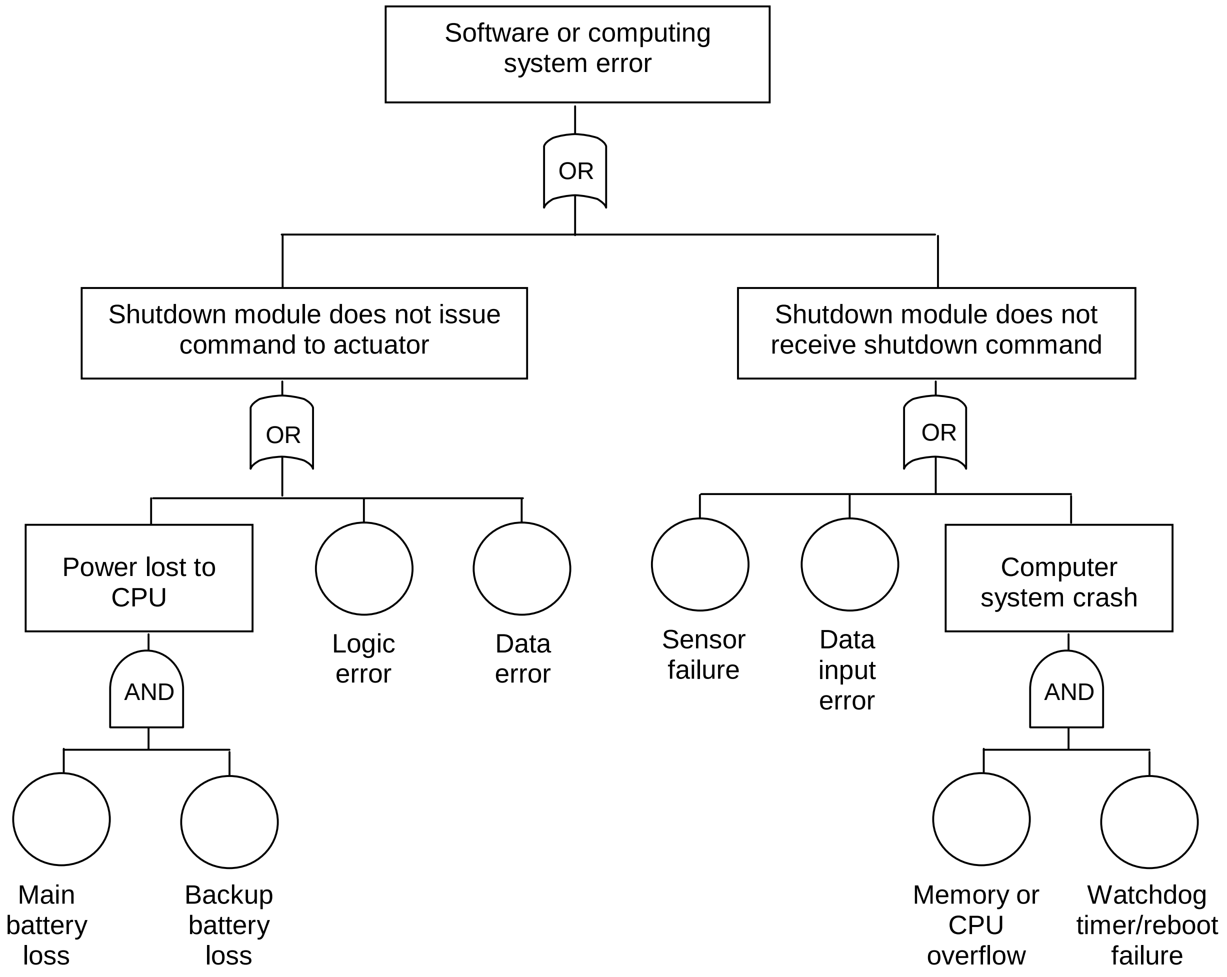

Software or computing system error
OR
Shutdown module does not issue command to actuator
Shutdown module does not receive shutdown command
OR
OR
Power lost to CPU
Logic error
Data error
Sensor failure
Data input error
Computer system crash
AND
AND
Main battery loss
Backup battery loss
Memory or CPU overflow
Watchdog timer/reboot failure

Figure 3. Fault tree for software or computing system errors

APPENDIX C: SPACE VEHICLE FAILURES AND AIRCRAFT ACCIDENTS

C.1 Space Vehicle Failures

Software and its associated computing system hardware have played a significant role in the root cause of several high-profile space vehicle failures, as described in various accident investigation reports and studies. Although the following is not a comprehensive list of all failures where software played a role, the descriptions help provide an understanding of the types of failures that can be traced to software and computing systems and provide lessons learned for the design of future systems.

