U.S. Department
of Transportation
Federal Aviation Administration

Advisory Circular

Subject: Computing System Safety

Date: 10/15/2020
Initiated By: AST-1

AC No: 450.141-1

This Advisory Circular (AC) provides guidance for an applicant to identify computing system safety items, develop safety requirements for each computing system safety item, and mitigate the risks presented by computing system safety items in accordance with title 14 of the Code of Federal Regulations (14 CFR) § 450.141. An applicant must identify all computing system safety items and implement safety requirements for each computing system safety item based on level of criticality, in accordance with § 450.141(a) and (b). An applicant must then implement a development process appropriate for each computing system safety item's level of criticality, in accordance with § 450.141(c).

The FAA considers this AC an accepted means of compliance with the regulatory requirements of § 450.141. It presents one, but not the only, acceptable means of compliance with the associated regulatory requirements. The FAA will consider other means of compliance that an applicant may elect to present. The contents of this document do not have the force and effect of law and are not meant to bind the public in any way. The document is intended only to provide clarity to the public regarding existing requirements under the law or agency policies.

If you have suggestions for improving this AC, you may use the Advisory Circular Feedback form at the end of this AC.

Wayne R. Monteith
Associate Administrator
Commercial Space Transportation

Contents

Figures

Contents (continued)

Paragraph **Page**

Tables

Number **Page**

1 PURPOSE.

1.1 This Advisory Circular (AC) provides guidance for identifying computing system safety items, developing safety requirements for each computing system safety item, and mitigating the risks presented by computing system safety items in accordance with 14 CFR § 450.141. It is intended to provide guidance for an applicant in developing software and computing system safety analyses and processes to comply with § 450.141.

1.2 This AC presents one, but not the only, acceptable means of compliance with the associated regulatory requirements. The FAA will consider other means of compliance that an applicant may elect to present. Other means of compliance may be acceptable, but must be approved by the FAA Administrator in accordance with § 450.35(a)(1).

2 APPLICABILITY.

2.1 The guidance in this AC is for launch and reentry vehicle applicants and operators required to comply with 14 CFR part 450. The guidance in this AC is for those seeking a launch or reentry vehicle operator license, a licensed operator seeking to renew or modify an existing vehicle operator license, and FAA commercial space transportation evaluators.

2.2 The material in this AC is advisory in nature and does not constitute a regulation. This guidance is not legally binding in its own right and will not be relied upon by the FAA as a separate basis for affirmative enforcement action or other administrative penalty. Conformity with this guidance document (as distinct from existing statutes and regulations) is voluntary only, and nonconformity will not affect rights and obligations under existing statutes and regulations. This AC describes acceptable means, but not the only means, for demonstrating compliance with the applicable regulations.

2.3 The material in this AC does not change or create any additional regulatory requirements, nor does it authorize changes to, or deviations from, existing regulatory requirements.

2.4 Throughout this document, the word “must” characterizes statements that directly flow from regulatory text and therefore reflect regulatory mandates. The word “should” describes a requirement if electing to use this means of compliance; variation from these requirements is possible, but must be justified and approved as an alternative means of compliance. The word “may” describes variations or alternatives allowed within the accepted means of compliance set forth in this AC. In general, these alternative approaches can be used only under certain situations that do not compromise safety.

3 APPLICABLE REGULATIONS AND RELATED GUIDANCE DOCUMENTS.

3.1 Applicable Statute.

- 51 U.S.C. Subtitle V, Chapter 509.

3.2 Applicable FAA Commercial Space Transportation Regulations.

The following 14 CFR regulations must be accounted for when showing compliance with § 450.141. The full text of these regulations can be downloaded from the U.S. Government Printing Office e-CFR. A paper copy can be ordered from the Government Printing Office, Superintendent of Documents, Attn: New Orders, PO Box 371954, Pittsburgh, PA, 15250-7954.

- Section 401.7, *Definitions*.
- Section 450.35, *Means of compliance*.
- Section 450.141, *Computing System Safety*.

3.3 Technical Reports Related to Computing System Safety.

1. Department of Defense, Standard Practice MIL-STD-882E, System Safety, May 11, 2012 (https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=36027).
2. Institute of Electrical and Electronic Engineers, ISO/IEC/IEEE 29119-3:2013, *Software and systems engineering – Software testing – Part 3: Test documentation*. https://www.iso.org/standard/56737.html.
3. Institute of Electrical and Electronic Engineers, IEEE 1012-2016/Cor 1-2017 - *IEEE Draft Standard for System, Software and Hardware Verification and Validation* - Corrigendum 1 https://standards-stg.ieee.org/standard/1012-2016-Cor1-2017.html.
4. Institute of Electrical and Electronic Engineers, IEEE 1228-1994, https://standards.ieee.org/standard/1228-1994.html.
5. Joint Services Software Safety Committee, *Software System Safety Handbook*, Version 1, dated August 27, 2010, http://www.acqnotes.com/Attachments/Joint-SW-Systems-Safety-Engineering-Handbook.pdf.
6. National Aeronautics and Space Administration, NASA-STD-8739.8, *Software Assurance and Software Safety Standard.* https://standards.nasa.gov/standard/osma/nasa-std-87398.
7. National Aeronautics and Space Administration, *Software Formal Inspections Standard* (NASA-STD-8739.9 https://standards.nasa.gov/standard/osma/nasa-std-87399.
8. National Aeronautics and Space Administration, NASA-GB-8719.13, *NASA Software Safety Guidebook*, dated March 31, 2004. https://standards.nasa.gov/standard/nasa/nasa-gb-871913

9. National Aeronautics and Space Administration, NASA-HDBK-2203, *NASA Software Engineering and Assurance Handbook*, dated April 4, 2020. https://standards.nasa.gov/standard/nasa/nasa-hdbk-2203.

10. Range Commanders Council, Range Safety Group, *Flight Termination System Commonality Standard*, RCC 319-19, White Sands, NM, 2019. https://www.wsmr.army.mil/RCCsite/Documents/319-19_FTS_Commonality/319-19_FTS_Commonality.pdf.

11. Society of Automotive Engineers, GEIA-STD-0010A, *Standard Best Practices for System Safety Program Development and Execution*, dated October 18, 2018, https://www.sae.org/standards/content/geiastd0010a/?src=geiastd0010.

4 DEFINITION OF TERMS.

For this AC, the terms and definitions from § 401.5, § 401.7, and this list apply:

4.1 Commercial off-the-shelf (COTS) software

Operating systems, libraries, applications, and other software purchased from a commercial vendor and not custom built for the applicant's project.

4.2 Computing system safety item

Any software or data that implements a capability that, by intended operation, unintended operation, or non-operation, can present a hazard to the public. A computing system safety item often contains several software functions assembled to meet a group of related requirements (e.g. an autonomous flight safety system (AFSS) or GPS)).

4.3 Degree of control

A computing system safety item's importance in the causal chain for a hazard, in either causing or preventing the hazard.

4.4 Failure

The inability of a computing system item to fulfill its operational requirements.

4.5 Failure Modes and Effects Analysis (FMEA)

An analysis of each potential failure in a system to determine the effects of each potential failure on the system and to classify each potential failure according to its severity and likelihood.

4.6 Fault

An imperfection or deficiency in a computing system item that may contribute to a failure.

4.7 Fault Tree Analysis (FTA)

An analysis that identifies potential faults in the system and determines how those faults lead to a failure of the system to achieve its purpose. An FTA can include deductive system reliability analysis that provides qualitative and quantitative measures of the propensity for failure of a system, subsystem, or event.

4.8 Firmware

Computer programs or data loaded into a class of memory that cannot be dynamically modified by the computer during processing. Firmware is treated as software.

4.9 Functional Hazard Analysis

Systematic, comprehensive examination of vehicle and system functions to identify potentially hazardous conditions that may arise as a result of a fault, malfunction, or failure.

