An Analysis of Current Guidance in the Certification of Airborne Software

by

Ryan Erwin Berk

B.S., Mathematics | Massachusetts Institute of Technology, 2002
B.S., Management Science | Massachusetts Institute of Technology, 2002

Submitted to the System Design and Management Program
In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2009

Signature of Author [illegible]
Ryan Erwin Berk
System Design & Management Program
May 8, 2009

Certified by [illegible]
Nancy Leveson
Thesis Supervisor
Professor of Aeronautics and Astronautics
Professor of Engineering Systems

Accepted by [illegible]
Patrick Hale
Director
System Design and Management Program

This page is intentionally left blank.

An Analysis of Current Guidance in the Certification of Airborne Software

by

Ryan Erwin Berk

Submitted to the System Design and Management Program on
May 8, 2009 in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Engineering and Management

ABSTRACT

The use of software in commercial aviation has expanded over the last two decades, moving from commercial passenger transport down into single-engine piston aircraft. The most comprehensive and recent official guidance on software certification guidelines was approved in 1992 as DO-178B, before the widespread use of object-oriented design and complex aircraft systems integration in general aviation (GA). The new technologies present problems not foreseen by the original guidance material and may complicate demonstration of compliance with such guidance material.

The Federal Aviation Administration has deferred to industry experts in the form of the Radio Technical Commission for Aeronautics (RTCA) and the Society of Automotive Engineers (SAE) to create software and system approval guidance. The FAA's Advisory Circular (AC) 23.1309-1C created a four-tier approval system to reduce software and system approval burden on smaller aircraft, creating a lower standard for small aircraft in the hopes that safety gains from technology would outweigh the potential cost of defects from less stringent requirements. Further guidance regarding the role of software in the overall system is spread across other documents including Aerospace Recommended Practice (ARP) 4754, ARP 4761, and applicable SAE Aerospace Standards (AS).

The guidance material, spread across several documents, lacks sufficient specificity to be of use to an everyday practitioner. In addition the guidelines are not necessarily rigorously enforced when the same artifacts are required at different levels of design assurance as stated within DO-178B itself.

After a literature review and regulatory analysis, this thesis investigates some example defects that escaped detection during application of the guidance in a real-world product, making specific recommendations to improve the guidance in a usable way.

Thesis Supervisor: Nancy Leveson
Title: Professor of Aeronautics and Astronautics, Professor of Engineering Systems

This page is intentionally left blank.

Acknowledgments

First and foremost I must thank my wife, Leigh-Ann, for encouraging me to return to school. Without your support, none of the last two and a half years would have been possible. The late nights, weekends, and vacations I spent reading, writing, and performing simulations are finally over!

To my mother and father, thank you for your never-ending support of my educational aspirations. It began when I was just a boy and continues now as I approach 30 years old. You convinced me to come to Boston and attend MIT as an undergraduate, and I have not regretted it. An MIT connection brought me my first real job, and that job has returned me to MIT. It was your encouragement to travel far from our home in Wisconsin to the East Coast that has led me down the path I now find myself. Thank you for letting go enough to allow me to find my own way.

To my sponsor company, thank you for your support and commitment to me. To my first boss, Steve "Jake" Jacobson, thank you for taking a chance on me. I know I didn't fit the mold of a typical employee, but then again, you never did hire me. Thank you for not throwing me off the premises when you returned from vacation and found me sitting at a computer. Thank you for your support over the last seven years, and thank you for your help in acquiring sponsorship. I doubt I would be where I am today without your patronage. To my current boss, Steve Rosker, thank you for your flexibility and understanding as I complete this journey. Without your support I might have dragged this thesis out for a few years. To Yossi Weihs, thank you for introducing me to the Engineering Systems Division at MIT and encouraging me to return to school. I would not be where I am today without your guidance and support of my career growth.

To my advisor, Nancy Leveson, thank you for opening my eyes to the different facets of my profession. Thank you for the guidance in selecting a topic, the reviews, and the late-night email responses to my questions. The more I talk to people, the more I realize what a valuable resource I found in you. I will never look at another safety document quite the same way since meeting you.

To Pat Hale, Bill Foley and Chris Bates, thank you for running a great program and continuing to evolve the program as the needs of the students change. And to the rest of my SDM cohort, thank you for all the effort on our group projects and assignments as we worked our way through the program. I appreciate all that my teammates have done for me and all that we have accomplished together.

It's been an enlightening six and a half years at MIT.

This page is intentionally left blank.

Table of Contents

This page is intentionally left blank.

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1 Introduction

The Growth of Software

Aircraft have flown with mechanical instruments since long before the first electromechanical instruments were conceived. Eventually the mechanical instruments have given way to more sophisticated electromechanical instruments and electronic displays, exchanging those old steam gauges with high-tech replacements.

Technological advances in the 1970s and 1980s allowed hardware to shrink in size, and newer technologies such as liquid-crystal displays have further reduced the space required to create electronic instrument displays. These advances have allowed for a rapid increase in the functionality allocated to software instead of hardware. Software controls a vast number of decisions and connections in modern aircraft: from fuel pumps, to autopilots, to radios, to primary instruments. As an example, the nation is covered with radio towers that broadcast on specific VHF frequencies to allow determination of position. The first radios were completely analog, but over time the analog tuners were replaced with digital tuners that use software to configure the receiver. Now all-digital radios are being produced that use software to tune the frequencies and process the signals received from the VHF stations.

To address the growing use of software in aviation, the Radio Technical Commission for Aeronautics, known as the RTCA, released a document of guidelines for software development in 1982. The document was the result of consensus opinion from a wide range of domain experts periodically meeting as part of a working group tasked with the creation of guidelines. This document was titled DO-178, Software Considerations in Airborne Systems and Equipment Certification. Its purpose was to provide industry-accepted best practices to meet regulatory requirements for airworthiness such as Part 23 and Part 25 of the Federal Aviation Regulations (FARs). The document was revised in 1985 (DO-178A) and again in 1992 (DO-178B) to reflect the experience gained and criticisms of previous revisions. The current revision (in 2009) is DO-178B, and there are many references to DO-178B throughout this document.

It should be noted that DO-178 is not the only means of compliance with airworthiness regulations and that no portion of DO-178 is official government policy. RTCA is an association of government and private organizations that working together to develop recommended practices, but applicants for software approval are free to demonstrate compliance to the certification authority using any other acceptable means. Despite the possibility of developing software using alternative means of compliance, DO-178B is the de facto standard (Jackson, et al. 2007) by which all software submissions for approval are judged at the local Aircraft Certification Offices (ACOs).

Motivation

I became interested in improving the certification process for software from my first month after college graduation. I started my first "real" job, and the first document handed to me after the employee handbook was DO-178B. I read the standard twice, and I still did not comprehend what I would be doing with the document. Even now, after seven years, I can re-read the standard and discover subtleties I have missed in previous readings. As I gained experience I could relate what I have learned to seemingly innocuous statements and gain critical insight into the "why" of some aspects of the standard.

One of my concerns with the standard is the lack of specificity in the guidance. Two different designated engineering representatives (DERs) may come to different conclusions regarding the acceptability of a particular document or artifact based on their particular experiences and preferences because the guidelines are quite vague. One of the issues with consensus-based documents may the lack of specificity to ensure that each of the members can continue their current practices while abiding by the guidelines because the guidelines are quite broad.

Reading the standard, I realize now that the guidance document tries to balance the need for tangible content against the heterogeneity of airborne software and company processes. While the standard provides information, it does not provide specific, useful information that can be immediately applied.

One of my first tasks as a new-hire was to develop low-level requirements for a product that was fairly well along the prototype path. I went to DO-178B to figure out how to write a low-level requirement, and the standard was almost silent on the topic. The only definition I could find in DO-178B was, "Low-level requirements are software requirements from which Source Code can be directly implemented without further information." I asked other engineers, but no one had significant experience with low-level requirements or DO-178B software developed to a design assurance of Level C, the lowest design assurance level that requires low-level requirements be written, traced, and verified.

From then on I have been involved in refining our corporate processes and standards in addition to auditing for compliance and "goodness" of process outputs from other development teams. Through this document I intend to provide practical recommendations for improvements to the software certification process under DO-178B in addition to providing meaningful root cause analysis and process improvements to my sponsor company.

Chapter 2
Literature Review

Research and study has been devoted to software standards and certification for as long as certification guidelines have existed. Examining the revisions of the certification guidelines demonstrates a fact that guidelines are constantly being reviewed, revised, and replaced. DO-178, the first release of Software Considerations in Airborne Systems and Certification was a goal oriented process document written at the conceptual level released in 1982 (Johnson, 1998). Companies developed to the standard by meeting the "intent" of the document. DO-178A was the first major revision of the standard, released in 1986, followed by DO-178B, another major revision, in 1992. The guidelines changed dramatically between DO-178A and DO-178B as practitioners gained experience with DO-178A and recommended changes as part of an industry working group.

Comparisons

Civilian aviation software was not the only area developing software standards. The United States Military provided its own software standards such as MIL-STD-1679, MIL-STD-2167, and MIL-STD-498. The European Space Agency has also released PSS-05-0, titled ESA Software Engineering Standards, in response to the growth of software and complex systems safety concerns. With these standards brings the comparisons of DO-178B with MIL-STD-498 and

ESA PSS-05-0. The ESA standard is more direct and specific than DO-178B, providing templates and examples of methods of compliance while DO-178B is more abstract and process oriented (Tuohey, 2002).

Software approval and certification standards are not unique to aerospace. Software security professionals along with the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) have established the Common Criteria for Information Security Evaluation to provide guidelines for developing secure software for information technology. Further comparisons map Common Criteria assurance classes on to DO-178B processes. Such mapping demonstrates gaps in the DO-178B processes such as assurance of Delivery and Operation (Taylor et al., 2002).

Value

Others have questioned whether certification makes a difference in the quality of the final product or whether other factors are at work (Jackson, et al., 2007). Many of the DO-178B mandated procedures are difficult to trace to direct effects on the quality of the software output. Some have hypothesized that the development of software within a "safety culture" may be the key to developing software with fewer defects. While software engineers are notoriously protective of their source code, operating in an environment with human lives at stake may encourage them to value the opinions of colleagues, act cautiously, and pay meticulous attention to detail (Jackson et al., 2007).

While Jackson laments the strictures put in place by DO-178B, others appreciate the flexibility provided by the standard. Though DO-178B was released in 1992, it does not preclude the use of agile methods that gained popularity in the 1990s. Agile methods rose in response to the traditional waterfall methods used in "big-aerospace" for twenty-five years. DO-178B Section 4 emphasizes transition criteria in an apparent endorsement of a waterfall development process, but the individual development process sections allow for feedback and iteration at any stage. Some agile methods may be used to satisfy DO-178B objectives. At higher certification levels, (Levels A and B), satisfaction of the development objectives requires demonstration of independence in the development of requirements, code, and test. The goal of the independence objectives may be to minimize potential interpretation problems arising from a single person reading and implementing a requirement. The agile method of pair programming can be used to meet the independence objectives (Wils, 2006).

General Concerns

The generality of the DO-178B standard likely derives from the way in which the standard was developed and ratified, specifically by consensus. It would be very difficult to achieve consensus on specific templates or document formats as in ESA PSS-05-0 in an environment of competing aviation professionals. (Hesselink, 1995) argues that DO-178B is not intended to be a software development process, but a point of common understanding between the applicant and the certification authority in his comparison of DO-178B with MIL-STD-2167A and ESA PSS-05-0. Others support that thought by describing DO-178B as an *assurance standard*, not a *development standard* (Sakugawa et al, 2005). None of those standards are self-contained, but by maintaining a process-oriented standard, DO-178B makes it the most flexible of the three standards.

The broadest identification of concerns in the certification of airborne software comes from Sakugawa, Cury, and Yano. They provide a list of relevant technical issues and brief discussion ranging from high-level concerns, such as the purpose of DO-178B, to detailed issues, such as configuration control of software compatibility spread across systems. Much of the discussion identifies concerns that should be addressed in future revisions to the guidance material but fails to provide any evidence or suggestions as to improvements. By contrast, this document includes specific recommendations to improve the certification guidance material supported by examples from industry.

Chapter 3
Software Approval Process

The software approval process is just one part of an overall aircraft certification. The FARs govern the airworthiness of an aircraft, and the software covers just a portion of all that is required by the airworthiness regulations. As outlined by section 2.1 of DO-178B and shown in **Figure 1** below, the system life cycle processes including a system safety assessment (SSA) process evaluate the product development at the aircraft installation level.