Phobos 1. The Phobos 1 spacecraft was launched on July 7, 1988, on a mission to conduct surface and atmospheric studies of Mars. The vehicle operated normally until routine attempts to communicate with the spacecraft failed on September 2, 1988, and the mission was lost. Examination of the failure showed that a ground control operator had omitted a single letter in a series of digital commands sent to the spacecraft. The computer mistranslated this command and started a ground checkout test sequence, deactivating the attitude control thrusters. As a result, the spacecraft lost its lock on the Sun. Because the solar panels pointed away from the Sun, the on-board batteries were eventually drained, and all power was lost.

A lack of specifications taking the human and software interface into account contributed to the failure. Additionally, error-checking functions had been turned off during the data transfer operation.

Lesson Learned: Error checking and isolating test software from flight software are important aspects of software assurance (Norman 1990, Perminov 1999).

Clementine. The Deep Space Program Science Experiment, also known as the Clementine spacecraft, was launched on January 25, 1994. The spacecraft entered lunar orbit, functioned flawlessly, and departed from the Moon on May 3, 1994, to rendezvous with its target, asteroid 1620 Geographos. However, 4 days later, a flaw in the software resulted in the computer firing the attitude control thrusters until the supply of propellant had been exhausted. The malfunction left the spacecraft in a stable spin that, when combined with the spacecraft’s heliocentric orbit, would ultimately prevent the generation of adequate power to operate the spacecraft. This condition led to abandonment of the mission.

Although the root cause of the problem could not be definitively determined, some researchers have suggested that a floating-point exception may have caused the computer to crash, allowing the thrusters to operate continuously. Inadequate testing, tight schedule, and cost pressures also may have increased the chances of failure.

Lesson Learned: A watchdog timer may have been used to automatically reset the computer and avert failure (Chapman and Regeon 1996, Ganssle 2000, Harland and Lorenz 2005).

Ariane 501. On June 4, 1996, the Ariane 5 launch vehicle veered off course and broke up approximately 40 seconds into launch. The vehicle started to disintegrate because of high aerodynamic loads resulting from an angle of attack greater than 20 degrees. This condition led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher. This improper angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcain main engine. The on-board computer software commanded these nozzle deflections based on data received from the active Inertial Reference System. Ultimately, these improper deflections resulted from specification and design errors in the Inertial Reference System software, including improper error handling. An unexpected horizontal velocity component led to an overflow condition which was not handled properly by the software.

Reused software from the Ariane 4 program, including the exception handling code used in the Inertial Reference System, contributed to the failure. The source of the fault occurred in a function that was not required for Ariane 5, but rather was a function carried over from the Ariane 4 software. The development team believed that faults would be caused by a random hardware failure, handled by redundancy in the hardware. However, because the problem was a requirements problem instead of a random hardware failure, both the primary and backup Inertial Reference Systems shutdown nearly simultaneously from the same cause. In addition, no end-to-end tests were conducted to verify that the Inertial Reference System and its software would behave correctly when subjected to the countdown sequence, flight time sequence, and trajectory of Ariane 5.

Lesson Learned: Multiple factors can contribute to failure, including a misunderstanding of the software risks, especially of reused software; complex software design; insufficient system engineering efforts; flawed requirements and failure to fully analyze those requirements; and insufficient testing (Lions 1996, O'Halloran 2005).

Delta III/Galaxy. On August 27, 1998, the first Boeing Delta III ever flown was launched from Pad 17B at Cape Canaveral Air Station, Florida. Its mission was to place the GALAXY X commercial communications spacecraft into a nominal transfer orbit. At 65 seconds after liftoff, the air-lit Solid Rocket Motors (SRMs) ceased to swivel, leaving two motors in positions that helped overturn the vehicle. The vehicle yawed about 35 degrees. Approximately 71 seconds after lift-off, it began to disintegrate at an altitude of about 60,000 feet. A destruct signal was sent 75 seconds into the flight, which completed destruction of the vehicle. Analysis revealed that between 55 and 65 seconds into the flight, roll oscillations around 4 Hz prompted the control system of the vehicle to gimbal its three swiveling SRMs. The control system software commanded the system to respond to the oscillation, and the SRMs gimbaled with these commands until the hydraulic system ran out of fluid. Once the hydraulic fluid was expended, the oscillations appeared to smooth out. Unfortunately, however, after the hydraulic fluid had been expended, two of the three swiveling SRMs were stuck in the wrong position, and wind shear forced the Delta III to yaw and break up 7 seconds later.