4.10 Government off-the-shelf (GOTS) software

Operating systems, libraries, applications, and other software obtained from a government entity and not custom built for the applicant’s project.

4.11 Level of criticality

The risk posed by a computing system safety item, which is a combination of the severity of the hazards associated with the computing system safety item and the computing system safety item’s degree of control.

4.12 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.

4.13 Risk

Measure that takes into consideration the probability of occurrence and the consequence of a hazard to a population or installation. For software, risk takes into consideration the software’s contributions to hazard causation or mitigation and the consequence of a hazard to a population or installation.

4.14 Risk mitigation

Process of reducing either the likelihood or the severity of a risk.

4.15 Safety-critical computer system function

Any computer system function whose proper recognition, control, performance, or tolerance, is essential to ensuring public safety and the safety of property.

4.16 Safety requirement

A computing system requirement or software requirement defined for a computing system safety item that specifies an attribute or function that presents, prevents, or is otherwise involved in a hazard to the public.

4.17 **Software**

Computer programs, databases, and, possibly, associated documentation and data pertaining to the operation of a computer system. Operating system software that controls the basic functions of the computer system and application software that enables the computer to perform tasks are included, as are configuration files, databases, firmware, and supporting data structures.

4.18 **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 ensures that the right safety measure is implemented, and that the safety measure is well understood.

4.19 **Verification**

An evaluation to determine that safety measures derived from a system safety process are effective and have been properly implemented. Verification provides objective evidence that a safety measure reduces risk to acceptable levels.

5 MEANS OF COMPLIANCE.

The guidance in chapters 6 through 9 of this AC details methods that an operator may use to demonstrate compliance with each of the requirements in § 450.141.

5.1 **RCC 319-19 Tailoring.**

An applicant may choose to demonstrate compliance with § 450.141 by tailoring Range Commanders Council, Range Safety Group, *Flight Termination System Commonality Standard*, RCC 319-19. The FAA will work with applicants to tailor RCC 319-19 to comply with § 450.141. A tailored RCC 319-19 used as a means of compliance for § 450.141 must be submitted to the FAA for acceptance prior to being included in a license application.[1]

[1] RCC 319-19 is a flight termination system design standard. Applicants should carefully consider the requirements throughout RCC 319-19 prior to tailoring the requirements for systems other than flight termination systems.

6 IDENTIFICATION OF COMPUTING SYSTEM SAFETY ITEMS.

6.1 Identifying each Computing System Safety Item.

6.1.1 Section 450.141(a)(1) requires the applicant to identify any software or data that implements a capability that, by intended operation, unintended operation, or non-operation, can present a hazard to the public. In order to meet the requirement of § 450.141(a), an operator should define computing system safety items in a way that encompasses all of the software functions that work together in a given component. It is possible to isolate or partition a computing system safety item from non-safety functionality on a single hardware component such that the non-safety functions would not be part of the computing system safety item. RCC 319-19 Appendix A discusses partitioning in greater detail.[2] An applicant's list of computing system safety items should at least include software that performs common safety-related functions, such as

1. Software used to control or monitor safety-related systems.
2. Software that transmits safety data, including time-critical data and data about hazardous conditions.
3. Software used for fault detection in safety-related computer hardware or software.
4. Software that responds to the detection of a safety-related fault.
5. Software used in a flight safety system.
6. Processor-interrupt software associated with safety-related computer system functions.
7. Software that schedules the execution of safety related functions.
8. Software that computes safety-related data.
9. Software that accesses or manages safety-related data.
10. Software that displays safety data.
11. Software used for wind weighting.

6.1.2 Computing system safety items and their effects on public safety should be evident from the applicant's functional hazard analysis, performed in accordance with § 450.107(b). If a system or subsystem hazard analysis identifies any software or data as potential hazard sources or hazard controls, then the applicant should perform a software hazard analysis to assess the hazard and the degree of control of any computing system safety item over the hazard, in accordance with § 450.141(a). Software hazard analyses identify potential software faults and their effects on the computing system and the system as a whole, as well as mitigation measures that can be used to reduce the risk. Typical software hazard analyses include Software Failure

[2] If an operator isolates or partitions non-safety functionality from computing system safety items on a single hardware component, then the isolation or partition would become a safety requirement that must be verified by testing appropriate for the highest criticality function performed by the hardware component, in accordance with § 450.141(b).

Modes Effects Analysis (SFMEA) and Software Fault Tree Analysis (SFTA). Appendix B of this AC and the Joint Services Software Safety Committee (JSSSC) *Software System Safety Handbook* provides examples of SFMEA and SFTA. The analytical method and level of detail in the analysis should correspond to the complexity of the software and computing system, intricacy of the operations, and scope of the program. An applicant's software hazard analyses should consider a range of potential error conditions (see Table 1 of this AC). Appendix B of this AC provides an example of a classification scheme for software and computing system errors that an applicant can use to develop its hazard analysis.

Table 1 – Examples of 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 continues 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.

Error Condition	**Examples**
Command out of sequence	A module may be executed in the wrong sequence, or a system operator may interrupt a process leading to an out of sequence command.
Race condition/incorrect timing	Software processes may have timing dependencies, within or between processes, which cause unintended data alterations at random times. 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 or missing, 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.

6.2 **Assessment of Computing System Criticality**.

6.2.1 Section 450.141(a)(2) requires an applicant to identify the level of criticality of each computing system safety item, commensurate with its degree of control over hazards to the public and the severity of those hazards. A level of criticality is specified by the combination of the severity of hazards associated with a computing system safety item and the computing system safety item's degree of control over those hazards.

6.2.2 To satisfy § 450.141(a)(2), an applicant should first define the severities of hazard consequences of interest for its system, then assign degrees of control to computing system safety items using a framework that will enable the FAA to validate appropriate severity and control categorization. An applicant could assess degrees of control using any of the following methods:

1. Assumption of high criticality: An applicant could eliminate the need to characterize the criticality of each computing system safety item by subjecting all computing systems to the process controls needed at its highest criticality level.
2. RCC 319-19, Section A.3.1.1 defines software categories for flight safety systems.
3. MIL-STD-882E, Section 4.4 (or GEIA-STD-0010A, Section A.6) defines software control categories and a software criticality index.
4. NASA-GB-8719.13, Section 3.1.2 defines software control categories and a software risk index.
5. Fault tolerance: an applicant could also use the fault tolerance method described in this section to assess system level criticality.

6.2.3 <u>Consequence Definitions</u>.

To assess the criticality of computing system safety items, an applicant should first define the hazard consequences of interest for its system. Applicants should define hazard consequences such that computing system safety items fit in one or more consequence categories. An applicant should consider the FAA's definitions of the words "mishap," "anomaly," "casualty," "public," and "safety critical" in § 401.7 when composing its consequence definitions.

6.2.3.1 **Option 1: Public Safety Consequence Categories**.

Guidance document MIL-STD-882E and GEIA-STD-0010A define hazard consequence categories (see references [1] and [11] of paragraph 3.3 of this AC). Since the severity categories in these standards do not explicitly identify public safety consequences, an applicant for a Part 450 license should tailor the MIL-STD-882E or GEIA-STD-0010A severity tables for public safety. This may result in something similar to Table 2, Public Safety Severity Categories, with revisions in **<u>bold underline</u>**.

Table 2 – Public Safety Severity Categories

Description	**Severity Category**	**Hazard Consequence**
Catastrophic	**1**	Could result in one or more of the following: death **<u>or</u>** permanent total disability **<u>of a member of the public</u>**, or irreversible significant environmental impact.
Critical	**2**	Could result in one or more of the following: permanent partial disability, injuries, or illness that may result in hospitalization of **<u>a member of the public</u>**, or reversible significant environmental impact.
Marginal	**3**	Could result in one or more of the following: injury or occupational illness resulting in one or more lost work day(s) **<u>to a member of the public</u>**, or reversible moderate environmental impact.
Negligible	**4**	Could result in one or more of the following: injury or occupational illness not resulting in a lost work day, or minimal environmental impact.