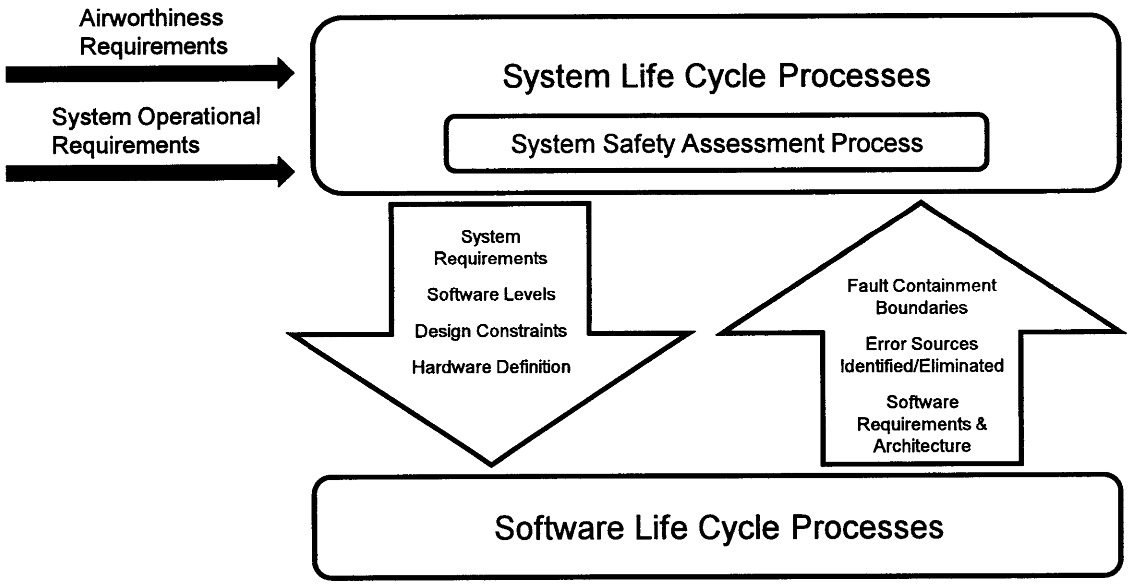
Airworthiness Requirements
System Operational Requirements
System Life Cycle Processes
System Safety Assessment Process
System Requirements
Software Levels
Design Constraints
Hardware Definition
Fault Containment Boundaries
Error Sources Identified/Eliminated
Software Requirements & Architecture
Software Life Cycle Processes

Figure 1 Overview of the Development Process

A major part of the development of a system for airborne installation is the analysis of the airworthiness and system operational requirements. The system requirements must be defined and determined to meet the airworthiness requirements set by the certification authority. Only after the airworthiness analysis has been performed should the system requirements be allocated between software and hardware. Guidelines for performing a system safety assessment are outlined in documents released by the Society of Automotive Engineers (SAE) as Aerospace Recommended Practice (ARP) documents. The specific documents that are called out by the FAA Advisory Circular (AC) 23.1309-1D are ARP 4754, Certification Considerations for Highly-Integrated or Complex Aircraft Systems, and ARP 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. A detailed discussion of the safety assessment process and compliance with airworthiness requirements is beyond the scope of this document.

As software design becomes more complex, features once allocated to hardware such as circuit breakers are more frequently allocated to software because the software can be changed with little marginal cost after the aircraft has been certified. A software development is a costly proposition, and more costly at higher design assurance levels, but changing a piece of dedicated hardware may be more expensive than designing in the flexibility in software from an early stage.

Once system requirements have been allocated to software and the software design has implemented the requirements or sufficiently determined the design regarding those requirements, feedback—in the form of derived requirements—must be provided back to the safety assessment process or team to ensure the architectural implementation meets the design assurance levels specified by the safety assessment and that no assumptions of the safety assessment have been violated by the software architecture. The constant feedback between the SSA process and the software development process is designed to ensure that the final product meets the design assurance levels required by the SSA as well as the airworthiness requirements to receive approval for installation in certified aircraft.

Part 23 vs. 25

The FARs are technically a portion of Title 14 of the Code of Federal Regulations (CFR). This thesis examines DO-178B only as it pertains to Part 23 of 14 CFR in an effort to restrict the scope to a manageable level. The main difference between Part 23 and Part 25 aircraft is the size and weight of the aircraft. Part 23 aircraft are restricted to have gross takeoff weights of 12,500 pounds or less, while aircraft certified under Part 25 have no such restrictions. In more concrete terms, Part 23 covers small aircraft up through Commuter category aircraft, while Part 25 governs larger business jets up through air transport and commercial airliners. Certain ACs are specific to Part 23 or Part 25 of the FARs and this document will only discuss the Part 23 ACs. DO-178B was originally intended for aircraft certified under Part 25 of the FARs. It was not until the release of AC 23.1309-1C in 1999 that DO-178B could be applied to small aircraft in a cost-effective manner.

AC23.1309-1C, 1999

Published in 1999, AC23.1309-1C was a watershed document for light general aviation. This AC lowered the minimum required design assurance levels for Part 23 aircraft from the DO-178B mandated levels intended for Part 25 aircraft. The reduction in minimum DALs opened the door to lower-cost avionics development projects by reducing the certification burden and overhead required to receive approval from the certification authority. A critical product of the SSA as outlined in ARP 4761 is the required minimum software design assurance levels for different aircraft functions. There are five levels of design assurance as outlined in DO-178B (expanded from three levels in DO-178A), and the five levels mirror the five levels of failure conditions.

Classification of Failure Conditions	**No Safety Effect**	**<----Minor----->**	**<----Major---->**	**<--Hazardous--->**	**< Catastrophic>**
Allowable Qualitative Probability	**No Probability Requirement**	**Probable**	**Remote**	**Extremely Remote**	**Extremely Improbable**
Effect on Airplane	No effect on operational capabilities or safety	Slight reduction in functional capabilities or safety margins	Significant reduction in functional capabilities or safety margins	Large reduction in functional capabilities or safety margins	Normally with hull loss
Effect on Occupants	Inconvenience for passengers	Physical discomfort for passengers	Physical distress to passengers, possibly including injuries	Serious or fatal injury to an occupant	Multiple fatalities
Effect on Flight Crew	No effect on flight crew	Slight increase in workload or use of emergency procedures	Physical discomfort or a significant increase in workload	Physical distress or excessive workload impairs ability to perform tasks	Fatal Injury or incapacitation
Classes of Airplanes:	**Allowable Quantitative Probabilities and Software (SW) and Complex Hardware (HW) DALs (Note 2)**				
Class I (Typically SRE under 6,000 lbs.)	No Probability or SW & HW DALs Requirement	$<10^{-3}$ Note 1 & 4 P=D, S=D	$<10^{-4}$ Notes 1 & 4 P=C, S=D P=D, S=D(Note 5)	$<10^{-5}$ Notes 4 P=C, S=D P=D, S=D(Note 5)	$<10^{-6}$ Note 3 P=C, S=C
Class II (Typically MRE, STE, or MTE under 6000 lbs.)	No Probability or SW & HW DALs Requirement	$<10^{-3}$ Note 1 & 4 P=D, S=D	$<10^{-5}$ Notes 1 & 4 P=C, S=D P=D, S=D(Note 5)	$<10^{-6}$ Notes 4 P=C, S=C P=D, S=D(Note 5)	$<10^{-7}$ Note 3 P=C, S=C
Class III (Typically SRE, STE, MRE, & MTE equal or over 6000 lbs.)	No Probability or SW & HW DALs Requirement	$<10^{-3}$ Note 1 & 4 P=D, S=D	$<10^{-5}$ Notes 1 & 4 P=C, S=D	$<10^{-7}$ Notes 4 P=C, S=C	$<10^{-8}$ Note 3 P=B, S=C
Class IV (Typically Commuter Category)	No Probability or SW & HW DALs Requirement	$<10^{-3}$ Note 1 & 4 P=D, S=D	$<10^{-5}$ Notes 1 & 4 P=C, S=D	$<10^{-7}$ Notes 4 P=B, S=C	$<10^{-9}$ Note 3 P=A, S=B

Note 1: Numerical values indicate an order of probability range and are provided here as a reference. The applicant is usually not required to perform a quantitative analysis for minor and major failure conditions. See figure 3.
Note 2: The alphabets denote the typical SW and HW DALs for most primary system (P) and secondary system (S). For example, HW or SW DALs Level A on primary system is noted by P=A. See paragraphs 13 & 21 for more guidance.
Note 3: At airplane function level, no single failure will result in a catastrophic failure condition.
Note 4. Secondary system (S) may not be required to meet probability goals. If installed, S should meet stated criteria.
Note 5. A reduction of DALs applies only for navigation, communication, and surveillance systems if an altitude encoding altimeter transponder is installed and it provides the appropriate mitigations. See paragraphs 13 & 21 for more information.

Figure 2 Relationships Among Airplane Classes, Probabilities, Severity of Failure Conditions, and Software and Complex Hardware DALs, from AC23.1309-1D

The as-written regulations prior to AC 23.1309-1C required that primary instruments and other systems critical to the continued safe operation of an aircraft must be certified to Level A. The figure above shows the updated guidance information from 23.1309-1C. Examining Class I and Class II aircraft, the most critical failure category, Catastrophic, requires a minimum DAL of just Level C, two levels lower than the DO-178B guidance requires.

According to the FAA, accidents in Part 23 certificated aircraft are primarily attributed to pilot error and a loss of situational awareness (AC 23.1309-1C). The resulting behavior of a disoriented pilot is typically known as controlled flight into terrain or CFIT. The FAA believes that equipping these small aircraft with additional technologies that provide more easily understandable information regarding aircraft position and attitude will reduce the severity and frequency of such errors, improving overall aviation safety.

Certifying software to Level A is a very expensive proposition for development companies, and prior to 1999 there was limited adoption of complex avionics in Part 23 aircraft because of the relatively high prices required by avionics manufacturers to recoup the costs associated with a Level A certification. By reducing the minimum required DALs for small aircraft, the FAA encourages new avionics development at lower price points due to lower costs associated with the certification activities. The FAA believes that increasing the equipage rates of small aircraft will provide a safety benefit that outweighs the risks associated with reduced design assurance levels in Class I-Class IV aircraft.

One of the criticisms of the entire DO-178B/ARP4754/ARP4761 process is the lack of cohesiveness between the regulatory guidance provided and a concrete implementation of the process in certifying an aircraft. All of the aforementioned documents were developed for Part 25 aircraft. While DO-178 and DO-178A outlined the role of software and its relation to overall system safety, the current revision of the standard has removed the references to Type

Certification (TC), Supplemental Type Certification, Technical Standards Order Authorization and equivalent FARs (Johnson, 1998). AC23.1309-1C lowered the minimum bar for small aircraft but fails to address some of the changes required amongst all the documents to adequately handle Part 23-specific changes such as conducting a system safety assessment for Part 23 aircraft. The AC specifically cites ARP 4754 and ARP 4761 as only partially applicable for Part 23 certification, but fails to identify what portions of those documents should be satisfied and what portions can be disregarded. This oversight implies the certification system as a whole was not prepared or intended to handle such modernization in 1999 when the last well-prepared modifications to the software certification guidelines were released in 1992.

Another common criticism of the structure of the FAA can be seen in AC 23.1309-1C. The FAA's mission demonstrates a bit of a conflict of interest. The mission statements on the FAA website (FAA Website, 2009) indicate that the FAA's major roles include:

1. Regulating civil aviation to promote safety
2. Encouraging and developing civil aeronautics, including new aviation technology

The FAA is responsible for promoting safety, yet it also promotes new technologies. AC23.1309-1C promotes new technology by reducing required DALs for software in small aircraft, allowing installation of safety-enhancing equipment into aircraft where previous avionics were not cost-effective. It seems that the FAA is walking a fine line by reducing safety requirements to increase safety by increasing the installed base of advanced avionics. While I agree with the assertion that increasing the availability of advanced avionics in Part 23 aircraft will have a positive impact on safety for the Part 23 aircraft, the overall impact on the National Aerospace System (NAS) with increased air traffic in instrument meteorological conditions (IMC) may have a negative effect on air safety for the Part 25 aircraft relied upon by the majority of the traveling public.

Recommendation: *Separate the aviation promotion and safety aspects of the FAA to preclude any conflicts of interest regarding reduced safety in the name of advancing technology and popularizing general aviation.*

Within the past few months AC 23.1309-1C has been superseded by AC23.1309-1D, replacing a previous hardware design assurance standard with the new DO-254, Design Assurance Guidance for Airborne Electronic Hardware, released in April of 2000. AC23.1309-1D followed almost nine years after the release of DO-254. That delay is indicative of the speed at which some aerospace regulations can be expected to be updated; change happens slowly.

System Safety Process

DO-178B calls out SAE ARP 4754 and SAE ARP 4761 as recommended guidance in developing complex airborne systems and performing system safety assessments. Examining DO-178B Section 5.1.2, the document requires derived requirements to be provided to the system safety assessment process for evaluation against assumptions and calculations made while administering the process outlined in ARP 4761. A system safety assessment team would typically consist of different aircraft-level experts allocating functionality and defining high-level functional failure modes. Such personnel may be insufficient in assuring system safety. One of the recommendations from the literature is to ensure that software engineers and code developers understand the software-related system safety requirements and constraints (Leveson, 1995). By highlighting the requirements and features that are safety-related and needed to support the preliminary SSA, software engineers will have the insight into the intent and purpose of the requirement in addition to the text of the requirement.

Can the system safety assessment team adequately comprehend the implications of a single low-level design choice such as the use of a ping-pong buffer scheme as it relates to the aircraft level functionality? The aircraft level would typically exist at least two levels of abstraction higher than the level at which the software requirement is describing some internal design detail of the software. As shown in **Figure 1**, the software requirements must trace up to system requirements, and the system requirements derive from a combination of airworthiness requirements and system operational requirements. It would seem that the composition of aircraft-level safety analysts might not be the appropriate personnel to critically evaluate detailed design and architecture decisions as increases in software complexity have outpaced our ability to master new technologies such as object-oriented design.