The review team concluded that the flight would not have failed if the control system software had not commanded the system to respond to the 4-Hz roll oscillations because the vehicle oscillations would have smoothed out on their own. As a result of the investigation, Boeing changed an instruction to the flight control system, so the software would identify and ignore the 4-Hz roll oscillation in subsequent Delta III flights.

Lesson Learned: An inadequate understanding of the interactions between software and hardware could lead to failure (Boeing 1998).

Zenit/Globalstar. On September 9, 1998, a two-stage Ukrainian-built Zenit 2 rocket, carrying 12 Globalstar satellites, was launched from Baikonur, Kazakstan. According to the National Space Agency of Ukraine, the second stage of the booster rocket shutdown at approximately 276 seconds into flight. The nose cone carrying the 12 satellites automatically disengaged from the booster rocket with the shutdown and fell to Earth in remote southern Siberia. The booster rocket followed.

Although the root cause of the failure could not be definitively confirmed, a malfunction of the flight control computers or software, which led to the premature shutdown of the second stage, was the most likely cause. Telemetry data indicated that two of the three primary flight computers shut down, a situation that left the third computer unable to control the vehicle, resulting in the cutoff of the engine.

Lesson Learned: A lack of understanding of the risks associated with software and computing systems can lead to failure (Wired News 1998, Woronowycz 1998).

Mars Climate Orbiter. The Mars Climate Orbiter (MCO) was launched on December 11, 1998, and was lost on September 23, 1999, as it entered the Martian atmosphere in a lower than expected trajectory. The investigation board identified the failure to use metric units in the coding of a ground software file used in the trajectory models as the root cause. These trajectory models produced data ultimately used to define the vehicle trajectory for the flight computer. Thruster performance data were in English units instead of metric. As a result, an erroneous trajectory was calculated which led to the vehicle crashing into the surface rather than entering into an orbit around Mars. Formal acceptance testing failed to capture the problem because the test article used for comparison contained the same error as the output file from the actual unit.

Incomplete specifications were a contributing factor. The specifications did not dictate the units to be used. Also, a lack of warning marks in the original code, identifying the potential problem, contributed to the failure. The MCO investigators also cited inadequacies in risk identification, communication, management, and mitigation that compromised mission success. In part, these inadequacies resulted from cost and schedule pressures.

Lesson Learned: Multiple factors can lead to failure, including inadequate testing, incomplete specifications, and inadequate risk management (Leveson 2004, Stephenson 1999).

Mars Polar Lander. The Mars Polar Lander (MPL) was launched on January 3, 1999. Upon arrival at Mars, communications ended according to plan as the vehicle prepared to enter the Martian atmosphere. Communications were scheduled to resume after the Lander and the probes were on the surface. However, repeated efforts to contact the vehicle failed. The cause of the MPL loss was never fully identified, but the most likely scenario was that the problem involved deployment of the three landing legs during the landing sequence. Each leg was fitted with a Hall Effect magnetic sensor that generated a voltage when the leg contacted the surface of Mars. A command from the flight software was to shutdown by the descent engines when touchdown was detected. The MPL investigators believed that the software interpreted spurious signals generated at leg deployment as valid touchdown events, leading to premature shutdown of the engines at 40 meters above the surface of Mars, resulting in the vehicle crashing into the surface.

Although a possible failure mode whereby the sensors would falsely detect that the vehicle had touched down was known to exist, the software requirements did not account for this failure mode. Therefore, the software was not programmed to avoid such an occurrence. Although the verification and validation program was well planned and executed, the MPL failure report noted, analysis was often substituted for testing to save costs. Such analysis may have lacked adequate fidelity. Also, the touchdown sensing software was not tested with the Lander in the flight configuration. The MPL investigators specifically recommended that system software testing include stress testing and fault injection in a suitable simulation environment to determine the limits of capability and search for hidden flaws.

Lesson Learned: Multiple factors can lead to failure, including insufficient system engineering efforts, insufficient testing, flawed review processes, and flawed requirements (JPL 2000, Leveson 2004).