6.2.3.2 **Option 2: Functional Hazard Assessment Hazard Consequence Classifications**.

Computing system hazard consequences could also be determined in the course of a functional hazard assessment. In a Functional Hazard Assessment, such as that required by § 450.107(b) or § 450.109, the applicant needs to assess the effects on the public for all reasonably foreseeable hazardous events, and computing system safety assessments can use the consequence classifications from the functional hazard assessment to classify computing system hazard severity. The Advisory Circulars associated with § 450.107(b) and § 450.109 provide guidance on functional hazard assessment consequence classifications.

6.2.4 Method of Assigning Levels of Criticality.

The following are methods that an applicant could use to assess the degree of control of a computing system safety item has over a hazard to assign system level criticality.

6.2.4.1 **Option 1: Assumption of High Criticality**.

An applicant could determine analytically the highest criticality computing system safety item involved in its operations, and then define a development process that develops and tests all computing system safety items in a manner sufficient to mitigate the risks presented by its highest criticality computing system safety item. By subjecting all software to the process controls needed at its highest criticality level, an applicant could eliminate the need to characterize the criticality and degree of control of each computing system safety item, and FAA would accept that approach. Applicants could find this approach overly conservative.

6.2.4.2 **Option 2: RCC 319-19 Software Categories**.

An applicant could identify each of its computing system safety items as safety-critical, support-critical, or non-critical, according to the definitions in RCC 319-19, Section A.3.1.1. This method for identification of level of criticality and degree of control is appropriate for systems where an FTS is the only part of the system that contains computing system safety items because RCC 319-19 is an FTS standard.

6.2.4.3 **Option 3: MIL-STD-882E/GEIA-STD-0010A Software Control Categories**.

An applicant could identify each of its computing system safety items in the appropriate software control category from MIL-STD-882E, Section 4.4 (or GEIA-STD-0010A, Section A.6) (see references [1] and [11] of paragraph 3.3 of this AC).

Table 3 – MIL-STD-882E/GEIA-STD-0010A Software Control Categories

Level	**Name**	**Description**
1	Autonomous (AT)	Software functionality that exercises autonomous control authority over potentially safety-significant hardware systems, subsystems, or components without the possibility of predetermined safe detection and intervention by a control entity to preclude the occurrence of a mishap or hazard. *(This definition includes complex system/software functionality with multiple subsystems, interacting parallel processors, multiple interfaces, and safety-critical functions that are time critical.)*
2	Semi-Autonomous (SAT)	Software functionality that exercises control authority over potentially safety-significant hardware systems, subsystems, or components, allowing time for predetermined safe detection and intervention by independent safety mechanisms to mitigate or control the mishap or hazard. *(This definition includes the control of moderately complex system/software functionality, no parallel processing, or few interfaces, but other safety systems/mechanisms can partially mitigate. System and software fault detection and annunciation notify the control entity of the need for required safety actions.)* Computing system safety item that displays safety-significant information requiring immediate operator entity to execute a predetermined action for mitigation or control over a mishap or hazard. Software exception, failure, fault, or delay will allow, or fail to prevent, mishap occurrence. *(This definition assumes that the safety-critical display information may be time-critical, but the time available does not exceed the time required for adequate control entity response and hazard control.)*
3	Redundant Fault Tolerant (RFT)	Software functionality that issues commands over safety-significant hardware systems, subsystems, or components requiring a control entity to complete the command function. The system detection and functional reaction includes redundant, independent fault tolerant mechanisms for each defined hazardous condition. *(This definition assumes that there is adequate fault detection, annunciation, tolerance, and system recovery to prevent the hazard*

		occurrence if software fails, malfunctions, or degrades. There are redundant sources of safety-significant information, and mitigating functionality can respond within any time-critical period.) Software that generates information of a safety-critical nature used to make critical decisions. The system includes several redundant, independent fault tolerant mechanisms for each hazardous condition, detection, and display.
4	Influential	Software generates information of a safety-related nature used to make decisions by the operator, but does not require operator action to avoid a mishap.
5	No Safety Impact (NSI)	Software functionality that does not possess command or control authority over safety-significant hardware systems, subsystems, or components that does not provide safety-significant information. Software does not provide safety-significant or time sensitive data or information that requires control entity interaction. Software does not transport or resolve communication of safety-significant or time sensitive data.

6.2.4.4 **Option 4: NASA-GB-8719.13 Software Control Categories**.

An applicant could categorize each of its computing system safety items using the software control categories from NASA-GB-8719.13, Section 3.1.2.1 (see reference [8] of paragraph 3.3 of this AC).

Table 4 – NASA-GB-8719.13 Software Control Categories

Software Control Categories	**Descriptions**
IA	Partial or total autonomous control of safety-critical functions by software.
	Complex system with multiple subsystems, interacting parallel processors, or multiple interfaces.
	Some or all safety-critical functions are time critical.
IIA & IIB*	Control of hazard but other safety systems can partially mitigate.
	Detects hazards, notifies human operator of need for safety actions.
	Moderately complex with few subsystems and/or a few interfaces, no parallel processing.
	Some hazard control actions may be time critical but do not exceed time needed for adequate human operator or automated system response.
IIIA & III B* * A = software control of hazard. B = Software generates safety data for human operator	Several mitigating systems prevent hazard if software malfunctions.
	Redundant sources of safety-critical information.
	Somewhat complex system, limited number of interfaces.
	Mitigating systems can respond within any time critical period.
IV	No control over hazardous hardware.
	No safety-critical data generated for a human operator.
	Simple system with only 2-3 subsystems, limited number of interfaces.
	Not time-critical.

6.2.4.5 **Option 5: Fault Tolerance**.

An applicant could perform a detailed analysis to specify the system's tolerance to faults in computing system safety items. The system's tolerance to software faults should be the basis for degree of control assessments at the following levels, with additional levels as appropriate:

1. Zero fault tolerant – a single fault in the computing system safety item could result in an adverse consequence.
2. Single fault tolerant – a fault in the computing system safety item requires a second, independent fault in order to result in an adverse consequence.
3. Dual fault tolerant – a fault in the computing system safety item requires two or more independent faults in order to result in an adverse consequence.
4. Critical informational – a fault in the computing system safety item results in erroneous information that the operator could not readily perceive as erroneous, which can cause an operator to take actions that result in an adverse consequence.
5. Informational – a fault in the computing system safety item results in evidently erroneous information that, if not detected, could cause an operator to take actions that result in an adverse consequence.
6. Non-safety – a fault in the computing system safety item has no potential to result in an adverse consequence.

Note: These categories are not arranged in a hierarchical order; instead, each category represents a degree of control that a computing system safety item could exert on the system through nominal or faulty operation. For example, a computing system safety item that can cause an adverse outcome with a single fault, and for which no other system could credibly prevent an adverse outcome as a result of that fault, is zero fault tolerant. A computing system safety item that can cause an adverse outcome with a single fault, but that requires a hardware component to fail or a human to make a mistake in order for the software fault to cause an adverse outcome, may be one fault tolerant if the software fault is independent of the second fault. A computing system safety item that could misinform a system operator in a way that the operator could independently and credibly detect and correct would be informational. A computing system safety item that could misinform a system operator in a way that the operator could not independently detect or correct would be critical informational.

7 SAFETY REQUIREMENTS.

7.1 **Identification of Safety Requirements**.