Recommendation: *The system safety assessment team must include software safety expertise to evaluate software design decisions and the implications of those decisions in the context of overall system and aircraft safety margins.*

Human Factors

With the increasing complexity in avionics and airborne software comes an increasing level of information conveyed to the pilot via displays, buttons, and knobs. Safety standards and TSOs govern basic aircraft functionality and required depictions and abbreviations, but there has been very little guidance regarding human factors and the human-machine interface. Safety assessments have focused on failure rates of hardware and design assurance levels of software but fail to explicitly consider the exact manner in which the data is presented to the pilot. As an example, a digital readout of manifold pressure is less visually compelling than a digital readout below a tape or dial presentation of the same information. The impact of user interface design is difficult to quantify, as the evaluation is subjective. That subjectivity makes it very difficult to provide consistent safety analysis guidelines for user interfaces. The National Transportation Safety Board (NTSB) realizes that the only evaluations of software user interface occur late in the certification process (NTSB, 2006) as part of the FAA's Multi-Pilot System Usability Evaluation (MPSUE). A MPSUE consists of a group of FAA expert pilots that evaluate the proposed system for compliance with FARs and other industry guidance such as readability, color choices, and pilot workload. The evaluations are subjective by their very nature and only peripherally address aircraft safety issues as impacted by user interface.

Structural Coverage

As software complexity increased in the 1970s and 1980s it became obvious that exhaustive path testing of a software program would be cost-prohibitive. Imagine a program consisting of 60 different conditional checks. To exhaustively test each possible path in such a program, it would require 2^60 (1.15e18) tests. By comparison, the universe is only 4e17 seconds old, so developing and running such a test suite for even a simple program as described is not feasible (Myers, 1976). To that end, structural coverage has been accepted as a proxy for path coverage.

DO-178B defines three different levels of structural coverage for demonstration at three different design assurance levels. The lowest DAL that requires structural coverage is Level C. Structural coverage of code developed per a Level C process requires statement coverage, or coverage of every executable statement or block of source code. Level B structural coverage adds the requirement of branch coverage in addition to statement coverage. Branch coverage requires execution of every decision or local path at least once. In source code written in C/C++, this requirement implies that implicit "else" cases and switch/case defaults are explicitly tested and covered. Level A structural coverage requires demonstration of modified condition/decision coverage (MCDC). MCDC adds the requirement that each part of a conditional check can affect the outcome of the decision.

Each of the structural coverage requirements presented by the ascending DALs requires more effort than the preceding level. Total productivity can slow to 100 lines of source code per man-month at the highest level of design assurance (Binder, 2004). Some evidence has recently been presented indicating that the additional certification requirements for Level A including MCDC do not show any increase in safety or reliability of the software (Jackson et al, 2007). Other research recommends additional coverage metrics such as mandatory loop coverage for safety-critical software development (NASA 2004). Loop coverage ensures that software control loops are verified to perform the intended function when each loop body is executed zero times, exactly once, and more than once (consecutively). Such loop testing provides additional confidence that the loops under test do not exhibit adverse behavior that may be missed when performing MCDC or branch coverage.

Reuse

One of the more recent criticisms of DO-178B in the wake of AC 23.1309-1C has been the difficulty of reusing software in avionics. Prior to the relaxed standards introduced in 1999, Commercial Off The Shelf (COTS) software was not widely implemented because of the certification costs for software. COTS software is more general than a custom piece of software designed for a specific goal because COTS parts are developed to be sold for use in more than one environment. That flexibility requires source code beyond the minimum required set for any given use, and that extra code requires certification under DO-178B as prescribed under Section 2.4 (f). Certifying COTS under DO-178B requires a significant amount of effort (Maxey, 2003) and the use of COTS has not been demonstrated to make avionics products safer.

The FAA has heard the complaints of authors such as Maxey and attempted to address the changing landscape of embedded software systems by releasing AC 20-148 in December 2004. That document establishes a certification mechanism for creating a Reusable Software Component (RSC) that can be used for software features such as algorithm libraries and operating systems. An RSC can be used to satisfy some of the certification objectives of DO-178B by providing an integration package of life-cycle data to attach to the applicant submission for approval. RSCs require verification of the code in the target environment as well as satisfaction of design constraints within which the original software approval was granted. The virtues of RSCs are just now becoming apparent, but some believe the use of RSCs will usher in a new era of productivity as development firms can focus on product specifics instead of core services (Matharu, 2006).

Development

Another criticism of DO-178B has been the rigor with which the development standard was followed for a piece of software. It is possible to develop a piece of software using any process and subsequently back-fill the requirements, design information, and reviews (Kim et al, 2005). Creating the life-cycle data *a posteriori* defeats the purpose of developing to a standard (Brosgol, 2008), yet systems often must be adapted to meet certification guidelines after development. DO-178 has evolved over three revisions and DO-178B Section 12.1.4 discusses the upgrading of a development baseline to bring software into compliance with the current guidance. Section 12.1.4 sub-paragraph d specifically calls out the use of reverse engineering to augment inadequate life-cycle data to meet the objectives specified throughout the standard. Such reverse engineering has been encouraged and purported to provide value to legacy systems (Hicks, 2006), though others have claimed that it is not possible to adequately reverse engineer a product towards a standard (Hesselink, 1995).

The first drafts of DO-178B appeared in 1989 and the final version of DO-178B was released in 1992, before the widespread use of object-oriented (OO) design in airborne systems. While DO-178B does not preclude the use of OO, it seems clear that parts of OO design complicate the satisfaction of objectives in DO-178B. The guidance documents require traceability between requirements and source code. Such traceability is relatively easy to demonstrate when programming procedurally as a method or group of methods contained within a module perform a task that traces directly to requirements. OO design creates mental abstractions that may spread directly related functions across modules and complicate traceability.

Another DO-178B objective complicated by OO is the handling of dead and deactivated code. DO-178B provides the following definitions of dead and deactivated code:

> Dead code - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (Data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers.
>
> Deactivated code - Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.

OO design provides the ability to override a method provided by a base class in a child class. The overriding may create dead or deactivated code in a module that has not been modified in the software release under development, yet such effects may be difficult to detect outside of a human review. In response to the shortcomings of DO-178B with regards to OO technology, certification authorities from around the world have highlighted unique problems presented by OO in the Certification Authorities Software Team (CAST) Position Paper CAST-4. CAST-4 does not provide any additional regulation or guidance over DO-178B, but instead highlights the relevant concerns regarding the use of OO by an applicant. The issues raised by the position paper should receive consideration in DO-178B documents such as the Plan for Software Aspects of Certification (PSAC) and the Software Accomplishment Summary (SAS). Industry experts have also generated the Object-Oriented Technology in Aviation (OOTiA) Handbook in an effort to identify other concerns in the use of OO in airborne systems. The OOTiA Handbook was released in 2004 (12 years after DO-178B) and provides discussion of the issues and examples of how to use a subset of OO techniques that permit the demonstration of compliance with DO-178B objectives.

Verification

Others have criticized the regulatory guidance for its lack of testing discussion. Section 2.7 of DO-178B is titled "Software Considerations in System Verification". The very first sentence of the section reads, "Guidance for system verification is beyond the scope of this document." With the advent of low-cost MEMS sensors for attitude, air data, and navigation systems, one software "product" can effectively satisfy an aircraft level feature such as VHF navigation, for example. The scope of the software has increased since 1992, and system verification may need to be addressed because the software and hardware within a single product is the entire system. Whether I put an integrated attitude display system in a car, a boat, or a plane, the entire set of functionality (aside from power) is under the control of a single software product and software testing becomes synonymous with system testing. Furthermore the software requirements flow down from system requirements, yet there is no explicit mention of system requirements testing as DO-178B only provides guidance on software considerations. DO-178B was written under the assumption that the aircraft certification applicant is the same entity as the software developer (Sakugawa et al., 2005), but the avionics industry has expanded to include development firms that sell complex avionics to original equipment manufacturers (OEMs). How can a development firm create system requirements and allocate/expand those system requirements into software requirements but never verify the system requirements are satisfied at a system level?

Recommendation: *Regulatory guidance should outline testing processes such as an Integrated System Test Plan (ISTP) to verify that System Requirements are met as the system is integrated into a target aircraft.*

Others have suggested that DO-178B Section 6, Software Verification Process, will need revision soon. Software complexity is increasing rapidly and current testing methods will not scale to meet orders of magnitude more complex software (Allerton, 2006). DO-178B recognizes that verification is not just testing, but a combination of reviews, analyses, and tests. The difference between analysis and review is reviews are more qualitative while analysis is repeatable and often quantitative. "Analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness." For example, a verification engineer could verify the system timing to ensure that attitude display meets published minimum operating performance standards (MOPS) for application of a Technical Standards Order (TSO). One technique might be to review the source code and associated timing tables to ensure that the system/software requirements are met in addition to the MOPS. The review relies on a human's reading of the applicable requirements and review of source materials that directly determine the software behavior and characteristics. An equivalent analysis might include development of minimally intrusive software or hardware instrumentation that measures or signals the timing characteristics of the software. Such an analysis is repeatable as required by the definition from the document. In addition the analysis could provide an objective answer as to confidence that the software behaves as expected. The repeatability of an analysis makes it preferable to a qualitative or subjective review as a means of verification.

Other verification tasks do not lend themselves to analytic techniques as above. DO-178B requires demonstration of traceability from source code and tests all the way up to system requirements (unless intermediate requirements are derived). While analysis may be used to verify that all requirements are traced, only review can determine the correctness of the trace between requirements because human interpretation is required to understand the implications of any given requirement. The implications must be considered not only for the directly traced requirements but also for the untraced but applicable requirements. Human review techniques are better suited to such qualitative judgments than are analyses.

Section 6.4 of DO-178B discusses the Software Testing Process as it pertains to overall software verification and calls out two objectives from testing. The first objective is to show that the requirements are fully satisfied, known as requirements coverage. The second goal of testing is to demonstrate "with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed". The second objective is noble, but rather abstract. To what level of confidence should software be tested? Earlier in the guidance document the writers acknowledge software does not exhibit hardware-like failure rates because software is deterministic. Full path testing is not possible for complex software as demonstrated above in the example from (Myers, 1976). The guidance document is silent as to how to achieve that level of confidence through testing; leaving the development firm to make an engineering judgment as to how much testing is enough.

There are many gaps in DO-178B, AC 23.1309-1C/-1D and the associated ARPs for software approval. Through review, refinement, and iteration the guidance writers should be able to release a new draft of the DO-178 that addresses the concerns raised over the last seventeen years, incorporating experience and suggestions from published research such as this document.

This page is intentionally left blank.

Chapter 4
Case Studies

Methodology

Each of the following cases represents a real-world example of defects in software design according to DO-178B compliant processes. Some of the examples demonstrate gaps in DO-178B coverage or treatment of specific artifacts required for a successful product development. Other examples demonstrate software developed following the guidance but still include defects that the process does not protect against.

The five examples that follow each introduce the problem briefly, identifying the high level feature and the defect that was observed. Each example then covers the technical details necessary to understand the complexity of the defect. In addition the exact source code that caused the behavior is identified. A discussion of the DO-178B process artifacts follows in the certification discussion section for each example, identifying the associated requirements, tests, and development processes associated with the defect. Those artifacts demonstrate defects in the process from which the source code is developed. By identifying the process defects, I will identify recommendations for improvements to the DO-178B guidance material to assist others in preventing the same types of defects that are under investigation. While every software project is unique, the suggested DO-178B process improvements should provide concrete improvements in certified software development processes.

Product Information

All of the examples presented below are extracted from the defect tracking system for a piece of avionics known as a Primary Flight Display, or PFD. The PFD incorporates eleven different compiled applications spread across nine processors, but all the defects under review here are from the highest-level processor that controls the display and user interface functionality. The PFD's main purpose is to provide critical aircraft flight parameters, integrated in a single piece of avionics. Aircraft air data and attitude information, such as altitude, airspeed, pitch, roll, and heading, are provided via an integrated sensor solution. Other flight parameters such as the active flight plan, traffic information, and radio navigation aids are provided over standardized digital databus interfaces.

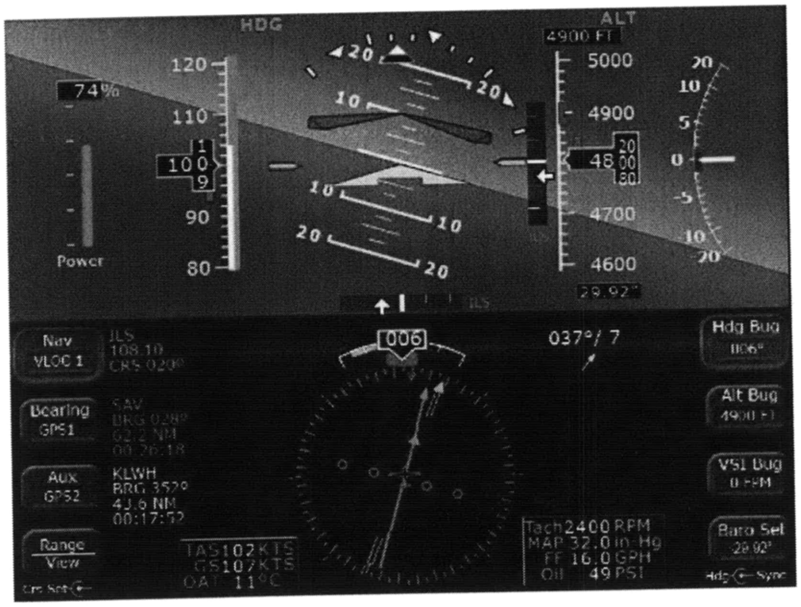

Figure 3 Standard View of the Primary Flight Display

In addition to the obvious graphical features of the PFD, other features are embedded below the graphics, such as supplying navigation information to the traffic and autopilot systems.

A data recorder was added to the PFD after the initial release. One of the goals of the recorder was to provide data for internal use to determine life-cycle reliability and error conditions in returned units. According to an industry publication 35%-60% of avionics returns result in "No Fault Found" or "Could Not Duplicate" resolutions (Avionics Magazine, 2005). A data recording mechanism was developed in an effort to provide better information for our service center to decrease the rate of such problems as those problems can end up becoming multiple returns for the same failure. Those multiple returns cost a significant amount of money both to the company and to the aircraft owner due to Aircraft on Ground (AOG) situations.