Titan/Centaur-Milstar. On April 30,1999, a Titan IV B vehicle (Titan IV B-32), with a Titan Centaur upper stage (TC-14) was launched from Space Launch Complex 40 at Cape Canaveral Air Station, Florida. The mission was to place a Milstar satellite into geosynchronous orbit. The flight performance of the Titan solid rocket motors and core vehicle was nominal, and the Centaur upper stage separated properly from the Titan IV B. The vehicle began experiencing instability about the roll axis during the first Centaur burn. That instability was greatly magnified during Centaur's second main engine burn, resulting in uncontrolled vehicle tumbling. The Centaur tried to compensate for those attitude errors by using its Reaction Control System. Such attempts ultimately depleted available propellant during the transfer orbit coast phase. The third engine burn ended early because of the tumbling vehicle motion. As a result of the anomalous events, the Milstar satellite was placed in a low elliptical final orbit instead of the intended geosynchronous orbit.

The Accident Investigation Board concluded that a failed software development, testing, and quality assurance process for the Centaur upper stage caused the failure of the Titan IV B-32 mission. That failed engineering process did not detect nor did it correct a human error in the manual entry of the roll rate filter constant entered in the Inertial Measurement System flight software file. Evidence of the incorrect constant appeared during launch processing and the launch countdown, but its impact was insufficiently recognized or understood. Consequently, this error was not corrected before launch. The incorrect roll rate filter constant zeroed any roll rate data, resulting in the loss of control. The Board noted that the manually input values were never formally tested in any of the simulations before launch, and simulator testing was not performed as the system was supposed to be flown.

Lesson Learned: Flawed engineering processes, underestimation of the software risks, and inadequate software reviews can lead to failure (Leveson 2004, Pavlovich 1999).

Sea Launch/ICO F1. On March 12, 2000, a Sea Launch lifted off from the Odyssey launch platform positioned on the Equator in the Pacific Ocean. The vehicle was carrying the ICO Global Communications F-1 satellite. Shortly before the launch, however, the ground software failed to close a valve in the pneumatic system of the second stage of the rocket. This system performed several actions, including operation and movement for the steering engine of this stage. Loss of more than 60 percent of the pneumatic system's pressure reduced the capability of the engine; therefore, the rocket did not gain the altitude and speed necessary to achieve orbit. About 8 minutes into the flight, as the Zenit's second stage was nearing the completion of its firing, the launch was aborted under command of the on-board automatic flight termination system. The rocket issued the command once it sensed a deviation in attitude. Both the rocket and its satellite cargo crashed into the Pacific Ocean about 2700 miles southeast of the launch site.

The software error was traced back to a change of a variable name in the ground operations software. This name change resulted in a change to the software logic such that the valve failed to close before launch. Ultimately, Sea Launch discovered flaws in their configuration management and software engineering processes, including identifying changes in the system and verifying proper operation after those changes.

Lesson Learned: Flawed configuration management and software engineering processes can lead to failure (AW&ST 2000, Ray 2000).

Cosmos/Quickbird 1. On November 28, 2000, the QuickBird 1 satellite was launched aboard a Russian Cosmos-3M rocket from the Plesetsk Cosmodrome. However, ground stations did not detect signals from the satellite after launch. Investigators suggested that a computer error inside the satellite might have caused the U.S.-built spacecraft to deploy its electricity-generating solar arrays while the rocket was still climbing through the atmosphere. The computer error may have resulted from a hold in the launch, which was delayed 1-hour because a Norwegian tracking station was not ready to monitor the satellite.

Russian officials proposed that an operator forgot to reset the satellite computer to account for the new launch time. As a result, the flight command sequence of the spacecraft began at the original launch time and, following its preprogrammed time line, tried to deploy the solar panels while the satellite was still attached to the rocket during the early phase of the flight.

Lesson Learned: Failure to understand software risks can lead to mission failure (Clark 2000).

Mar Rover Spirit. NASA's Mars Exploration Rovers, Spirit and Opportunity, landed on Mars on January 4 and 25, 2004. On January 21, 2004, Spirit abruptly ceased communications with mission control. When contact was re-established, mission control found that Spirit could not complete any task that requested memory from the flight computer. Examination of the problem showed that the file system was consuming too much memory, causing the computer to reset repeatedly. The root cause of the failure was traced to incorrect configuration parameters in two operating system software modules that controlled the storage of files in memory. Effects of overburdened memory were not recognized or tested during ground tests.