7.1.1 In accordance with § 450.141(b)(1), applicants are required to develop safety requirements for each computing system safety item identified under § 450.141(a). Safety requirements specify the implementation of public safety-related functions, capabilities, or attributes in a computing system safety item. Safety requirements are not necessarily obvious, which makes their methodical and explicit identification an important step in understanding a system. In general, an increase in either the severity of hazards, or the computing system's degree of control over those hazards, will increase the level of rigor applied to that computing system to protect the public. The requirements in § 450.141 are based on this relationship between a computing system's potential consequences, degree of control in the causal chain for those consequences, and the rigor applied to development and testing. Software requirements and safety requirements are frequently inherited or derived from system requirements. In addition to the examples listed in Appendix A of this AC, common safety requirements for computing system safety items might include:

- Shall use metric units.
- Shall compute vacuum instantaneous impact point.
- Shall contain a boundary polygon that ensures that the consequence of any reasonably foreseeable failure mode, in any significant period of flight, is no greater than 1×10^{-3} conditional expected casualties.
- Shall compare vacuum instantaneous impact point to boundary polygon.
- Shall not use flight safety system memory or CPU.
- Shall accept and execute "abort" command from ground control at any time.

7.1.2 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, some of which are described in Appendix C. Faults in requirements can originate from the adoption of requirements that are incomplete, unnecessary, contradictory, unclear, unverifiable, untraceable, incorrect, in conflict with system performance requirements, otherwise poorly written, or undocumented. It is important that operators properly identify and document safety requirements, and per industry standards, ensure that safety requirements are internally consistent and valid at the system level for the resulting computing system to work safely. Applicants should implement a process for managing safety requirements throughout the lifecycle. The IEEE 1012-2016 with Cor. 1-2017 (see reference [3] of paragraph 3.3 of this AC) provides examples of approaches that can assist in managing requirements. The IEEE 1228-1994 (reference [4]) and NASA GB 8719.13 (reference [8]) also provide methods for managing and analyzing safety requirements.

7.2 **Ensuring Safety Requirements are Complete and Correct.**

Section 450.141(b)(2) requires that the applicant ensure that safety requirements are complete and correct. The applicant may use standards, such as NASA-STD-8739.9 (see reference [7] of paragraph 3.3 of this AC) or similar formal inspection standards for software, to meet this requirement.

7.2.1 Ensuring Safety Requirements are Complete.

A complete set of safety requirements for a given computing system safety item is the set of requirements that includes all functionality and attributes associated with public safety. To ensure complete safety requirements, an applicant should have a robust interface between system safety and software safety, robust software documentation, and a process to close any gaps in requirements identified by testing or operation of the system.

7.2.2 Ensuring Safety Requirements are Correct.

A set of safety requirements is correct if it specifies exactly and only the functions and attributes intended by the applicant, and when that intention aligns with the requirements for a safe and successful mission. To ensure correct safety requirements, an applicant should have processes for identifying and reviewing safety requirements at the system level and at the software level with coordination between levels, independent validation of safety requirements for safety-critical computing system safety items, and a process to close any gaps in requirements identified by testing or operation of the system. Reviews of safety requirements need not be single events, but can be accomplished progressively as individual computing system safety items mature.

7.3 **Implementation and Verification of Safety Requirements**.

Section 450.141(b)(3) requires the applicant to implement each safety requirement. There need not be a separate implementation process for safety requirements; the applicant’s normal process for implementing software requirements is sufficient. This step is required as a bridge between §§ 450.141(b)(2) and 450.141(b)(4), and records produced during this step are important for meeting § 450.141(d)(5).

7.3.1 Independent Verification and Validation.

In accordance with § 450.141(b)(4), applicants are required to verify and validate the implementation of each safety requirement by using a method appropriate for the level of criticality of the computing system safety item. This regulation requires that when testing a safety-critical computing system safety item, its verification and validation must include testing by a test team independent of the software development division or organization. This verification and validation should take place within the development cycle and contribute iterative findings to the design of the computing system safety item.

7.3.2 Choosing Appropriate Verification and Validation Methods.

Verification and validation methods should be proportional to the level of criticality of the computing system safety item, meaning the method produces a degree of certainty that leaves little risk of fault in the computing system safety item. One acceptable method of implementing verification and validation in proportion to criticality is provided by MIL-STD-882E (see reference [1] of paragraph 3.3 of this AC), in its discussion of “level of rigor tasks.” Acceptable methods of verification include analyses, formal inspections, and testing. In most cases, testing is the preferred verification approach. These methods are often used in combination, depending on the feasibility of the method and the maturity of the vehicle and operations.

8 DEVELOPMENT PROCESS.

8.1 **Development Process Rigor**.

Once the applicant has determined the safety requirements needed for each computing system safety item based on the level of criticality of each item, the applicant can assign the appropriate development tasks to its computing system safety items.
Section 450.141(c) requires an applicant to implement and document a development process for computing system safety items that is appropriate for the level of criticality of each computing system safety item. The FAA defines the performance objectives for software development tasks in § 450.141(c), but relies on the applicant to design the tasks that fit its development process while achieving the performance objectives. The tasks required to demonstrate the necessary rigor for a computing system safety item’s criticality should be assigned at the item level. The required tasks should be based on the highest criticality function of a computing system safety item and assigned through the software development process required by § 450.141(c).

8.2 **Development Process Requirements**.

8.2.1 Responsibility Assignments.

An applicant must define development responsibilities for each task associated with a computing system safety item, in accordance with § 450.141(c)(1). Applicants may achieve this requirement in a wide variety of ways, including documentation of the engineers responsible for computing system safety items, logging of approvals for changes to requirements and software, and definition of software build or release processes. An applicant has met this requirement when the applicant can determine who conducted and approved each step in the development of a computing system safety item retrospectively.

8.2.2 Review and Approval Processes.

An applicant must define processes for internal review and approval, including review that evaluates the implementation of all safety requirements, such that no person approves that person’s own work, in accordance with § 450.141(c)(2). This requirement is related to § 450.141(c)(1) because responsibility assignments need to be known in order to determine whether each review was conducted with the appropriate degree of independence. To meet this requirement, reviews and approvals should include reviews of safety requirements and approvals for their implementation, and reviews and approvals of verification and validation evidence at a minimum.

8.2.3 Training.

In accordance with § 450.141(c)(3), development personnel are required to be trained, qualified, and capable of performing their role. This training should include learning and practicing operations and procedures that protect the public, including operations and procedures for computing system safety item development. Training can be included as a risk mitigation measure in hazard analyses, as it limits the potential for a range of errors in computing system safety item development. The applicant should develop plans that describe its training process. This training should include, but is not limited to, training for development tools, development methods, installation and testing, hazard analysis approaches, computing system use, and software maintenance. Since training needs and methods are specific to each applicant, FAA's application evaluation will verify that an applicant's training process meets the performance objectives in this regulation. In its application review, the FAA does not intend to verify the qualifications of individual development personnel, but rather to verify that the operator has a process in place to put appropriately trained and experienced personnel in public safety roles.

8.2.4 Traceability.

An applicant must define processes that trace requirements to verification and validation evidence, in accordance with § 450.141(c)(4). This traceability enables the applicant to demonstrate that its verification and validation of each safety requirement is sufficient. An applicant should connect the computing system requirements to the analytical and test evidence that demonstrates their implementation in a manner suited to its development process. The connections should be verifiable and human-readable, and the connections for safety requirements should be included in the application materials. FAA does not prescribe the technical methods for making these traceability connections, but will evaluate the selected method for opportunities for error.