General Technical Details

One of the common sources of defects in the PFD has been access and writing to high reliability flash memory. The PFD contains onboard memory storage in the form of embedded flash memory chips. The flash memory chips read/write/access times are significantly slower than system RAM. This slower speed requires special consideration in embedded real-time systems because of the non-negligible impact to thread timing. The PFD's flash memory includes 128 blocks of 256 Kbytes (where 1kByte equals 1024 bytes and 1 byte is 8 binary bits).

When examining a block of erased memory, every bit in the block appears as a '1'. Writing to a block is the equivalent of masking off some of the bits, switching them from '1' to '0'. Once data has been written, the only way to rewrite to the same address is to erase the entire block of 256 Kbytes and then write the new data. The writing of data to an erased address is relatively quick, on the order of 10 microseconds, while a block erase can take up to 1.5 million microseconds. The erasing of a block of flash memory locks up the entire flash memory chip for the duration of the erase, so care must be taken to only perform an erase before the system is held to its runtime schedule, or the erase must be performed by a background process while the foreground processes buffer their data as they wait for flash memory access.

Case 1: Green Screen

Introduction

As part of ongoing product improvements and feature additions, a flight data recorder was added to the software inside the primary flight display (PFD) software. During a subsequent release, additional data files were added to the system configuration. The additional files caused a reboot of this flight-critical system whenever the flight data recorder attempted to access the physical end of the file system.

Technical Discussion

The code for the main processor resides on 32 MB of high reliability flash memory. As previously discussed, the flash memory is addressed by physical blocks, and the chips chosen for this application are broken down into 128 blocks of 256 Kbytes each. Blocks are numbered from 0 to 127, and a block allocation scheme was chosen early on in the design process once the approximate size of difference components was determined. The memory layout is shown in the table below.

Block Number	Description
Block 0	Header Block
Block 1	System Information Block
Block 2	Major fault block
Block 3	Pilot non-volatile data block ID A
Block 4	Pilot non-volatile data block ID B
Blocks 5-19	Application data blocks
Block 20	Base block for FAT table
Blocks 21-127	Texture (graphics) files, configuration data, and log files

Table 1 PFD Flash Memory Layout

From **Table 1** above one can see that Block 20 is reserved for creating a rudimentary file system and blocks 21 through 127 are reserved for "files" such as graphical texture sheets, configuration files, and data logs. The set of texture files that are loaded onto a unit are typically identical from release to release. To simplify the graphics drivers used in the product, all textures are moved from flash memory into video memory and statically stored in video memory so that dynamic loading of data would not be required. The list of textures and the sizes of the textures change only when additional textures are required because of the fixed amount of VRAM dedicated to hold textures. If a new texture is needed, some of the other textures would need their bit depths to be cut, or decals within each texture sheet would be moved to a new texture sheet to optimize the use of the fixed size texture sheets, replacing an empty texture with the new texture file to be fit into VRAM.

To prevent any errant textures from slipping into the product, all textures are check summed, and a master checksum of all texture files is calculated and displayed on a maintenance page to permit auditing of the conformity of all loaded software and data.

The manufacturing process loads all texture files in contiguous blocks starting at Block 21. In addition to the texture files, a configuration file is loaded as part of the manufacturing process that determines whether the system should enable a specific set of features related to autopilot integration. The remaining portion of flash memory is left blank after the manufacturing procedure.

The software inside the PFD records different data to each of three files that reside in the file area of blocks 21 through 127. Because the manufacturing process loads all texture and configuration data before any software is running on the main processor, any files created at run-time by the main processor must be created in the memory remaining after the manufacturing files are loaded. During the first power-up of the full system after manufacturing, the application software attempts to open the three log files used by the data recorder to fulfill the recording requirements. The first two files are each 512 Kbytes (2 blocks), while the third file was initially sized to 1 megabyte (4 blocks).

Later in the development process a high level requirement indicated that 20 hours was the minimum recording time desired. The third file was resized to meet the new high-level requirement. The new number of blocks (84) allocated to the periodic data consumed all available remaining flash memory as calculated in the flash memory analysis.

The flash memory usage analysis relied on the assumption that the exact same files used in the analysis were present in flash memory prior to log file creation. The list of files from the analysis was compared with the list of files loaded on the PFD, and the lists matched exactly. In addition the checksum of the texture files matched the expected checksum, indicating the exact specified textures were properly loaded.

Tests for Credit (TFCs) were performed with conforming hardware and software as reported on the status page. The hardware was loaded by a copy of load media prepared by engineering to be used during TFC. The formal build procedure was run before testing began and after testing completed to ensure the software was properly archived and could be regenerated (see DO-178B, Section 8.3 – Software Conformity Review). After all testing was completed, the formal, complete build procedure was run again for all parts to be included in the shipping product, and again all software matched expected checksums. A new piece of load media was created to become the manufacturing gold master, and the master was released through the configuration management organization.

During manufacturing acceptance testing, the PFD was seen with random displays of an all-green display. That green screen indicates a reboot of the main processor that controls the graphics. A system reboot completes in under a second, so visually catching a reboot requires patient attention. The screen is loaded with a green rectangle early in the boot sequence while texture files are moved from flash memory into VRAM for use by the graphics chip. A system reboot is a very serious issue that must be investigated until a cause is found, even if the event is only witnessed in one unit. Other manufacturing units were loaded from the same load media, and eventually all such units began rebooting randomly.

The PFD has a variety of mechanisms by which it could command a reboot of itself via software or via hardware. All software-commanded reboots log data to internal flash memory (Block 2, above) for diagnostic purposes. None of the units from manufacturing showed any data stored in the major fault block. Testing was performed on the offending units to ensure that the major fault logging was working, and the feature was confirmed to be active and functional. Once software-commanded reboots were ruled out, suspicion moved toward hardware-commanded reboots. The PFD's main processor has a companion supervisor processor that will reboot the main processor if active communication is lost between the two processors. When the timer expires the supervisor triggers a hardware reset by toggling the main processor reset line. To prevent such a reset, the main processor must "ping" the supervisor within the time period of the timer by writing a specific data value to a specific shared memory address that can be read by the supervisor. The main processor sends that ping in a periodic manner such that a ping is transmitted every major run cycle that completes in the time allowed. That internal time is measured by a precision timer interrupt running on the main processor.

One of the more useful pieces of information discovered during the subsequent investigation into the anomalous reboots was that the PFDs used by the engineering group during the TFCs were not afflicted by the same symptoms. As the units used during the TFCs were conformed to be identical to the manufacturing units, some suspicion was cast upon the manufacturing process. The key to understanding the root of the issue occurred when a "wall of shame" was built to run several PFDs side by side on a tabletop to perform extended run-time tests. Two of the PFDs rebooted within seconds of each other, signifying that a timing related issue might be at work. By counting back from the exact time the units were loaded with fresh software and configuration data, it was determined that approximately 19 hours had passed between the initial power-on after first load and the observed reboots on each PFD. Once word reached the developers of a 19-hour period on the reboots, the search for the cause immediately focused on the data recorder. Within a matter of hours the cause had been determined.

The cause of the reboot was an attempt by the main processor to write a data record to an address that was not within the address bounds of the flash memory. **Table 2** below shows the overall PCI memory layout by address of the physical devices.

Device	Address
Flash Memory Base	0xF8000000
PLX 9030 Base	0xFA000000

Table 2 Partial PFD Physical Memory Map

Recall that the flash memory was built as 128 blocks of 256 Kbytes. 128 Blocks x 256 Kbytes/Block x 1024 bytes/Kbytes = 0x2000000 bytes. Starting at the FLASH_BASE address of 0xF8000000, one can see that the PLX9030 PCI bridge memory space was adjacent to the flash memory part at 0xFA000000. Any attempt to write at an address off the far end of the flash memory would result in an attempt to write random data into the PCI bridge device. Writing garbage into the PCI bridge caused the main processor to wait for an appropriate response from the flash memory controller, however the flash memory controller had not received a command and therefore the main processor waited in a loop, failing to ping the supervisor. The supervisor would wait for the timeout period then externally reset the main processor.

The proximal cause can almost always be traced to software because software is the final output of the development process and software is the only part of the development process that actually flies in the airplane. This case is no different.

Certification Discussion

The defect traced to the software was that no software modules performed any checks as to the size of the files created or the addresses of any attempted reads or writes. An engineering analysis provided the maximum size of the data recorder file and the size was implemented as a constant. No low level requirement existed to document the expected behavior in the event of attempted flash reads or writes outside the address space of the part. The original flash driver was developed when only 30-35 flash blocks were in use, and the use of the entire flash device was not foreseen. Because the driver software was certified to DO-178B Level B, structural coverage objectives may have deterred the developer from adding additional branches to the code. Those additional conditions would need to be tested for both true and false cases, requirements would need updating, and tests would need to be written for a case that was not expected during normal or abnormal operation.

The high level requirement (PFD-REQ-14891) states, "The FDR shall be able to store at least 19 hours of flight data". The data logging rates specified in other high level requirements were analyzed to use of 84 blocks of flash memory. The implementation of that 84-block value was hard-coded in a macro, leaving it untraced to any requirement or design information. The logging-related low-level requirements described the timing mechanisms used to ensure periodic logging of data, while the flash driver low-level requirements described the memory layout in blocks with block and offset access. No comment or limitation is documented within the flash driver design or requirements information, and no device bounds checks were included in the source code.

The testing that traces to the high and low level requirements does verify that at least 19 hours of data are recorded, and those steps passed when run during the formal TFC.

As part of the engineering process improvement group, a new procedure was written to formalize the steps previously undertaken to create load media. Prior to this document, a single build master was responsible for creating the formal build of the released load media. A default load media image was stored in our source control system and engineers developed a procedure to be executed alongside the released software build procedures to create the released load media. When TFC completed, the newly drafted load media creation procedure was executed and the output of that procedure was provided to our configuration management organization for archival and distribution. It was this load disc that was provided to manufacturing along with a system traveler containing the checksums of all components to be loaded from that disc. Manufacturing executed the released load procedures on conforming hardware and signed off on the release of the disc as acceptable. Only days later did the problems appear in the manufacturing units.

The subsequent investigation determined the default load media stored in the source control system was never reviewed or compared against previously released load media. The stored media contained the flawed file. The source of the default load media in source control was the released load media from a release that preceded the current by two versions—not the current prototype load media used during engineering test for credit. The current release was not the first to use the entire amount of memory inside the flash part. The logging features were added in the release immediately preceding the current development effort. It was during that immediately preceding development effort that the size of a configuration file was changed from 1 byte to 0 bytes. That change allowed the system to record an additional 14 minutes of flight data by freeing up an additional flash block for use by the recorder. The default version of the load media was created from two releases prior, (without the configuration file change); therefore the configuration file change was not included in the source control tree.

One of the shortcomings of DO-178B and its cross-references is the lack of any guidance about being able to manufacture a production version of the software and hardware together. The development company is free to determine how to best produce the equipment in mass quantities while maintaining compliance with the conformity documentation provided as part of the original approval process. In this case the approved software parts all identically matched their expected checksums, but still the error reached the manufacturing stage.

The specific process defect that caused the offending behavior was the incorrect creation of load media for manufacturing. DO-178 Section 11.4 and our Software Configuration Management Plan (SCMP) focus heavily on the control of source code and executable object code, but fail to mention associated utilities required to load the approved software onto the hardware.

Section 7.2.8 of DO-178B, Software Load Control, only discusses guidance regarding executable object code. The omission of databases, graphical textures, and configuration data are glaring omissions. It's possible that additional guidance in this section regarding the conformity of the load media used in both verification and manufacturing may have prevented this defect from reaching the post-TFC state. Section 8.3, Software Conformity Review, requires that approved/released software be loaded via released instructions, but again, there is no mention of the configuration data and tools associated with the executable object code. A single file with a single character difference (empty vs. a "1" character) caused a block of memory to be allocated for a file that should have only existed in the file allocation table (FAT). That extra data caused the properly sized data recorder file to run off the end of the flash part. When the software attempted to write to that address every 19+ hours, the supervisor processor would reboot the main processor, leaving no stack trace or hint as to the source of the problem. Complex and parameterized systems rely on data to operate properly, and the lack of control on such associated data seems to be an area that current guidance neglects.

Recommendation: *In addition to executable object code, all utilities and data files used to load and configure the system should be under the strictest control and subject to review processes.*

This defect in the load media is not a new problem. A similar mistake was made in the Titan IV B/Centaur TC-14/Milstar-3 Accident. A roll filter constant was mistyped as -0.1992476 instead of -1.992476 when loading the satellite launcher inertial measurement system. The result was a loss of roll control of the satellite launcher and eventual loss of the Milstar satellite (Pavlovich, 1999).

One of our internal process shortcomings was the lack of sufficient review of the drivers and other APIs provided in the logging release. Each of the follow-on development efforts relied heavily on the use of previously developed software (PDS). There is no evidence of in-depth re-review of the use of the linked library that contains the flash driver. Previously the driver had been approved for use in the existing product and worked flawlessly for more than four years. The addition of the logging feature used the same application-programming interface (API) as before, but only a cursory review was performed on the unmodified flash driver. The certification overhead of modifying the flash driver discouraged any changes. Similar to the Ariane 5 rocket failure (Lions, 1996), the developers believed that changing software that previously worked well would be unwise due to potential side effects. The thought process was that inputs to the driver were statically determined and could not be changed once the system initialized, therefore if the system initialized properly, there was no danger of accessing an out-of-bounds memory location.