Mission operations personnel recovered Spirit by manually reallocating system memory, deleting unnecessary files and directories, and commanding the computer to create a new file system. Although the rover was recovered, the malfunction took 14 days to diagnose and fix, thereby reducing the nominal mission duration.

A post-anomaly review showed that memory management risks were not understood. In addition, schedule pressures prevented extensive testing and understanding of software functions.

Lesson Learned: Memory management strategies are important for software assurance (Reeves and Neilson 2005).

CryoSat. On October 8, 2005, a Russian-built Rockot launch vehicle, carrying the CryoSat satellite, blasted off from Russia's northwestern Plesetsk Cosmodrome. Analysis of the telemetry data indicated that the first stage performed nominally. The second stage performed nominally until main engine cut-off was to occur. However, the second stage main engine failed to shutdown at the proper time, and continued to operate until depletion of the remaining fuel. As a consequence, the second stage was not separated from the third stage, and the third stage engine was not ignited. This lack of engine capability resulted in unstable flight, causing the vehicle flight angles to exceed allowable limits. The on-board computer automatically ended the mission at 308 seconds into flight. For the second stage shutdown to succeed, pressurization of the low-pressure tank of the third stage had to have been completed before issuance of the shutdown command.

Failure analysis showed that the command to shutdown the second stage engine was generated correctly. However, the completion time for the pressurization sequence was erroneously specified; therefore, pressurization completed after the shutdown command was generated. This failure case had not been identified in development and was not tested. No built-in tests existed for the pressurization time.

Lesson Learned: Adequate consideration must be given to off-nominal inputs and conditions during design and verification (Briggs 2005, Eurocket 2005).

C.2 Commercial, Military, and Experimental Aircraft Accidents

The space vehicle failures described here discuss incidents resulting in mission failures or anomalies without impacts to the uninvolved public. However, such incidents illustrate the importance of software and computing systems in the operation of space and launch vehicles. Unfortunately, software has been a cause of several accidents resulting in injury and loss of life in commercial and military aircraft. Some of those accidents are described next.

Lufthansa A320. On Sept. 14, 1993, a Lufthansa Airbus A320-200 was landing in bad weather at the airport in Warsaw, Poland. The pilots had been warned of gusting crosswinds, rain, and possible wind shear conditions. To compensate for the bad weather problems, the crew added 20 knots of speed to the landing approach and used a standard crosswind landing technique, keeping the right wing low and landing first on the right gear. However, because of the gusting winds and heavy rains, the wheels aquaplaned during the first 9 seconds on the ground. The extra wind and water combined to fool the Airbus computer, indicating the jet had not landed. The computer responded by disabling the aircraft braking systems. With no brakes, the Lufthansa jet skidded off the end of the runway and struck a hill, killing the first officer and one passenger and injuring 45 others. The A320 airplane was totally destroyed in the crash.

Lesson Learned: Misunderstanding of the software risks and flawed software requirements can lead to an accident (Ladkin 1996).

Chinook Helicopter. In June 1994, a British Royal Air Force Chinook helicopter ZD576 crashed on the Mull of Kintyre, Scotland, killing 29 people. Although pilot error was originally suggested as the cause, further investigation revealed that the Full Authority Digital Electronic Control (FADEC) software triggered the accident. The FADEC software maintains a correct balance between the two engines for the flow of fuel; therefore, it is critical for the power output. In test flights, the helicopter would increase power to one or more engines for no apparent reason. Unexplained illumination of warning lights in the helicopter cockpit was another commonly reported fault. Both problems appeared to result from faulty FADEC software. These problems could have been factors in the Mull of Kintyre crash.

Lesson Learned: Understanding and mitigating software risks are critical aspects of software safety (BBC News 1999).

X-31. An X-31 U.S. government research aircraft was destroyed when it crashed in an unpopulated area just north of Edwards Air Force Base while on a flight originating from the NASA Dryden Flight Research Center, Edwards, California, on January 19, 1995. The crash occurred when the aircraft was returning after completing the third research mission of the day. The pilot safely ejected from the aircraft but suffered serious injuries, including two fractured vertebrae and a broken ankle and rib.

A mishap investigation board studying the cause of the X-31 accident concluded that an accumulation of ice in or on the unheated pitot-static system of the aircraft provided false airspeed information to the flight control computers. The resulting false reading of total air pressure data caused the flight control system of the aircraft to automatically misconfigure for a lower speed. The aircraft suddenly began oscillating in all axes, pitched up to over 90 degrees angle of attack and became uncontrollable, prompting the pilot to eject.