8.2.5 Configuration Management.

An applicant must define processes for configuration management that specify the content of each released version of a computing system safety item, in accordance with § 450.141(c)(5). The applicant must also comply with § 450.103(c) in its configuration management approach. This is the minimum performance requirement for configuration management, but an applicant may need to use other aspects of configuration management (e.g., hardware supply chain requirements or administrative procedures) to achieve the performance requirement. The development process should produce a record of each version of its resultant software or data and each hardware component on which each software or data component is installed. The applicant should retain a record of the system configuration for each computing system safety item in order to demonstrate compliance with the regulatory requirements. Changes to the computing system, especially on safety-critical systems, can have significant impacts on public safety. This configuration management and control process should be in force during the entire life cycle of the program, from initiation of development through 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. More information on configuration management can be found in paragraph

A.10.5 of Appendix A of this AC. As required by § 450.141(c)(5), the applicant must implement a process for configuration management that specifies the content of each released version of a computing system safety item. The process should, at a minimum:

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

8.2.6 Verification and Validation.

Safety analyses generate top- and design-level safety requirements that are used to meet the applicant's system safety goals. These requirements typically result from implementation of mitigation measures, or operational controls to reduce risk. Other sources of safety 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, and safety requirements verification and validation depends on the integrity of the set of safety requirements.

8.2.6.1 **Testing**.

In accordance with § 450.141(c)(6), applicants are required to define processes for testing that verify and validate all safety requirements to the extent required by § 450.141(b)(4). Testing should verify the correct functionality of a computing system safety item and validate its performance in the system. Tests should check the implementation of safety requirements. To demonstrate than an applicant sufficiently tested its computing systems based on their criticality, an applicant must trace its verification and validation evidence to requirements, as required by § 450.141(c)(4), and should describe the components of its testing process, such as the test plan, test cases, test logs, and procedures for testing computing systems in representative environments. Verification and validation testing in appropriately representative environments may include tests performed on flight-like hardware, Monte Carlo simulations, branch and boundary tests, mathematical validation, fault injection testing, or other methods appropriate for the criticality of the computing system safety item. The degree of testing required by § 450.141(c)(6) will depend on the complexity of the system—that is, the nature of the computing system safety items and their criticality, as identified in § 450.141(b)(4). Computing system testing is conducted as part of a larger system and vehicle verification program. The system's responses to computing system safety item faults should be tested whenever the system is part of the mitigation strategy for a fault or failure of a computing system safety item.

8.2.6.2 **Test Plan**.

A computing system testing process begins with a test plan that demonstrates how the results of testing will be used to meet all safety requirements. An applicant should develop its test plan before verification testing begins. A plan normally prescribes the scope, approach, resources, and schedule of the testing activities. The applicant’s plan should include a description of the test environments, including software tools and hardware test equipment. Tests may include, but are not limited to those contained in Table 6, Testing Types, of this AC.

Table 5 – Testing Types

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 throughout the planned mission duration.
Stress	Confirms the software will not cause hazards under abnormal circumstances, such as unexpected input values, overload conditions, or off-nominal mission timelines.
Regression	Demonstrates changes made to the software did not introduce conditions for new hazards.

8.2.6.3 **Test Cases**.

The applicant should also define specific test cases with pass and fail criteria. Test cases describe the inputs, predicted results, test conditions, and procedures for conducting the test. The applicant 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.

8.2.6.4 **Test Log**.

The applicant should record the results of the tests; this is often done in a test log. Anomalies discovered during testing should also be recorded. ISO/IEC/IEEE 29119-3:2013 (see reference [3] of paragraph 3.3 of this AC) provides additional information on test documentation. The recorded test results need to include the version of the software that was tested.

8.2.6.5 **Verification Tests**.

Testing traditionally has been relied on to verify that computing system requirements have been met and have been implemented correctly. There are several types of tests available, shown in Table 6, and verification testing is normally a combination of many or all types. The applicant should use a combination of verification approaches that are appropriate for its software system (analysis, inspection, and test), including testing to the extent practicable. The applicant should also use proven methods to verify the software requirements, which include, but are not limited, to the verification tests listed in Table 6 of this AC.

Table 6 – Verification 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 with a precision of 0.01, then test inputs could include 0, 100, -0.01, 100.01, 0.01, and 99.99. Test inputs that exceed the implicit boundaries of allocated memory, such as 129 or -128 for an 8-bit signed integer, should also be included.
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 instruction is executed at least once and instruction execution produces the expected response.
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 at least once and produced the expected results.

8.2.7 Previously Developed Software and Computing Systems.

As required by § 450.141(c)(7), an applicant must verify and validate the safety requirements for reused computing system safety items. In addition, an applicant must verify and validate the safety requirements for third-party products, as required by § 450.141(c)(8). Using previously developed computing system safety items 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 computing system safety items include commercial off-the-shelf (COTS) software, government off-the-shelf (GOTS) software, and "reused" software. Although another vendor may have developed the software or product, reducing the risks of using third-party products remains the responsibility of the applicant. These risk reduction efforts should include evaluating the differences between the computing system safety item's role in the new system and its use in the previous system, including assessment of any identified issues found during use in the previous system and implementation of all preconditions for its use in the new system. For third-party computing system safety items, risk reduction efforts should include verification of compliance with the developer's specified uses for third-party products and verification of safety requirements for its use in the system.

8.3 **Development Process Considerations**.

The rest of this chapter outlines considerations that may guide applicants in formulating development processes that satisfy § 450.141(c) and, more generally, the performance-based requirements of § 450.141.

8.3.1 Analysis.

Analyses to verify that the software requirements are implemented correctly could include the components described in Table 8 of this AC. Additional information about software analysis methods is available in IEEE 1228-1994 and the Joint Services Software Safety Committee (JSSSC) *Software System Safety Handbook* (see references [4] and [5] of paragraph 3.3 of this AC).

Table 7 – Software Implementation Analysis

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

8.3.2 Development Standards.

The applicant may identify development standards that define the rules and constraints for the development process in accordance with § 450.141(c). The use of development standards can enable uniformly designed and implemented computing system safety items and prevent the use of methods that are incompatible with safety requirements; referencing a standard may produce a compelling rationale for the acceptance of a development process. RCC 319-19 defines a development standard that is sufficient for a flight safety system, and an applicant can meet § 450.141(c) by demonstrating compliance with it (see reference [10] of paragraph 3.3 of this AC). Regardless of the standard or its use in the proposed operation, the application could establish an understanding of the parts of each standard that have been adopted by the applicant, how each part of each standard is used in the proposed development and operation, and how each part of each standard supports public safety.

Development standards include requirements, design, coding, and safety standards, as follows:

- Requirements standards may include methods for developing requirements and a description of how the requirements flow down to coding.
- Design standards may include restrictions on the use of scheduling and interrupts, specification of usable code libraries, or rules for conditional branches to reduce complexity.
- Coding standards may include specifications 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 or GEIA-STD-0010A (see references [1] and [11] of paragraph 3.3 of this AC).

8.3.3 Quality Assurance.

Quality assurance may support the achievement of the performance objectives in § 450.141. Quality assurance verifies that the objectives and requirements of the software system safety program are satisfied and confirms that deficiencies are detected, evaluated, tracked, and resolved. An acceptable quality assurance function could include audits and inspections of elements and processes, such as plans, standards, and problem tracking and configuration management systems. In addition, the quality assurance function could evaluate the validity of system safety data. *NASA Software Assurance and Software Safety Standard* (NASA-STD-8739.8), and *NASA Software Engineering and Assurance Handbook* (NASA-HDBK-2203) provide examples of acceptable software quality assurance methods (see references [6] and [9] of paragraph 3.3 of this AC).

8.3.4 Formal Inspections.

Formal inspections are well thought out technical reviews that provide a structured way to find and eliminate defects in documentation products, ranging from a requirements document to the actual source code. These inspections differ from informal reviews or walkthroughs since there are specified steps to be taken and roles assigned to individual reviewers. A process must be defined for internal review and approval such that no person approves the person's own work, as required by § 450.141(c)(2). Further information regarding formal inspections can be found in NASA's *Software Formal Inspections Standard*, NASA-STD-8739.9, (see reference [7] of paragraph 3.3 of this AC).