Recommendation: *A call tree analysis should be performed for substantive use of PDS to ensure usage is consistent.*

Internal process checklists have subsequently been updated to ensure all formal builds are created on the conformed build machine using controlled instructions. The build procedures now require creation of not only the executable object code per DO-178B, but also the manufacturing load media for every release candidate build. TFCs can only be performed using conformed hardware loaded from conformed load media as output from the formal build process. Making any changes to the location of the source to be released as part of the configuration management process can be complex. To combat that complexity, all branching of source code, object code, and load media must occur prior to formal, for-credit testing and manufacturing acceptance to ensure the artifacts used in engineering match the artifacts provided to manufacturing.

Summary

The output of the engineering development process is a completed product as well as the hardware and software component specifications and procedures required to manufacture the product. DO-178B discusses control of executable code, omitting other digital data stored alongside the executable code such as databases, graphical textures, and configuration files. Internal processes have been updated to address this deficiency in the guidance material. The defect reached the pre-production product due to software fragility, a lack of sufficient review of PDS in a prior release, and new processes introduced in the current release.

Case 2: Persisted Settings Fail to Persist

Introduction

The PFD keeps track of pilot settings such as the selected heading, altitude, and vertical speed targets in addition to the altimeter barometric correction. The goal is to provide consistent settings across power cycles of the system should the system be rebooted in flight or powered off during a refueling stop. After a specific number of different values and user interactions, the recording mechanism would stop working until reset by the factory.

Technical Discussion

To ensure the pilot settings are retained across power cycles, the software developer chose a "ping-pong" design. By alternating between blocks of flash memory, one block will always be erased and ready to accept the latest pilot settings. Another constraint of embedded flash memory is its limited number of write-cycles. These parts can only be erased and rewritten 10,000 times before reliability may decline. To minimize the number of erases and writes to flash memory, more than one version of the settings is stored in a single flash block. The software picks the most recent settings by starting at the end of the current block of flash memory and decrementing until a non-empty address is found, indicating the last written entry in the current block. If no entries are found, the block is marked as blank.

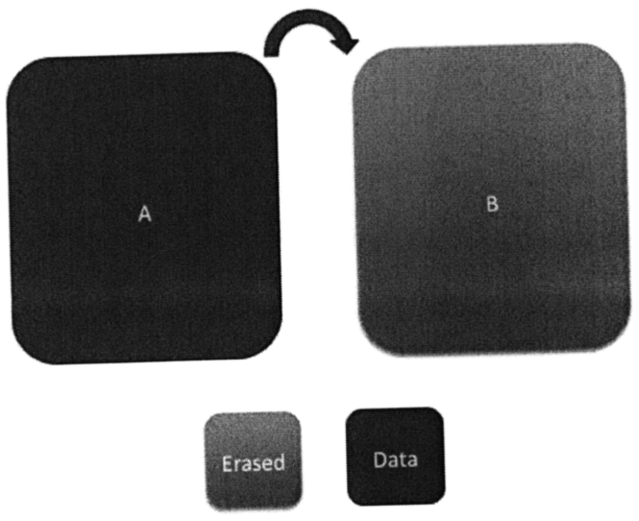

Figure 4 Ping-pong Buffers with Block A Full and Block B Empty

On startup the PFD selects the appropriate block to use as the active block. At the same time the current block is selected, the opposite block is checked to ensure it is blank and ready for recording once the current block is filled. When a PFD is delivered from the factory to the aircraft, both Block A and Block B are erased. Every time the pilot changes a setting such as the barometer correction or selected heading, the PFD records the new value (at 1 Hz) to ensure the most recent data is available in case power is removed. The data is initially recorded to Block A until Block A cannot hold another complete pilot settings record. At that point the records are written to Block B and a command is sent to the memory controller to erase Block A in the background. Eventually Block B fills with additional records after a few months of flight and the records are written to Block A. The code that erases a block in the background does not wait for the erase to complete, preventing a check of a return code. Because no value is returned from the function, there is no way to determine if the erase command worked.

Upon transition from Block B to Block A, the non-blocking erase call is sent to erase Block B. Using production hardware it was determined that the erase call never succeeds when called in the context of the pilot settings recording (See **Figure 4** above).

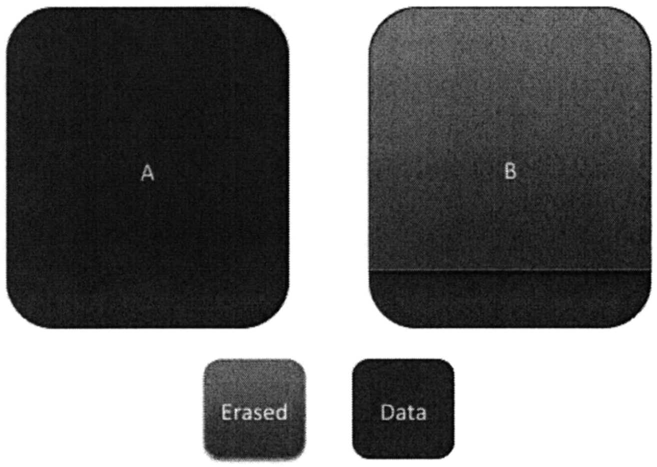

Figure 5 Ping-pong Buffers with Block A Full and Block B Partially Full

The timing of other calls to the memory controller inhibits that particular erase command because the call to erase is followed by a call to protect the flash memory from unauthorized writing. That protection call is implemented by removing power from the program enable line

(VPEN). The effect is removing power to the circuitry inside the flash part that sets the bits back to all "1"s (erased). The developer appears to have understood that the on-the-fly command to erase a block of flash may not succeed as similar logic is applied at startup. In contrast to the run-time call to the non-blocking erase call, the startup code uses the erase command that waits for the erase to complete. The startup code can use the blocking erase command because the system has not started its normal task scheduling and started its hardware watchdog that reboots the system if all tasks are not reporting nominal status.

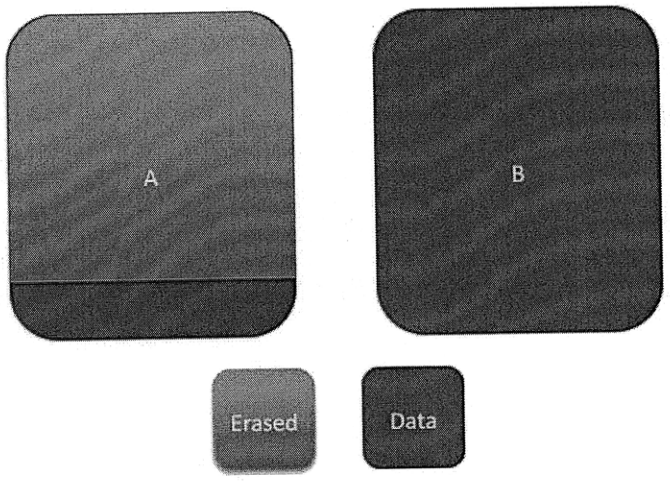

Figure 6 Ping-pong Buffer with Block A Partially Full and Block B Full

Despite the defensive programming employed to ensure the appropriate blocks are erased at startup and run-time, the pilot settings would not be retained after power cycling due to a second defect. The startup code that chooses the active block always chose Block B as the active block because it was not empty on startup. If the erase commands worked as intended, one block would always be blank on startup. The software initially looked at Block B, an arbitrary decision, saw the block was not empty, and selected it as the active block. At the same time the blocking erase call was sent to ensure Block A was erased to handle the transition when Block B filled. Upon the first attempt to record the pilot settings, the software properly sees that Block B is full and transitions to Block A, sending the erase command that always fails to erase Block

B. Records are appropriately written to Block A until the system is shut down (See **Figure 6** above).

The high-level feature requires that data be restored from the flash memory to the system RAM upon startup. The startup block selection logic determines Block B to be non-empty, selects Block B as the active block, and performs the complete erase of Block A, returning the system to the state shown below (See **Figure 7** below). The last value stored in Block B is restored to system RAM as the last saved settings, throwing out the correct data when Block A is erased. This sequence repeats itself on each power cycle after 52,428 pilot settings have been recorded. Under average use this value amounted to approximately three years of storage.

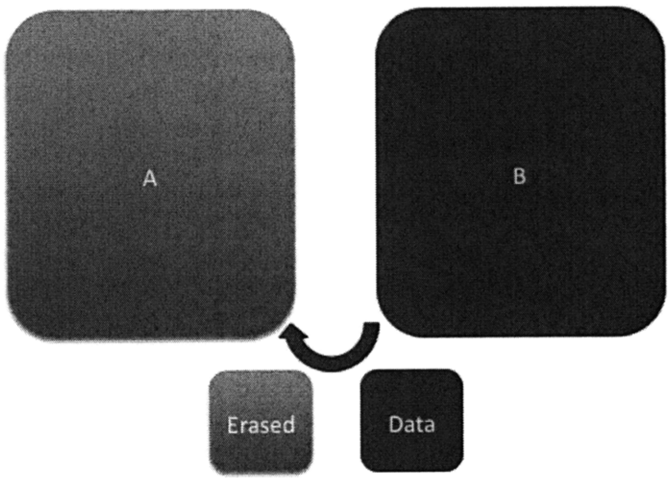

Figure 7 Ping-pong Buffer with Block A Empty and Block B Full

The associated test involves using a hardware debugger on the target to set a breakpoint. The test sets up the system to have a full Block B, and then verifies that data is appropriately written to Block A to ensure proper transition between the ping-pong blocks. The unit is power cycled to ensure that the last used data appears. Unfortunately for the test, a debugger adjusts the timing of the calls to the memory controller such that the erase command is accepted only because the call to protect the flash memory is delayed by the single stepping in the debugger.

Certification Discussion

The cause of this system deficiency was certainly a software defect, but where did the process fail? There were requirements that specified the desired behavior when Block B was full. There was a very detailed and specific test to ensure proper switching from Block B to Block A, and that test traced to the applicable requirement. The test demonstrated structural code coverage. The requirements, code, tests, traces, and coverage data were reviewed with independence, yet the bug was not discovered until two years after the initial product certification.

I maintain that the requirements were adequately specific and that the traceability between system, software high-level requirements, software low-level requirements, source code, and verification procedures was adequate. The process defect was a failure of the tests to verify the persisted data feature in a fully integrated environment. The use of a debugger to test is acceptable to demonstrate detailed design, data, and control flow within an application, but all high level features must also be tested in an integrated, conforming environment.

DO178B Section 6.4, Software Testing Process, states:

> Note: If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing or software integration testing and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested.
>
> a. Test cases should be based primarily on the software requirements
> b. Test cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.
> c. Software requirements coverage analysis should determine what software requirements were not tested.
> d. Structural coverage analysis should determine what software structures were not exercised.

The high-level requirements only specify that the pilot settings be persisted over power cycles. The use of a ping-pong scheme to record the data was an architectural decision documented in design artifacts. The implementation could have used 3+ blocks, put one entry per block and constantly erased the opposite block, or implemented any of a variety of other data storage schemes. Because the choice of a ping-pong scheme was architectural, the requirements that explicitly describe the behavior were low-level requirements and therefore specifically tested using target tests. The high level requirements were met, and the high level tests verified the desired behavior of persisted data. Only the low-level tests would have known about any potential block changes.

Examining the low-level tests, one can see that in fact the tests did verify proper transition between blocks. The critical detail that was missed was the timing differences caused by use of the debugger on those particular statements. Had a test been run with breakpoints outside of the function call to store the settings, it would have been apparent that data was properly written to Block A, but Block B was not being erased. Of course the test would have to ask the tester to verify Block B was erased at the exact right time.

PFD software requirement PFD-REQ-9168 specifies "The system controller during a write shall erase the alternate block if the write pointer is set to the base address of the current block." There was a low level test that linked to that low level requirement; however the test also linked to three other requirements. The expected result of the test step where the requirement is traced contains a high level result, namely that the barometric correction setting showing on the PFD after a power cycle matched the last recorded value upon transition. There was no check of the exact low-level response demanded by one of the traced requirements presumably because the high level requirement was met. A testing standards update to demand traceability of one expected result to one requirement may have prevented such a defect from reaching the final product as the existing traceability existed but failed to provide sufficient precision to determine if any combination of passes and failures could generate the same externally observable result.

Recommendation: *Ensure that no more than one requirement trace to a single verification step to ensure adequate traceability between the requirements and tests.*

A second low-level requirement specifies that the initialization "shall clear the alternate block (one of the two blocks that is not currently active) if it is found to contain data." There is a specific test that toggles the active block to the opposite block and verifies that the original block is properly erased. Once again this is a debugger test that could have been written to use a breakpoint at a different location in the code. As the initialization code calls the blocking erase, there is no risk of power being removed from the VPEN line and this test is sufficient to test the associated requirement.

A more appropriate test for the run-time block erase may have been to craft data records and load them onto the hardware to simulate a PFD that had been used for an extended period of time. While more difficult than just setting breakpoints and forcing the code down different paths, the test would be more representative of true operational characteristics and behavior than artificially forcing a path for expediency. Debugger tests should be a technique of last resort to demonstrate coverage of difficult-to-reach paths due to defensive programming or delicate timing tolerances. In this case execution of the code paths that failed to meet the requirement was an expected behavior during normal operation of the system, and that behavior is testable at the system level.

It is unreasonable to expect a high level requirement to specify an exact minimum number of consecutive records to be successfully stored. The high level requirements state that the data must always be stored while rightfully ignoring any implementation details about potential block rollover issues. A simple analysis shows the exact number of records that can be recorded in a single block, but no such analysis was documented in the low level requirements and design information to highlight the block rollover issues that were understood by the original developer of the low level requirements. The lack of detailed design information in the software design document failed to specify or identify the intended number of records per block and the total recording capacity of the design.