The mishap investigation board also faulted the safety analyses, performed by Rockwell and repeated by NASA, which underestimated the severity of the effect of large errors in the pitot-static system. Rockwell and NASA had assumed that the flight software would use the backup flight control mode if this problem occurred.

Lesson Learned: Estimating and mitigating software risks, including software used to mitigate hardware anomalies, are critical aspects of software safety (Dornheim 1995, Haley 1995).

V-22 Osprey. On December 11, 2000, four Marines were killed in the crash of a V-22 aircraft near Camp Lejeune, North Carolina. According to published reports, the left-hand nacelle titanium hydraulic line developed a leak during flight. As a result, the hydraulic line reservoir was depleted, and the V-22 experienced a total loss of the hydraulic system. The pilots responded to the master alert and primary flight control system annunciation by depressing the flight control alert and resetting the flight controls. When the primary flight control reset button was pressed in accordance with established procedures, a software anomaly caused significant pitch and thrust changes in both prop rotors. These fluctuations resulted in decreased airspeed; reduced altitude; and incorrect pitch, roll, and yaw motions that eventually accompanied increasing rates of descent and angle of attack. Essentially, the vehicle swerved out of control, stalled, then crashed. The crew had pressed the reset button to reset the primary flight control logic 8 to 10 times in an attempt to reset the system and regain control during the emergency. However, each primary flight control system reset aggravated the situation until the aircraft entered a stall condition. The software anomaly was found in a previously overlooked path in the flight control laws associated with the propeller rotor governor. This anomaly allowed large torque and RPM changes to be introduced when multiple failure conditions existed.

Lesson Learned: Estimating and mitigating software risks are critical aspects of software safety (Murray 2001).

REFERENCES

"12 Satellites Go Down in Russia." 1998. *Wired News*, 10 September. http://www.wired.com/news/politics/0,1283,14929,00.html (accessed May 19, 2006).

American Institute of Aeronautics and Astronautics. 2005. *Guide to the Identification of Safety-Critical Hardware Items for Reusable Launch Vehicle (RLV) Developers (May 1, 2005)*. Reston, Virginia. http://www.aiaa.com/pdf/industry/rlvguide.pdf (accessed May 23, 2006).

Benediktsson, O., R. B. Hunter, and A. D. McGettick. 2001. "Processes for Software in Safety Critical Systems." *Software Process: Improvement and Practice*, vol. 6, no. 1, pp. 47-62.

Briggs, Helen. 2005. "Cryosat Rocket Fault Laid Bare." *BBC News*, 27 October. http://news.bbc.co.uk/1/hi/sci/tech/4381840.stm (accessed May 19, 2006).

Chan, Steven. 2001. *System Safety Lessons Learned Handbook*, U.S. Army Communications–Electronics Command, CECOM-TR-01-4. http://www.armymars.net/ArmyMARS/Safety/Resources/system-safety-lessons-learned.pdf (accessed May 24, 2006).

Chapman, R. Jack and Paul A. Regeon. 1995. "The Clementine Lunar Orbiter Project." Unpublished paper presented at the Austrian Space Agency Summer School. 26 July– 3 August. Alpbach, Germany.

Clark, Stephen. 2002. "Commercial Eye-In-The-Sky Appears Lost in Launch Failure." *Spaceflight Now*, 21 November. http://spaceflightnow.com/news/n0011/20quickbird/ (accessed May 19, 2006).

Covault, Craig. 1998. "Boeing Delta III Explodes; Commercial Debut Ruined." *Aviation Week & Space Technology*, 31 August, vol. 23.

Czerny, Barbara J., et al. 2005. "Effective Application of Software Safety Techniques for Automotive Embedded Control Systems." Paper presented at the Society of Automotive Engineers World Congress, 11-14 April at Detroit, Michigan. SAE Technical Series Paper 2005-01-0785.

Dehlinger, Josh and Robyn R. Lutz. 2004 . "Software Fault Tree Analysis for Product Lines." Paper presented at the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE '04), 24-26 March at Tampa, Florida.

Department of Defense. 2000. *Standard Practice for System Safety*. MIL-STD-882D.

Dornheim, Michael A. 1995. "X-31 Board Cites Safety Analyses, But Not All Agree." *Aviation Week & Space Technology*, 4 December, pp. 81-86.

Dunn, William R. 2002. *Practical Design of Safety-Critical Computer Systems,* Solvang, California: Reliability Press.