8.3.5 Anomaly Reports.

To help prevent recurrence of computing system safety-related anomalies, the applicant could develop a standardized process to document anomalies, analyze the root cause, and determine corrective actions. Software anomaly reports (also known as problem reports) are a means to identify and record:

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

8.3.6 Maintenance and Repair of Computing System Hardware.

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 applicant could perform activities to aid maintenance and repair of computing system hardware.

8.3.7 Maintenance of Computing System Software.

Software maintenance differs from hardware maintenance because software does not wear out or degrade in the same way that hardware does. However, software maintenance corrects defects; adds or removes features and capabilities; compensates for or adapts to hardware changes, wear-out, or failure; and accommodates changes in other computing system safety items or system components. Changes to both the hardware and software after deployment can produce computing system anomalies. An applicant could identify a process for verifying the integrity of the safety-critical software and computing systems 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.

8.3.8 Building Maintainable Software.

Because software changes can be expected over the lifecycle of the product, an applicant could build maintainable software to facilitate those changes and reduce the likelihood of introducing new hazards. NASA-GB-8719.13 (see reference [8] of paragraph 3.3 of this AC) provides additional information on software maintainability. The following are some considerations for building maintainable software:

- Planning early for expected changes.
- Using strong configuration management practices.
- 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.

9 APPLICATION MATERIALS.

This chapter provides guidance on satisfying the application requirements set forth in § 450.141(d). In accordance with § 450.141(d), an application must include:

- Descriptions of all computing system safety items involved in the proposed operations;
- All safety requirements for each computing system safety item;
- Documentation of the software development process that meets § 450.141(c);
- Evidence of the execution of appropriate development processes for each computing system safety item; and
- Evidence of the implementation of each safety requirement.

9.1 **Computing System Safety Items**.

An applicant must identify and describe all computing system safety items involved in its proposed operation, in accordance with § 450.141(d)(1). These descriptions should include the severity of hazards associated with each computing system safety item and the computing system safety item's level of criticality regarding each hazard. The descriptions of computing system safety items and their criticality allow the applicant to know, and the FAA to verify, how each computing system safety item influences public safety when combined with the other requirements in § 450.141.

9.2 **Safety Requirements**.

An applicant must identify the safety requirements for each computing system safety item, in accordance with § 450.141(d)(2). Although § 450.141(d)(2) only requires applicants to identify safety requirements, an applicant may, and is encouraged to, submit all computing system requirements for its computing system safety items in order to effectively convey the software's intended functionality to the FAA. State diagrams, user manuals, flow charts, and other documents are helpful to communicate computing system requirements.

9.3 **Development Process**.

Section § 450.141(d)(3) requires applicants to submit documentation of the development process that meets § 450.141(c), including the minimum attributes required by the regulation. Similarly, § 450.141(d)(4) requires an applicant to provide evidence of the execution of the appropriate development process for each computing system safety item. An applicant must demonstrate its development process for each computing system safety item in accordance with § 450.141(d)(3), and each development process must meet § 450.141(c). If a development process contains more than one distinct set of development process requirements, such as a set of development requirements for applicant-developed computing system safety items and another set for third-party products, then the applicant should specify which set was executed for each computing system safety item. The documentation required of an applicant to meet these requirements will vary somewhat among applications, depending on the complexity of the computing system safety item, but will generally include some combination of audit results, milestone results, and outputs of automated code checking tools. Substantive guidance on demonstrating compliance with § 450.141(c) is provided in paragraph 8.2 of this AC. A software hazard analysis is often needed to ensure that test cases address all necessary nominal and off-nominal conditions, and expands in scope throughout the development process. Although the test cases trace to requirements, the hazard analysis validates the content of the tests.

9.4 **Providing Evidence of the Implementation of Each Safety Requirement**.

Section 450.141(d)(5) requires the submission of evidence of the implementation of each safety requirement. An applicant should submit the combination of test descriptions, test plans, test outputs, and analyses that verifies that each computing system safety item meets each applicable safety requirement. Applicants that plan repeated missions with identical computing system safety items may define a standard set of test reports.

Appendix A. Generic Computing System Safety Requirements.

This appendix provides generic computing system safety requirements that an applicant may use in the design and development of software and computing systems. These generic design requirements represent best practices used in the aerospace community, and are offered here as examples of safety requirements for any application under § 450.141. Additional information and sample 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 SAMPLE GENERAL COMPUTING SYSTEM REQUIREMENTS.[3]

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.

[3] The requirements in this appendix are computing system design requirements as opposed to regulatory requirements.

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

A.1.7 The computer system should verify periodically 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.1.23 NASA Software Engineering Requirements (NPR 7150.2C) requirement SWE-134 provides an alternative set of requirements that is helpful in determining a full set of software requirements for safety critical applications.

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, as appropriate, 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 vehicle processing, the software should maintain the Flight Safety System (FSS) in its current state in addition to meeting the requirements in paragraph A.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 the 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 to the operator of valid command entries, data entries, or both.

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 that 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 health data to vehicle software systems, consistent with anomaly detection requirements and anomaly response requirements. Vehicle 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 implemented correctly and completely. Test results should be analyzed to identify potential safety anomalies that may occur.

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

A.9.3 The 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 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 An 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, or 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.11.1 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.

A.13.1 Software should be designed, developed, and tested in a manner that complies with IEEE 12207, *Systems and software engineering – Software life cycle processes*, or its equivalent.

A.14 Software Coding Practices.

A.14.1 Software developers should apply software engineering criteria to select a programming language, or languages, for the safety software. This includes utilizing the information provided in ISO/IEC TR 24772, *Programming languages — Guidance to avoiding vulnerabilities in programming languages*. Project coding guidelines should be defined for each programming language used in the safety software implementation and should include mitigation for the vulnerabilities described in the relevant parts of ISO/IEC TR 24772.

A.15 Software Reuse.

A.15.1 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, an 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 COTS software, an operator should ensure that every software safety-critical function that the software supports is identified and satisfies the safety requirements.

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 safety 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.

B.1.1 **Typical SFMEA Procedural Steps**.

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 potential software faults and computing system failure modes.
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.

B.1.2 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 should be identified using generalized classifications. Table B-1 shows one example of a classification set derived from information in such standards as IEEE STD 1044 (Inactive). 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 B-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**
Requirements	• System responses not defined for all operating conditions. • Constraints not testable or left untested. • Safe states incorrectly defined. • Code documentation compiled inaccurately. • Programming practices not defined. • Test plan incorrect or incomplete. • Review processes incomplete or inaccurate. • Reused software not fully compatible with new application. • Program assumptions not documented.
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.

Software and Computing System Failure Mode (Software Error) Class	**Specific Software and Computing System Faults and Failures**
	• 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.
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.

Software and Computing System Failure Mode (Software Error) Class	**Specific Software and Computing System Faults and Failures**
	• 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.
Hardware	• CPU overload. • Memory overload. • Unexpected shutdown of the computer. • Wrong file writing. • Wrong interrupt activation. • Wrong data into register or memory. • Processor computation incorrect. • 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.

B.1.3 Table B-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).

B.1.4 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 B-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 B-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 B-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.

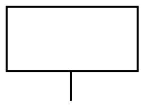
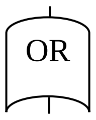
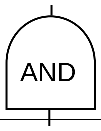
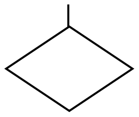
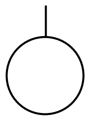
B.2.1 A top-down, deductive study of system reliability, Fault Tree Analysis (FTA) graphically depicts the sequence of events that can lead to an undesirable outcome. An FTA generates a fault tree, which is a symbolic logic model of the failures and faults. As an aid for system safety improvement, an 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.

B.2.2 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 not quantitative because of software's non-probabilistic nature. An SFTA produces safety requirements that can then be implemented in the software life cycle.

B.2.3 Standard logic symbols are used in constructing an SFTA to describe events and logical connections. Table B-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 should 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 requirements, 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 B-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.