Compounding this problem in the software was the existence of two bugs within the same feature. Recall that the first software bug was the removal of programming power prior to allowing the erase command to complete. If that command succeeds, the feature works as expected. The second bug was the incorrect selection of Block B as the active block when both Block A and Block B had data. The programmer considered the case when neither block had data (factory fresh), but neglected the case where both blocks had data. The neglect of this case likely resulted from an understanding that the intended design did not allow for two blocks to have data at the same time as one block is erased upon transition to the other block. Ideally the low-level unit tests would have tested different combinations of data, verifying proper block selection and erasure at an application level instead of within the driver.

Summary

Both the tests and the source code were reviewed with independence, and both were updated to satisfy the expectations of the reviewer, yet this defect was missed. While the defect can clearly be traced to an errant removal of programming power in software, other process defects allowed the software defect to reach the approved product. One specific omission here was a lack of robustness to handle the block selection logic when both blocks had data. This example only had 4 possible states of the blocks on startup, but the software failed to properly handle one of them. Software defects can survive our best attempts to discover and characterize them, even with independent review of high and low level requirements, tests, and traceability.

Case 3: Data Recorder Gaps

Introduction

A flight data recorder was added to the PFD product as part of a follow-on release to the initial release of the product. After an analysis of processor usage by the data logging application as part of the previously mentioned "Green Screen" issue, several recommendations were made to enhance the data recorder feature to make it less intrusive to the operational system. One of the changes was to write to flash memory at a rate independent of the rate at which data was being logged. These changes were ineffective and caused data recorder dropouts of approximately five seconds whenever the buffer filled before data was flushed to flash memory.

Technical Discussion

Resources are limited in any software development, and resources are even more constrained in an embedded application. The early PFD releases constrained the recording of data to flash memory to a single thread in the operating system. Flash access by block was the only method for reading or writing memory, and the only data updated on the fly was pilot selectable "bug" settings. The initial release provided a primitive file system was established for use in flight mode. With the addition of the more fully featured data recorder and multiple threads needing flash memory writable access, a new task was created to provide a semaphore to restrict access to a single task at any given time. That task will be referred to throughout the remainder of this section as the "flash manager".

The flash manager operates by reading requests from other tasks via a synchronous shared memory scheme. The flash manager publishes the flash state (locked or unlocked) as well as the task authorized to write to flash after each run cycle. If flash access is requested by a task and the lock is available, the flash manager updates the published state to "locked" as well as an identifier unique to the locking task. When more than one task requests flash access, the tasks are placed in a "First In, First Out" (FIFO) queue and each task is provided the exclusive access to flash in the order the requests were received by the flash manager.

Now that the flash manager scheme has been established, it is time to investigate the scheduling mechanism by which the flash manager runs alongside the tasks that require a lock on the flash device. The task scheduler uses a Rate Monotonic Architecture (RMA) that schedules tasks synchronously. The ordering of task execution is Output->Input->Run. Each of the task function points is called at the rate specified in the scheduler configuration. If two tasks run at different rates, then the output of the higher rate task will not be available to the lower rate task until the lower rate task is scheduled again. The details of the RMA are listed in Appendix B. Such a scheme ensures that a lower-priority/preempted task is provided with shared memory data updates while in the middle of execution. One of the disadvantages of such a scheme is that additional latency is introduced between the two tasks. It is this additional latency that shows the root of the software defect.

Writing to flash memory is significantly slower than writing to RAM; therefore the software must use caution when writing to flash to ensure all scheduling deadlines are met. To achieve this goal, only two flight data records are written to flash in any given run cycle of the data recorder. In addition other event data or system state information can preempt the writing of data from the RAM buffer allocated to the flight data. The periodic flight data is the lowest priority data in the system by design to ensure that all event log data is captured. Recall that the original purpose of the data recorder was to provide the service center with event data to assist in diagnosing units returned from the field.

The flash manager runs at a rate of 50 Hz, equal to the rate of the fastest task that requests the flash lock. The data recorder runs at a rate of 25 Hz or once for every two executions of the flash manager task. Because the two tasks run at different speeds, the previously mentioned latency problem affects the communication between the two tasks in the worst-case scenario. Aside from the potential for other tasks to preempt the recorder and claim the flash lock, sufficient spare timing and memory must be allocated to the data recorder to ensure no data is lost.

The buffer size is statically allocated at exactly 2048 bytes and each data recorder entry is a fixed size of 44 bytes, yielding a buffer capacity of 46.5 entries. Based on the implementation of the simple buffering scheme, adding an entry that would not completely fit inside the buffer causes the entire buffer to be erased. That implementation details results in a buffer that holds exactly 46 complete entries. Data is input to the RAM buffer by both periodic and event-driven mechanisms. The event-driven data results from pilot action and can be accounted for in the surplus budget allocated to ensuring the RAM buffer never fills. Examining the periodic mechanisms the following data is recorded:

Data Type	Rate
AHRS Data	5 Hz
Air Data	1 Hz
Nav Display	0.25 Hz
Nav Details	0.25 Hz
Position & Time	0.25 Hz
Flight Director Data	1 Hz
Engine Data	.16 Hz
Total	7.9 entries/second

Table 3 Data Logger Rates

Writing an entry to flash on the slowest flash part ever fielded was empirically determined to require 1.4 milliseconds. As previously stated only two entries are allowed to be written in any single task execution to ensure minimal impact to the timing of other tasks. The above analysis demonstrates the data recorder should write out two entries every 200 milliseconds for a total of ten entries written per second while 7.9 entries are loaded into the buffer each second, on average. Writing ten entries per second is the equivalent of 25% spare throughput to accommodate event-driven data entries and flash access delays from other tasks and block erasures. Dividing 46 entries by ten entries per second results in an effective buffer length of 4.6 seconds of data. The logs from the field indicated data dropouts of approximately 5 seconds, which correlates to the analyzed RAM buffer size.

The data recorder was analyzed to require access to flash every 5th cycle of the 25 Hz task for an effective write-rate of 5 Hz. Per the analysis the software was written and tested to attempt a write to flash at 5 Hz. When the attempt to write to flash is made, the lock is requested. An entire run cycle is wasted waiting for the subsequent Output call as each frame calls Output->Input->Run, then waits for the next schedule opening before calling Output again. On the 6th execution of the 25 Hz frame, the Output call is executed and data recorder writes its request to shared memory..Because the flash manager runs at an even multiple of the data recorder, the 50 Hz tasks are executed every time the 25 Hz tasks are executed, and the flash manager calls Input immediately after the outputs calls complete. In the best case the flash manager provides the lock to the data recorder in the same (6th) run of the data recorder and indicates the lock has been granted prior to the 7th run of the data recorder.

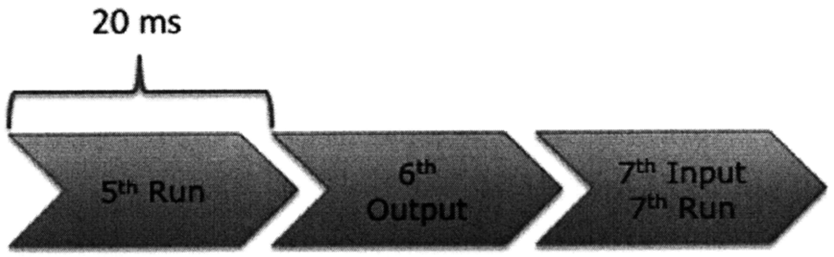

Figure 8 Data Recorder Timing

5th Run of the data recorder - Lock request placed in local shared memory
6th Output of the data recorder - Lock request is published to global shared memory
7th Input of the data recorder - Lock grant is read by data recorder
7th Run of the data recorder - Two records are written to flash

Due to the overhead from the RMA and the flash manager semaphore, attempting to write to flash every 5th run of the data recorder results in actually writing out data every 7th run. The data throughput analysis indicated that the required flash write rate was 5 Hz or every 5th run of the recorder, but the effective rate of the implemented software was 3.57 Hz, despite variable names and the intent to write to flash at 5 Hz to stay ahead of the incoming data. Writing out data 3.57 Hz results in an effective write rate of 7.14 entries per second, less than the rate of periodic data being inserted into the buffer. Based solely on the periodic data the buffer was falling behind on its write by 0.78 entries per second.

7.92 entries added/sec - 7.14 entries removed/sec = 0.78 entries/sec

With a maximum buffer size of 46 complete entries, one can calculate the period of the data loss:

*1 second / 0.78 entries * 46 entries = 58.97 seconds to fill RAM buffer*

With a buffer capacity of 46 complete entries, the data from the buffer would be flushed every 59 seconds. Due to the proximity to 60 seconds, several investigations attempted to locate all events and tasks that were based on a one-minute timer, but those investigations were misguided.

The architectural overhead from the RMA and the flash manager resulted in two extra run cycles of the data recorder before data was truly flushed from the RAM buffer to flash memory. The lack of accounting for the architectural overhead of synchronous task communications resulted in a periodic and deterministic loss of data for a particular release of the flight data recorder.

Certification Discussion

Two objects in the requirements tracking system discuss the specifics of buffering flight data prior to writing to flash memory. ACR-8240-4321, a design object, states:

> To minimize the time the FDR server is holding on to the permission to write to flash (provided by the flash manager) and [to] allow other tasks to have ready access, buffer all the entries for say 1 second before requesting permission and writing. Don't buffer for too long or we will spend too much time in one run cycle writing to the flash.

The associated low-level software requirement (ACR-8240-4320) that details the specifics of the buffering reads, "The software shall buffer its FDR entry data for 200 milliseconds before writing to flash". The source of the initial estimate as to the frequency of buffering the data was based on current production flash memory speeds. Later the requirement was revised down to 200 milliseconds to accommodate the timing tolerances of slower flash memory originally delivered inside the product.

An important distinction exists between buffering data and writing the data at a given rate. The requirements specify that all data should be buffered for 200 milliseconds, which is slightly different than indicating the flight data shall be written out every 200 milliseconds. The analysis of data throughput indicated that the data must be written to flash every 200 milliseconds to ensure the PFD had sufficient timing margin to prevent the RAM buffer from filling faster than the server could empty the buffer to flash memory. The translation of the requirement from the data analysis stage to the requirements development stage failed to capture the true intent of the requirement. In addition, the requirement provided a specific timing value of 200 milliseconds without any tolerance on the measurement.

The low-level tests verified the low-level requirements regarding data recorder buffer behavior. In the event the RAM buffer filled completely the requirements dictate that the buffer be completely flushed to ensure the recorder does not fall too far behind the live flight information. This purge scheme ensures that the critical moments before or after an adverse event are not lost because the data was captured but not written to disk before power was interrupted. The associated requirement of the LogProxy class is ACR-8240-4208, "The software shall clear the local FIFO buffer in case of overflow." The test involves using a debugger to artificially fill the LogProxy object's buffer and verify that the FIFO buffer is reset to an empty state, resetting the data pointer to the beginning of the buffer.

The low-level requirements verification of requirement ACR-8240-4320 (above) was achieved by testing and analysis. The timing of the buffering was performed using a utility class that provides a decrementing counter feature. An instance of that class takes two parameters across the entire public interface, the timer period and its enabled/disabled state. That utility class has its own set of requirements and tests that verify its detailed functionality. The data recorder was verified to set the timer for the equivalent of 200 milliseconds (5 run cycles at 25Hz), and the buffer was verified to contain the data from the previous cycles.

The design object that appears alongside the governing software requirement discusses the rationale for smoothing out the writes of data to flash, but the design information fails to identify the source of the 200-millisecond value. The design object even mentions the architectural requirement of requesting access to lock flash memory via the flash manager service, but nowhere is the overhead of using the locking system mentioned when discussing the timing requirement.

Reviewing the flash manager requirements and design information does not provide any additional insight into the timing and latency characteristics that result from using the flash manager locking mechanism. The design information discusses the allocation of functionality within the flash manager class across different functions required by a Task interface. The unique features, memory synchronization, and prioritization are discussed in detail along with verifiable software requirements, but nowhere is there a mention of the impact of using the locking mechanism.

The proximal cause of the data dropouts was the insufficient rate at which the RAM buffer was emptied to flash. The achieved rate was insufficient despite the intent to write to flash at an appropriate rate due to architectural latency spread across several modules. The latency introduced by use of the decrementing counter, FIFO buffer, flash manager, and RMA is actually an emergent property of the system that can be difficult to document with atomic requirements as required by DO-178B section 5.2.2.

The closest DO-178B comes to addressing the issue at the root of this problem appears in section 5.2.2, Software Design Process Activities:

> The software design process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Guidance for this process includes:
>
> a. Low-level requirements and software architecture developed during the software design process should conform to the Software Design Standards and be traceable, verifiable, and consistent.
> b. Derived requirements should be defined and analyzed to ensure that the higher-level requirements are not compromised.
> c. Software design process activities could introduce possible modes of failure into the software or, conversely, preclude others. The use of partitioning or other architectural means in the software design may alter the software level assignment for some components of the software. In such cases, additional data should be defined as derived requirements and provided to the system safety assessment process.
> d. Control flow and data flow should be monitored when safety-related requirements dictate, for example, watchdog timers, reasonableness-checks, and cross-channel comparisons.
> e. Responses to failure conditions should be consistent with the safety-related requirements.
> f. Inadequate or incorrect inputs detected during the software design process should be provided to either the system life cycle process, the software requirements process, or the software planning process as feedback for clarification or correction.