EUROCKOT Launch Services GmbH. 2005. "CryoSat Failure Analyzed – KOMPSAT-2 Launch in Spring 2006." Eurocket Press Release. 21 December, Bermen, Germany. http://www.eurockot.com/alist.asp?cnt=20040862 (accessed May 19, 2006).

Federal Aviation Administration (FAA). 1999. *Equipment, Systems, and Installations in Part 23 Airplanes*, Washington, D.C. Advisory Circular 23.1309-1A.

-----. 2000. *FAA System Safety Handbook*. Washington, D.C. http://www.asy.faa.gov/Risk/SSHandbook/cover.htm (accessed May 19, 2006).

-----. Associate Administrator for Commercial Space Transportation. 2003. *Guide to Reusable Launch Vehicle Safety Validation & Verification Planning*. Version 1.0. Washington, D.C.

-----. Associate Administrator for Commercial Space Transportation. 2005. *Guide to Reusable Launch and Reentry Vehicle Reliability Analysis*. Version 1.0. Washington, D.C.

-----. 2005. *Reusable Launch and Reentry Vehicle System Safety Process*. Washington, D.C. Advisory Circular 431.35-2A.

Feng, Qian and Robyn R Lutz. 2005. "Bi-Directional Safety Analysis of Product Lines." *J. Systems and Software*, vol. 78, no. 2. pp. 111-127.

Ganssle, Jack G. 2000. "Crash and Burn." *Embedded Systems Programming,* http://www.embedded.com/2000/0011/0011br.htm (accessed May 30, 2006).

Gowen, Lon D. 1996. "Using Fault Trees and Event Trees as Oracles for Testing Safety-Critical Software Systems," *Professional Safety*. American Society of Safety Engineers. April, pp. 41-44.

Gowen, Lon D. and James S. Collofello. 1995. "Design-Phase Considerations for Safety-Critical Software Systems." *Professional Safety*. American Society of Safety Engineers, April, pp. 20-25.

Greenwell, William S. and John C. Knight. 2003. SAFECOMP 2003: "What Should Aviation Safety Incidents Teach Us?" Paper presented at the 22nd International Conference on Computer Safety, Reliability and Security, September at Edinburgh, Scotland.

Haley, Don. 1995. "Ice Cause of X-31 Crash." NASA Dryden Flight Research Center, Edwards, California. NASA Press Release 95-203.

Harland, David M. and Ralph D. Lorenz. 2005. *Space System Failures: Disasters and Rescues of Satellites, Rockets, and Space Probes.* Berlin: Praxis Publishing Ltd.

Holloway, C. Michael. 1999. "From Lessons to Bridges, Lessons for Software Systems." *Proceedings of the 17th International System Safety Conference*, 16-21 August at Orlando, Florida. pp. 598-607.

Herrmann, Debra S. 2000. *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*. Los Alamitos, California: Wiley-IEEE Computer Society Press.

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

-----. 1993. IEEE Standard Classification for Software Anomalies. New York. IEEE STD 1044-1993.

-----. 1998. *IEEE Standard for Software Verification and Validation.* New York. IEEE STD 1012-1998.

-----. 2002. *IEEE Standard for Software Safety Plans,* 1994. New York. IEEE STD 1228-1994 (R2002).

Joint Services Software Safety Committee (JSSSC). 1999. *Software System Safety Handbook: A Technical & Management Team Approach.*

NASA, Jet Propulsion Laboratory. 2000. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions.* JPL D-18709

Kaner, C., J. Faulk, and H. Q. Nguyen. 1999. *Testing Computer Software.* 2nd ed., Wiley and Sons.

Kit, Edward. 1995. *Software Testing in the Real World: Improving the Process.* Boston: Addison-Wesley.

Ladkin, P. 1996. *Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993*. Main Commission Aircraft Accident Investigation Warsaw.

Leveson, Nancy G. 1986. "Software Safety: Why, What, and How." *Computing Surveys*, vol. 18, no. 2, pp.125-163.

-----. 1995. *Safeware: System Safety and Computers, A Guide to Preventing Accidents and Losses Caused by Technology.* Boston: Addison-Wesley.

-----. 2001. "The Role of Software in Recent Aerospace Accidents." Paper presented at the 19th International System Safety Conference, 10-14 September at Huntsville, Alabama.

-----. 2004. "The Role of Software in Spacecraft Accidents." *AIAA Journal of Spacecraft and Rockets*, vol. 41, no. 4, pp. 564-575.