B.2.4 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.

B.2.5 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 B-1 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 B-2 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 B-1.

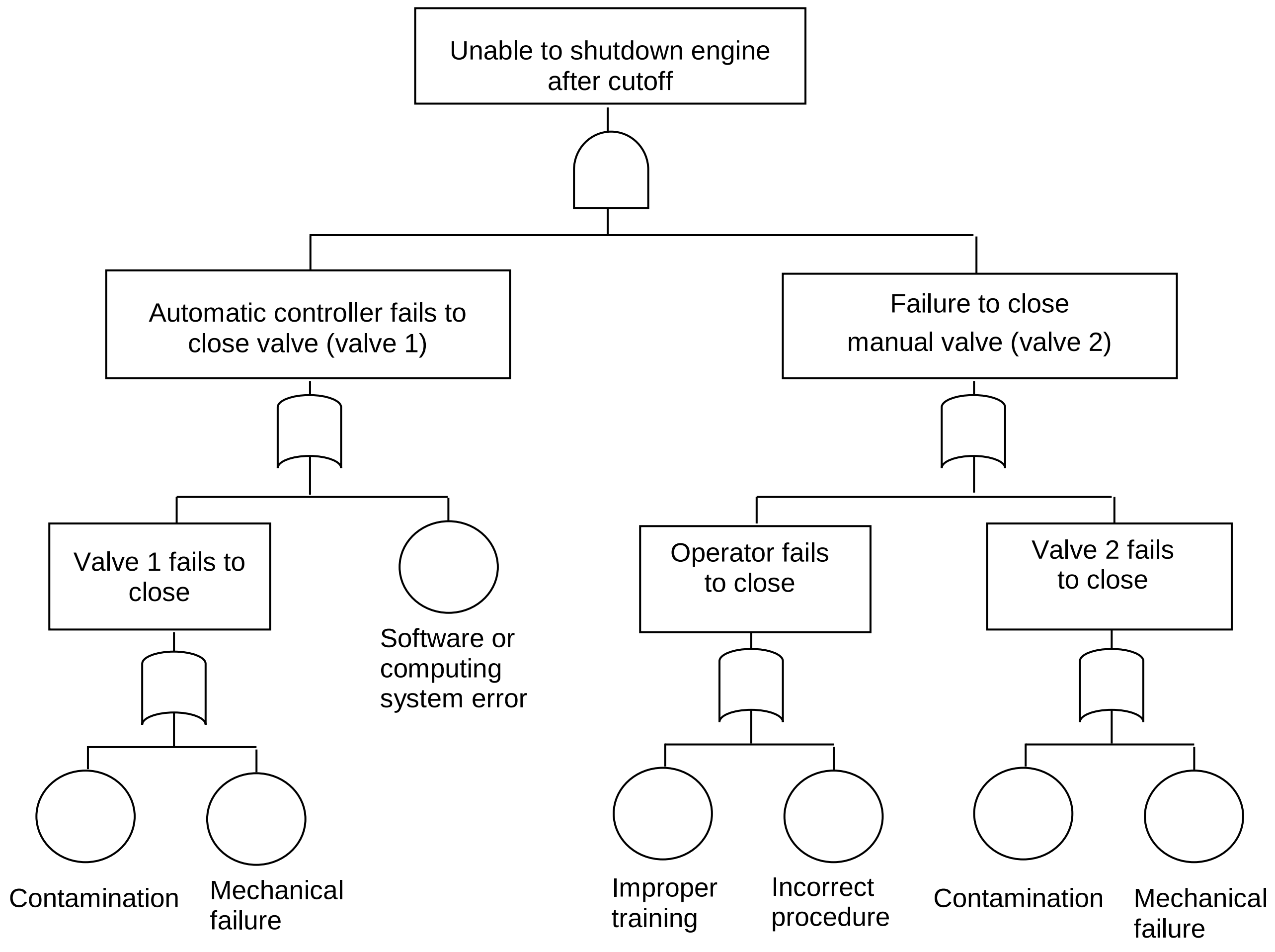


Figure B-1 Fault Tree for Engine Shutdown Failure.

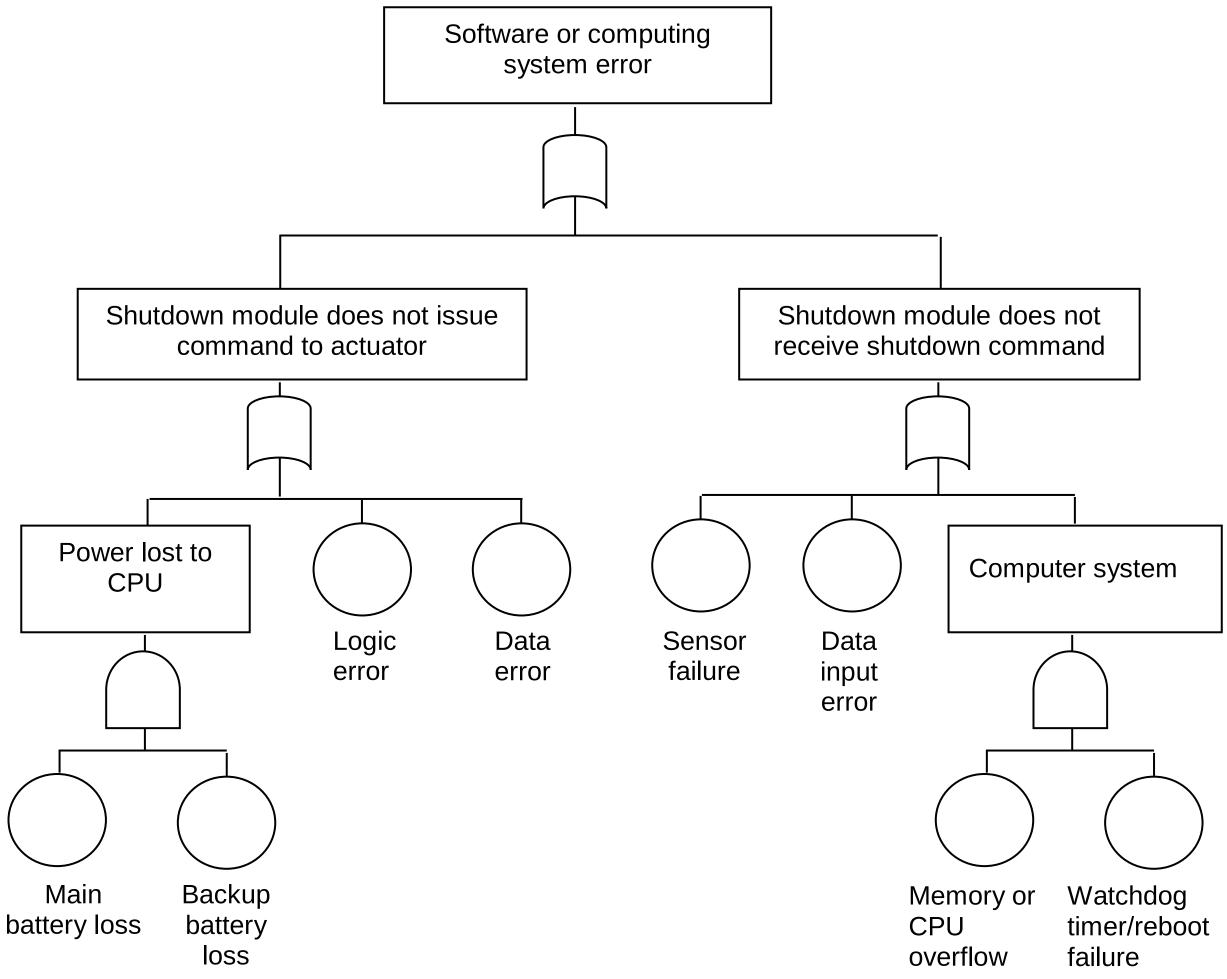

Software or computing system error
Shutdown module does not issue command to actuator
Shutdown module does not receive shutdown command
Power lost to CPU
Logic error
Data error
Sensor failure
Data input error
Computer system
Main battery loss
Backup battery loss
Memory or CPU overflow
Watchdog timer/reboot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

Figure B-2 Fault Tree for Software or Computing System Errors

Appendix C. Space Vehicles 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.