Sub-paragraph (d) discusses control and data flow, but only recommends monitoring such flow when safety-related requirements dictate. The data recorder and associated classes under examination here are not safety-related but maintenance-related. The pilot would notice no reduction in safety or performance of the system if the flight data recorder and other associated classes failed to provide their intended functionality.

Recommendation: *Control flow should be monitored for all functions, not just safety-related functions.*

Recommendation: *Every requirement, (both high- and low-level), that details timing should have a tolerance as well as detailed test steps to confirm the result of the software meets the both the high- and low-level requirements.*

Summary

Despite rigorous analysis and requirements indicating the required rate of flushing the RAM buffer to persistent memory, a software defect escaped detection during the software certification process and reached a fielded product. The failure to account for architectural overhead and sufficiently detailed low-level testing permitted the defect to go undetected. The requirements, tests, and code appeared consistent, but software architecture overruled the intent of the requirements. The feature only changed slightly from the previously developed software baseline, but the change was drastic enough as to cause the buffer to eventually fill and to cause the buffered data to be lost on a periodic basis.

Case 4: GPS Approach

Introduction

The PFD integrates to external GPS navigators to provide GPS-based guidance information such as course deviation. One of the specific navigator vendors recently released an upgraded unit to take advantage of additional GPS signals. The update changed some of the ranges of data types and uncovered an instance of defensive coding capable of providing Hazardously Misleading Information (HMI) to a pilot by indicating the aircraft was centered on final approach when no valid guidance information was actually calculated.

Technical Discussion

The ARINC 429 standard is a digital data bus specification that describes the physical and electrical formats for the bus as well as the data protocol. The basic protocol used in the ARINC 429 specification is a series of 32-bit words, each with a specific meaning. There are other types of more complex data represented using a series of words to make up a single record such as a waypoint in a flight plan, but the specific issue under investigation was the use of a single word.

A 429 word consists of a label (bits 1-8), representing a unique data identifier such as "Lateral Scale Factor", Source/Destination Identifiers (bits 9-10, also known as SDI), data (bits 11-29), a Sign/Status Matrix (bits 30-31, also known as SSM, indicating validity or sign information), and a parity bit (bit 32, to ensure the data was received properly). Each 429 label has a published specification as to the scaling (value per bit), sign convention, and SSM interpretation.

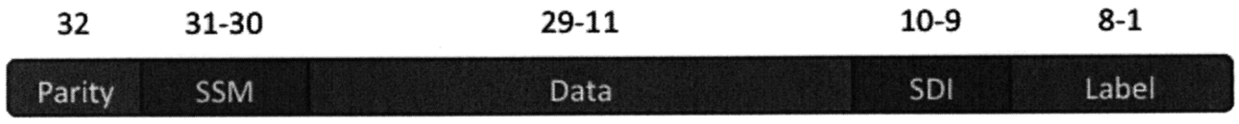

Figure 9 Breakdown of an ARINC 429 Word

The PFD interfaces with a variety of avionics to provide critical navigation information such as bearings to VHF stations and deviation from a flight plan measured by GPS. In the last 10 years a new GPS-based navigation technique has become available. The new standard is called Wide Area Augmentation System or WAAS. WAAS is a system of satellites, ground stations, and in-aircraft receivers that achieve greater accuracy than current GPS receivers. The use of WAAS allows GPS-only precision approaches into specific airports that have published WAAS approaches. Prior to the deployment of WAAS, aircraft were required to fly instrument approaches using VHF navigation equipment specific to the airport and runway. Now with the recent deployment of WAAS, any airport can have an instrument approach if the appropriate surveying is completed and published.

When flying an aircraft on an instrument approach, the PFD displays a pair of scales and pointers to indicate the aircraft's position relative to the published flight path for the approach. On analog navigation equipment the indicators for position were actually needles on hinges, and for that reason even modern glass displays have pointers referred to as "needles". The PFD displays both vertical and horizontal deviation indicators, known as the VDI and HDI, along the right and bottom of the attitude direction indication (ADI).

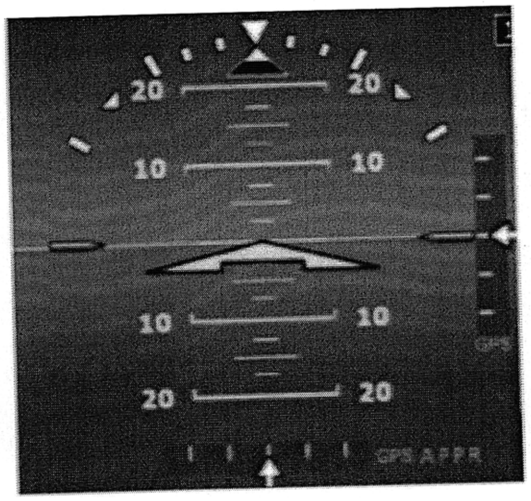

Figure 10 HDI and VDI Displayed with WAAS Data

The VDI and HDI needles receive their deviation data from external avionics sources such as a VHF radio receiver. With the commissioning of the WAAS system, aircraft could potentially display instrument approach deviation data from a GPS receiver in addition to the more traditional VHF radio receiver. Prior to the WAAS system, even non-precision GPS receivers could provide deviation data to external devices. The data was not high-precision approach data, but the data could provide course deviation information for display on a moving map or other system.

The course deviation display is calculated from two pieces of information, each transmitted over ARINC 429 from a GPS receiver. Label 326 provides the Lateral Scale Factor, while Label 116 provides the Cross-Track Error. Those two pieces of information are combined into a percentage deviation by taking the decoded Label 116 value and dividing it by the Label 326 value. The VDI and HDI display scales are percentage-based such that the result of the division of Label 116 by Label 326 results in the percentage of full-scale deflection of the needle.

When performing numerical computations in software, certain mathematical constructs require special handling to prevent errors from using nonsensical data. Examples of such constructs include taking the square root of a potentially negative number or dividing by a value that could potentially be a zero. The calculation of the needle deflection requires the use of division; therefore a divide-by-zero check (DO-178B Section 4.5, Note) would be appropriate. The original PFD certification included such a check on the division, ensuring the divisor was greater than a certain value, in this case 0.1 nautical miles (nm). It had been determined that the GPS receiver to which the PFD was connected in an aircraft installation never reported a lateral scale factor less than 0.3 nm, and therefore a lower bound of 0.1 nm seemed to be reasonable.

With the commissioning of the WAAS system and the installation of new WAAS-capable receivers, the accuracy and precision of the GPS solution and deviation data increased. As part of the initial validation process regarding the installation of WAAS capable GPS receivers, an experimental test aircraft received the new antennae and receivers. Those receivers were connected to the PFD, and WAAS approaches were flown to determine the compatibility of the PFD with the new receivers. It was during these initial flights that the WAAS GPS receivers were determined to be incompatible with any PFD software version due to the aforementioned divide by zero protection.

The WAAS-capable receivers transmitted their increased precision data by using a range of Label 326 that had not previously been possible with earlier technology. The new receivers transmitted a minimum lateral scale factor of 0.1 nm, the same value used to protect against a divide by zero while the previous receivers transmitted a minimum lateral scale factor of 0.3 nm. That difference in and of itself would not be reason enough to declare the new receiver incompatible. The exact mechanism by which the divide-by-zero was protected also had undesirable effects. The code checked that the received lateral scale factor was greater than 0.1 nm before computing a percentage of deflection. If the lateral scale factor was not greater than 0.1 nm, the deviation value was set to 0.

Normally this might seem like an acceptable solution when the divisor could not be determined. Setting the value to the middle of the range could have been acceptable. The main problem with the as-implemented divisor handling was setting the deviation value to 0% and failing to mark the data as invalid. The result was the centering of the HDI needle during one of the most critical phases of flight—final approach. The needle would show HMI by indicating the aircraft is perfectly lined up with the approach flight path when in reality the indicator is locked on the centered position with no indication that the required division is not being performed.

Certification Discussion

The only mention of defensive coding in the entire DO-178B document is a note in Section 4.5, Software Test Environment:

> Note: In developing standards, consideration can be given to previous experience. Constraints and rules on development, design, and coding methods

> can be included to control complexity. Defensive programming practices may be considered to improve robustness.

In fact the use of defensive coding is encouraged by the applicable corporate design standards, specifically the standards specified in DO-178B Section 11.8, Software Code Standards. Despite the structural coverage implications, all potential divide-by-zero cases must be trapped and checked against prior to performing the division to constrain the quotient to a reasonable value instead of Not-A-Number (NAN) or special infinity values.

Section 6.3.2 Reviews and Analyses of the Low-Level Requirements, subparagraph g of DO-178B states:

> Algorithm Aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

The applicable requirement (PFD-REQ-3610) provides a table of tolerances that trace directly to the TSO C146. Specifically, the requirement states FSD shall be 5.0 nm for Enroute Guidance, 1.0 nm for Terminal Guidance, and 0.3 nm for Non-Precision Approach Guidance. The regulations, requirements, and code comments all indicate that 0.3 nm is the expected minimum FSD for GPS guidance. In ARINC 429 terms, that 0.3 nm defines the lowest expected value of label 116G. Of course the label specification provides numerical support for less than 0.3 nm as shown by a least significant bit (LSB) value of .00390625 nm (GAMA ARINC 429 subset, 11/15/2002, 4th Edition, page 11). The LSBs cannot simply be masked off because they are required to achieve the necessary precision of the instrument. A regulatory value of 0.3 nm is expressed as 0x4D in the data bits of label 116G. The "D" indicates that three of the lowest 4 bits are required to express 0.3 nm using the bit values provided by the standard. The specification does include additional resolution bits, but those bits are masked off as unnecessary at this time. In addition, no GPS navigator that integrates with the PFD currently uses those reserved additional resolution bits.

The test requires running a simulator to set the FSD to TSO specifications and ensure that the values displayed on the screen match the deflection values set in the simulator. Values were chosen before, on, and after critical points to ensure proper transition behavior. In addition the lateral deviation was held constant and the FSD label was adjusted, verifying proper behavior when both the dividend and divisor are adjusted within levels specified in the requirements and TSOs. The tests completely test the requirements, but do not test behaviors that are not specified.

The protection against division by zero typically would not warrant its own high level requirement because the check is a protection against input data over which the system has no control in an operational setting. As part of this investigation, additional requirements were drafted to document the desired behavior for FSD values of less than or equal to 0.0 nm to ensure full specification of behavior across the entire range of possible inputs, whether or not the inputs are sensible.

Recommendation: *All behavior directly traceable to external interfaces should have requirements that govern the full range of possible input behaviors. Such specificity can limit requirements errors of omission by guaranteeing full specification of such parameters.*

One of my major criticisms of these types of development guidelines is the lack of testability of unspecified behaviors. It is not feasible to document every feature or behavior the system shall not perform. Of the three verification options presented by DO-178B in Section 6.2, Software Verification Process Activities, only reviews can catch this type of error. Structural coverage is one technique to ensure that all code is required and documented, but structural coverage is not enough to guarantee undocumented features do not exist in the final product. Independent source code reviews also assist in ensuring that all code trace to requirements. Writing a requirement for each conditional expression in a system might be possible for smaller projects, but as code complexity increases, it may not be possible to accurately review, trace, and test 400,000 requirements for 400,000 conditions.

Often common designs such as FIFO queues or circular buffers are used. These common design objects can be implemented in a variety of ways, each way satisfying the software requirement to buffer data. I contend that writing a unique requirement for all the possible cases of inserting and removing data from a circular buffer is not necessary as long as the black-box behavior of the module is well defined.

This example defect demonstrates both the beneficial and the detrimental aspects of integration testing. The initial software development used flight test, data logging, and regulatory analysis to determine appropriate ranges of input data. All three sources of information demonstrated that the lowest value reported by the GPS navigation equipment was 0.3 nm, and requirements were written to reflect the nominal behavior.

During software development a programmer saw the potential for a divide-by-zero and trapped the error in a conditional check to ensure the system did not reboot if such input data were injected into the system. Internal process documents require handling potential divide-by-zero conditions to ensure continuous and controlled operation of systems. Because the system was only tested to the operational ranges of the specific sensor to which the PFD connected, the defect in the handling of the HDI information remained latent. The programmer ensured the system did not reboot, and the behavior did not violate any traced requirement. The software code review, performed by another programmer, failed to flag the conditional as a potential error. The structural coverage testing demonstrated coverage by running a test step designed to ensure system robustness, but the test did not verify any specific functionality. Initial integration testing with target equipment guided the testing towards ensuring proper operation instead of guiding the testing towards trying to break the system.

This example also highlights the benefits of integration and flight-testing. When the new GPS navigator was released to the public for installation in a variety of aircraft, an OEM performed integration flight tests to determine the compatibility of the new GPS navigator with existing equipment. When it was determined that the current production PFD could not display appropriate deviation data due to the ARINC 429 label 116G range change, field installations and approvals were prohibited until such time as the PFD was updated to not display HMI when connected to the new avionics. A minor software update was issued to address this single issue, and a follow-on major release provided full integration for the new navigator features and ranges. The early integration and flight test alerted us to a problem with our software, requirements, and test. That early alert prevented field installations that might have decreased the safety and performance of the pilot and aircraft.

Summary

The source of the defect can clearly be traced to the specific software source code lines. Those lines were written to address a specific DO-178B note and internal company standards regarding defensive programming practices. It seems that even defensive code can generate inappropriate results when missing requirements. The aircraft certification process at the integration site discovered this incompatibility during preliminary integration testing, and the defects were resolved for the field in a minor release. The full integration followed in the next scheduled release.