Lions, J. L. 1996. *Ariane5: Flight 501Failure Report by the Inquiry Board*. Paris: European Space Agency.

"Magazine Disputes Chinook Tragedy Cause." 1999. *BBC News Online*, 26 May. http://news.bbc.co.uk/1/hi/uk/353063.stm (accessed May 19, 2006).

American National Standards Institute (ANSI), Inc., and U.S. Department of Defense. 1983. *Military Standard (MIL-STD): Ada Programming Language*. ANSI/MIL-STD-1815A.

Murray, Bill. 2001. "Corps Cites Software Failure in Osprey Crash." *Federal Computer Week*, 9 April. http://www.fcw.com/fcw/articles/2001/0409/news-osprey-04-09-01.asp (accessed May 19, 2006).

NASA. 1990. *NASA Software Quality Assurance Audits Guidebook.* NASA-GB-A301.

----- . 1993. NASA Software Formal Inspections Guidebook. NASA-GB-A302.

----- . 1993. NASA Software Formal Inspections Standard. NASA-STD-2202-93 (Revalidated March 29, 2001).

----- . 1997. NASA Software Assurance Guidebook. NASA-GB-A201.

----- . 2004. NASA Software Assurance Standard. NASA-STD-8739.8 w/Change 1.

----- . 2004. NASA Software Safety Guidebook. NASA-GB-8719.13.

----- . 2004. NASA Software Safety Standard. NASA-STD-8719.13B w/Change 1.

Norman, Don A. 1990. "Commentary: Human Error and the Design of Computer Systems." *Communications of the ACM*, vol. 33, pp. 4-7.

O'Halloran, Colin, et al. 2005. "Ariane 5: Learning from Failure." *Proceedings of the 23rd International System Safety Conference*, August at San Diego, California.

Ozarin, Nathaniel. "Planning and Performing Failure Mode and Effects Analysis on Software." 2006. Paper presented at the 52nd Annual Reliability and Maintainability Symposium, 23-26 January at Newport Beach, California.

Pavlovich, J. Gregory. 1999. *Formal Report of Investigation of the 30 April 1999 Titan IVB/Centaur TC-14/Milstar-3 (B-32) Space Launch Mishap*. Washington, D.C.: U.S. Air Force.

Perminov, V. G. 1999. *The Difficult Road to Mars: A Brief History of Mars Exploration in the Soviet Union*. NASA Monographs in Aerospace History, no. 15. NP-1999-06-251-HQ.

Ray, Justin. 2000. "Sea Launch Malfunction Blamed on Software Glitch." *Spaceflight Now*, 30 March. http://spaceflightnow.com/sealaunch/ico1/000330software.html (accessed May 22, 2006).

Reeves, Glenn and Tracy Neilson. 2005. "The Mars Rover Spirit FLASH Anomaly." Paper presented at the IEEE Aerospace Conference, March at Big Sky, Montana.

RTCA, Inc. 1992. Software Considerations in Airborne Systems and Equipment Certification. Washington, D.C. RTCA/DO-178B.

"Sea Launch Poised to Fly with PAS-9." 2000. *Aviation Week & Space Technology*, 3 July. http://www.aeronautics.ru/nws002/awst050.htm (accessed May 22, 2006).

Stephenson, Arthur, et al. 1999. *Mars Climate Orbiter: Mishap Investigation Board Phase I Report*, 10 November, Washington, D.C.: NASA.

Storey, Neil. 1996. *Safety Critical Computer Systems*, Addison-Wesley Longman.

System Safety Society. 1997. *System Safety Analysis Handbook*. 2nd ed., Unionville, Virginia.

The Boeing Company. 1998. "Boeing Pinpoints Cause of Delta 3 Failure, Predicts Timely Return to Flight." Boeing Press Release, http://www.boeing.com/defense-space/space/delta/delta3/d3results.htm (accessed May 19, 2006).

U.S. Air Force Space Command. 2004. *Range Safety User Requirements Manual: Air Force Space Command Range Safety Policies and Procedures*. Air Force Space Command Manual 91-710, vol. 1. Peterson Air Force Base, Colorado.

Wood, Bill J. 1999. "Software Risk Management for Medical Devices." *Medical Device & Industry Magazine*, January. http://www.devicelink.com/mddi/archive/99/01/013.html (accessed May 22, 2006).

Woronowycz, Roman. 1998. "Crash of Ukrainian Rocket Imperils Space Program." *The Ukrainian Weekly*, 20 September, vol. 66, no. 38.