C.2 PHOBOS 1.

C.2.1 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.

C.2.2 A lack of requirements 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.

C.2.3 Lesson Learned.

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

C.2.4 References.

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

C.3 CLEMENTINE.

C.3.1 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.

C.3.2 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.

C.3.3 Lesson Learned.

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

C.3.4 References.

- 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.
- Ganssle, Jack G. 2000. "Crash and Burn." *Embedded Systems Programming*, https://www.embedded.com/crash-and-burn/.
- Harland, David M. and Ralph D. Lorenz. 2005. *Space System Failures: Disasters and Rescues of Satellites, Rockets, and Space Probes.* Berlin: Praxis Publishing Ltd.

C.4 **ARIANE 501.**

C.4.1 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 requirement 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.

C.4.2 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.

C.4.3 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).

C.4.4 References.

- Lions, J. L. 1996. Ariane5: *Flight 501Failure Report by the Inquiry Board*. Paris: European Space Agency.
- O'Halloran, Colin, et al. 2005. "*Ariane 5: Learning from Failure*." Proceedings of the 23rd International System Safety Conference, August at San Diego, California.

C.5 DELTA III/GALAXY.

C.5.1 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.

C.5.2 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.

C.5.3 Lesson Learned.

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

C.5.4 References.

- The Boeing Company. 1998. “Boeing Pinpoints Cause of Delta 3 Failure, Predicts Timely Return to Flight.” Boeing Press Release.

C.6 ZENIT/GLOBALSTAR.

C.6.1 On September 9, 1998, a two-stage Ukrainian-built Zenit 2 rocket, carrying 12 Globalstar satellites, was launched from Baikonur, Kazakhstan. 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.

C.6.2 Lesson Learned:

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

C.6.3 References.

- *Wired News*. “*12 Satellites Go Down in Russia*.” dated September 10, 1998.https://www.wired.com/1998/09/12-satellites-go-down-in-russia/.
- Woronowycz, Roman. “*Crash of Ukrainian Rocket Imperils Space Program*.” *The Ukrainian Weekly*, vol. 66, no. 38, dated 20 September 20, 1998.

C.7 MARS CLIMATE ORBITER.

C.7.1 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.

C.7.2 Incomplete requirements were a contributing factor. The requirements 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.

C.7.3 Lesson Learned.

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

C.7.4 References.

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

C.8 **MARS POLAR LANDER.**

C.8.1 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 shut down 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.

C.8.2 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.

C.8.3 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).

C.8.4 References.

- NASA, Jet Propulsion Laboratory. 2000. Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions. JPL D-18709.
- Leveson, Nancy G. 2004. "The Role of Software in Spacecraft Accidents." AIAA Journal of Spacecraft and Rockets, vol. 41, no. 4, pp. 564-575.

C.9 **TITAN/CENTAUR-MILSTAR.**

C.9.1 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.

C.9.2 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.

C.9.3 Lesson Learned.

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

C.9.4 References.

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

C.10 SEA LAUNCH/ICO F1.

C.10.1 On March 12, 2000, a Sea Launch Zenit 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.

C.10.2 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.

C.10.3 Lesson Learned.

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

C.10.4 References.

- "Sea Launch Poised to Fly with PAS-9." 2000. *Aviation Week & Space Technology*, 3 July.
- Ray, Justin. 2000. "Sea Launch Malfunction Blamed on Software Glitch." *Spaceflight Now*, dated March 30, 2000. http://spaceflightnow.com/sealaunch/ico1/000330software.html.

C.11 COSMOS/QUICKBIRD 1.

C.11.1 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 one hour because a Norwegian tracking station was not ready to monitor the satellite.

C.11.2 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.

C.11.3 Lesson Learned.

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

C.11.4 References.

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

C.12 **MARS ROVER SPIRIT.**

C.12.1 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.

C.12.2 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.

C.12.3 A post-anomaly review showed that memory management risks were not understood. In addition, schedule pressures prevented extensive testing and understanding of software functions.

C.12.4 Lesson Learned.

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

C.12.5 References.

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

C.13 **CRYOSAT.**

C.13.1 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 shut down 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.

C.13.2 Failure analysis showed that the command to shut down 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.

C.13.3 Lesson Learned.

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

C.13.4 References.

- Briggs, Helen. 2005. “Cryosat Rocket Fault Laid Bare.” *BBC News*, dated October 27, 2005. http://news.bbc.co.uk/1/hi/sci/tech/4381840.stm.
- EUROCKOT Launch Services GmbH, “CryoSat Failure Analyzed – KOMPSAT-2 Launch in Spring 2006.” Eurocket Press Release. December 21, 2005, Bermen, Germany.

Appendix D. Commercial, Military, and Experimental Aircraft Accidents.

D.1 COMMERCIAL AND MILITARY SOFTWARE FAILURES.

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 below.

D.2 X-31.

D.2.1 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.

D.2.2 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 configure 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.

D.2.3 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.

D.2.4 Lesson Learned.

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

D.2.5 References.

- Dornheim, Michael A. 1995. "X-31 Board Cites Safety Analyses, But Not All Agree." *Aviation Week & Space Technology*, 4 December, pp. 81-86.
- Haley, Don. 1995. "*Ice Cause of X-31 Crash*." NASA Dryden Flight Research Center, Edwards, California. NASA Press Release 95-203.

D.3 **F-22 RAPTOR.**

D.3.1 On February 11, 2007, a flight of 12 F-22's, on their first deployment to Japan from Hawaii, encountered a multiple system failure at the International Date Line. When the fighters crossed the line, they lost all navigation and attitude indication systems, and parts of their communication and fuel systems including their radios. Fortunately, they were re-fueling at the time and were able to follow the tankers back to Hawaii. The errors were fixed in about 48 hours, and the planes completed their deployment following the software fix.

D.3.2 Lesson Learned.

Software should be tested across the entire environment in which it will operate (*Defense Industry Daily*, 2007).

D.3.3 References.

- "F-22 Squadron Shot Down by the International Date Line." *Defense Industry Daily*, http://www.defenseindustrydaily.com/f22-squadron-shot-down-by-the-international-date-line-03087/.

OMB Control Number: 2120-0746
(Expiration Date: 08/31/2021)

Advisory Circular Feedback Form

Paperwork Reduction Act Burden Statement: A federal agency may not conduct or sponsor, and a person is not required to respond to, nor shall a person be subject to a penalty for failure to comply with a collection of information subject to the requirements of the Paperwork Reduction Act unless that collection of information displays a currently valid OMB Control Number. The OMB Control Number for this information collection is 2120-0746. Public reporting for this collection of information is estimated to be approximately 5 minutes per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. All responses to this collection of

If you find an error in this AC, have recommendations for improving it, or have suggestions for new items/subjects to be added, you may let us know by (1) emailing this form to ASTApplications@faa.gov, or (2) faxing it to (202) 267-5450.

Subject: AC 450.141-1 Date:

Please check all appropriate line items:

☐ An error (procedural or typographical) has been noted in paragraph Click here to enter text. on page

☐ Recommend paragraph Click here to enter text. on page Click here to enter text. be changed as follows:

☐ In a future change to this AC, please cover the following subject: *(Briefly describe what you want added.)*

☐ Other comments:

☐ I would like to discuss the above. Please contact me.

Submitted by: ______________________________ Date: ________________________

Submit

FAA Form 1320-73 (06-2020)