DO-178B is a process that can be followed to receive software approval; however certification is an aircraft level achievement. Only when all the individual subsystems, both software and hardware, have been approved to work together in concert can an aircraft be certified. DO-178's discussion of integration testing focuses on integration of the software onto target hardware, not integration of the combined software/hardware system into the aircraft. Less stringent integration testing at the aircraft level may have inadvertently allowed a minor software defect to become a critical system defect.

***Recommendation:** Integration testing should include integration with external hardware and software beyond the product boundary to ensure requirements and interface specification gaps do not cause safety-critical defects.*

Case 5: ARINC Transmission Rates

Introduction

Requirements development and analysis has always been one of the major problems of complex software and systems. In this example, a previously developed software baseline was updated to include additional information on one of the ARINC 429 output channels. The additional data required lower data rates on some of the other labels to ensure the output channel would not be overloaded. The requirements were updated, but the released product failed to properly interface to one particular piece of avionics, forcing a subsequent minor release to correct the error.

Technical Discussion

A particular thread within the operating system owns each ARINC 429 channel. The thread is responsible for transmitting the required data over a digital databus to the I/O card that creates the electrical signals that comprise the ARINC 429 electrical standard. The 429 data specification also recommends data transmission rates so that lower priority data such as GMT Date can be transmitted alongside more critical data such as pressure altitude. Specifying rates allow interfacing avionics to be developed to a common data rate to preclude device-specific requirements.

In the example under investigation the particular output channel sent all configured labels at the thread's scheduled rate of 25 Hz or 40 milliseconds. The incremental software update required placing additional data labels on the channel, as well as configuring the labels to be transmitted on the channel based on the installed avionics configuration. Because of the required configurability of the transmission channel labels, a previously used pattern was implemented to allow a scheduler internal to the thread that controlled labels and transmission rates as specified in software-based configuration tables.

The originally applicable requirements were spread across multiple sections of a requirements module of more than 20,000 objects. The requirements regarding this particular transmission channel were rewritten to consolidate the requirements in one section for easier maintainability. Then the configuration tables were created within the software to meet the requirements as specified in the requirements document.

Requirements-based tests were developed per our DO-178B compliant development processes. The tests were executed against the conformed software and hardware and the test steps passed, indicating the transmission rate requirements were met.

Part of the software development process for certified avionics is the submission to the FAA once the software life cycle and conformity information is complete. The result of the submission to the FAA is the receipt of the Technical Standard Orders Authorization (TSOA). The TSOA is the documentation of approval to manufacture the system, but not the installation of the system on any aircraft. Installation approval requires a type certificate (TC) or supplemental type certificate (STC). Together the (S)TC and TSOA allows for the manufacturing and installation of the device on a particular model of aircraft.

After the software was submitted to the FAA for approval and the TSOA was granted, the approved software and hardware was installed in a target aircraft. While flight and integration testing was performed on a company aircraft, this target aircraft was different than any of the bench-top or airframe integration test beds. In particular this aircraft had a different piece of traffic avoidance avionics that required a higher minimum transmission data rate for particular labels. Specifically, the PFD was now transmitting pressure altitude, label 203, at a lower rate than the minimum requirement for that particular device. The traffic system failed to initialize or properly report traffic intruders because the altitude data was marked as invalid. The result of this system defect was the rejection of the new software by the OEM until a suitable modification was made to allow proper functioning of the particular traffic system installed on this model of aircraft.

Certification Discussion

Again, the cause of this airborne system defect can be directly traced to software source code—the software failed to meet the update rate required by the IHAS-8000 traffic sensor. In this case the DO-178B process was applied, yet the defect reached the final product. The independently reviewed requirements were met, but the requirements were flawed. Nowhere does DO-178B guarantee the validity of a requirement. The guidelines are there to ensure a consistent application of engineering but the process makes no representation as the adequacy of the requirements.

Recommendation: *Additional guidance should be given regarding integration of a system to the aircraft and the specificity of the product-system boundary. (See also the recommendation above on page 70.)*

Further complicating the requirements analysis is the fact that the same piece of equipment is configured differently in different aircraft installations at the same aircraft manufacturer. Some installations connect the traffic sensor to the low-speed ARINC 429 transmission channel from the PFD, while other installations connect the exact same sensor to high-speed ARINC 429 transmission channel. The information required by the traffic sensor appears on both channels, but at different data bit rates (12.5 kbps vs. 100 kbps) as well as different data update rates (6-10 Hz vs. 16-32 Hz).

All but one of the applicable requirements was updated to reflect the new update rates. It was the omission of that requirements update that likely would have triggered additional regression and impact testing for that particular traffic system.

One of the difficulties that belie most complex systems is the use of previously developed software and life cycle information. This particular software application was one of 11 different applications spread across 9 processors and 6 different circuit boards. The PFD application alone has more than 20,000 requirements objects traced to the source code, (3922 + 16895 = 20817 as of 12 March 2009). As part of any follow-on development, requirements are reviewed for applicability, and previous credit is sought for whole sections that are unmodified as part of the release. The analysis of requirements reuse did not capture the conflicting requirements spread far apart. This single point of failure was the likely root cause of the final product defect.

The tests, requirements, code, and traceability were all reviewed, with independence, to ensure cohesiveness between all the software life cycle information. Some of the requirements were updated to reflect the new rate of a particular label, but due to the requirements organization, one displaced requirement was overlooked during the reviews. Due to the sheer number of avionics that interface to that channel in the aircraft, exhaustive hardware-in-circuit testing was not feasible. Instead tests were written to ensure the software complied with the update and transmission rates specified in the requirements documents. The tests and the test traceability to requirements were reviewed with independence to achieve the desired design assurance level for all previously approved aircraft installations. The chosen subset of tests run were listed in a software Change Impact Analysis (CIA) (see 8110.49 Chapter 11, dated 3 June 2003), and the CIA was also independently reviewed as to software, testing, and regression impact.

Summary

In this example, multiple devices interfaced to a single ARINC 429 channel, with requirements for each device. Requirements-based testing was performed based on a change impact analysis, and the as-analyzed requirements were satisfied. It was the original requirements complexity that failed to indicate that such a software change was incorrect for some avionics. As has been previously shown, the code is the final product that reaches the aircraft, and the code is the proximate cause of the system deficiency. In this case the code performed exactly as specified in the new requirements but failed to meet the older conflicting requirements. The conflicting requirements were not flagged during the impact analysis or review process and therefore not explicitly tested as part of regression testing. The use of PDS and the associated artifacts from such software created requirements analysis problems that caused a failure of the system in one particular installation.

This page is intentionally left blank.

Chapter 5 Conclusions

The current regulatory guidance lacks specificity, both in techniques recommended and examples to assist practitioners. The lack of specificity is both a blessing and a curse. The blessing comes from the ability of each firm to develop or adapt current internal processes to meet the regulatory requirements set forth by the standards. By the very nature of consensus, razor-precise recommendations are difficult to approve because each of the members must believe the resultant guidance is acceptable, and no two development firms create software in the exact same ways. The curse of the lack of specificity is the creation of a barrier to providing detailed examples of implementation processes to meet the regulatory requirements. A practitioner can read the guidance material five times and still not understand how to develop the life-cycle data required to meet the standard.

In response to the many questions and issues that arose after the release of DO-178B, the RTCA published an additional document, DO-248B to address some of the concerns of the development community. If updating the draft of DO-178C to include a variety of examples and approved techniques for life-cycle data generation is deemed inappropriate, perhaps the release of an additional document similar to DO-248B would be more appropriate. This new document could provide tangible examples of what an association of DERs has found to be generally acceptable and unacceptable means of compliance with the life-cycle requirements established in DO-178B. DO-248B clearly states that no additional guidance material is included, but that the document's purpose is solely "to provide clarification of the guidance material". The proposed document of recommended examples would include a similar statement to indicate that the information in the new document is for clarification purposes only.

Recommendations Summary

Several recommendations were made in the preceding chapters to improve or clarify parts of the regulatory process:

Separate the aviation promotion and safety aspects of the FAA to preclude any conflicts of interest regarding reduced safety in the name of advancing technology and popularizing general aviation.

The system safety assessment team must include software safety expertise to evaluate software design decisions and the implications of those decisions in the context of overall system and aircraft safety margins.

Regulatory guidance should outline testing processes such as an Integrated System Test Plan (ISTP) to verify that System Requirements are met as the system is integrated into a target aircraft.

In addition to executable object code, all utilities and data files used to load and configure the system should be under the strictest control and subject to review processes.

A call tree analysis should be performed for substantive use of PDS to ensure usage is consistent.

Ensure that no more than one requirement trace to a single verification step to ensure adequate traceability between the requirements and tests.

Control flow should be monitored for all functions, not just safety-related functions.

Every requirement, (both high- and low-level), that details timing should have a tolerance as well as detailed test steps to confirm the result of the software meets the both the high- and low-level requirements.

All behavior directly traceable to external interfaces should have requirements that govern the full range of possible input behaviors. Such specificity can limit requirements errors of omission by guaranteeing full specification of such parameters.

Integration testing should include integration with external hardware and software beyond the product boundary to ensure requirements and interface specification gaps do not cause safety-critical defects.

Additional guidance should be given regarding integration of a system to the aircraft and the specificity of the product-system boundary.

Limitations and Future Research

The critical limitations of this research are the small sample size and limited scope. As any software researcher will admit, acquiring useful data on a large scale for any commercial software development is difficult because real-world commercial firms will not develop a product twice, once under controlled conditions, and once under experimental conditions. The best software research one could realistically hope for would likely come from an academic institution performing repeated, controlled experimentation with direct outcomes not directly tied to financial viability of the product under development.

Case studies are limited by their very nature in that they cannot prove a theory conclusively because they are by definition not broad enough to ensure there are no counter examples or even alternate theories. In addition the representativeness of the sample must be evaluated when determining the weight attributed to the conclusions drawn. DO-178B was developed by a consensus of aviation professionals and opinions vary widely on software development practices. One goal of this thesis is to present suggested improvements to the applicable regulatory guidance based on case study evidence, not to rewrite the software certification guidance of the last twenty-five years. The data presented herein was acquired from a commercial firm with staff involved in the developing the next revision of the regulatory guidance material, DO-178C. As a firm active in the development of software subject to the regulatory guidance under critique, one would expect the types of software errors presented might be typical of errors faced by other firms that develop software under these guidelines.

Commercial firms are hesitant to release any information regarding software defects. The software investigated here was developed under DO-178B, indicating the software flies aboard aircraft and likely has safety-critical dependencies. Because safety is involved, firms are even less likely to provide information unless compelled to do so by regulators or the judicial system. Investigators perform and publish safety-critical software research and investigation after an accident has happened because prior to an accident there may not be any reason or authority to investigate.

Software engineering is a relatively young discipline, requiring more research to fully understand the underlying principles, assuming there are underlying principles and laws as in other engineering disciplines. Within the vast expanse of the software engineering discipline lie the subset of safety-critical software and the guidelines for developing "safeware" such as DO-178B and associated ARPs. Additional research could be performed by applying the regulatory restrictions on non-safety-critical software in an effort to determine whether the process results in fewer defects (Hicks, 2006), or whether other factors such as a safety culture (Jackson, 2007) and programming language restrictions are the true drivers of "reliable" software. DO-178B is not the only standard available for guidance in development of safety-critical software. The medical and military industries provide their own guidance that could be used in other fields as alternate options for developing safety-critical software. Increasing the scope of companies involved including start-ups and established players may also highlight interesting differences such as the ability to adapt to guidance changes or the impact of previous experience on safety-critical defects and development time invested.

Software engineering is a relatively young field still trying to discover the core principles and laws of the discipline. Further effort should be devoted to controlled experimentation to provide evidence as to which certification guidelines encourage the development of the safest software.

This page is intentionally left blank.

Appendix A – Acronyms and Initialisms

AC - Advisory Circular
ACO - Aircraft Certification Office
ADI - Attitude Direction Indicator
AOG - Aircraft On Ground
ARP - Aerospace Recommended Practice
AS - Aerospace Standard
CAST - Certification Authorities Software Team
CFIT - Controlled Flight Into Terrain
CFR - Code of Federal Regulations
CIA - Change Impact Analysis
CND - Could Not Duplicate
COTS - Commercial Of The Shelf
DER - Designated Engineering Representative
DOT - Department of Transportation
DO - Document Order
FAA - Federal Aviation Administration
FAR - Federal Aviation Regulation
FDR - Flight Data Recorder
FIFO - First-In, First-Out
FSD - Full Scale Deflection
HDI - Horizontal Deviation Indicator
HMI - Hazardously Misleading Information
IMC - Instrument Meteorological Conditions
ISTP - Integrated System Test Plan
LSB - Least Significant Bit
MCDC - Modified condition/decision coverage
MPSUE - Multi-Pilot System Usability Evaluation
NAN - Not a Number
NAS - National Aerospace System
NASA - National Aeronautics and Space Administration
NFF - No Fault Found
NIST - National Institute of Standards and Technology
NSA - National Security Agency
NTSB - National Transportation Safety Board
OO - Object Oriented
OS - Operating System
PDS - Previously Developed Software
PFD - Primary Flight Display
PSAC - Plan for Software Aspects of Certification
RAM - Random Access Memory
RMA – Rate-Monotonic Architecture
RSC - Reusable Software Component
RTCA - Radio Technical Commission for Aeronautics
SAE - Society of Automotive Engineers
SAS - Software Accomplishment Summary
SCI - Software Configuration Index
SSA - System Safety Assessment
SSM - Sign/Status Matrix

STC - Supplemental Type Certificate
TC - Type Certificate
TFC - Test For Credit
TSOA - Technical Standards Order Authorization
VDI - Vertical Deviation Indicator
VHF - Very High Frequency
VPEN - Voltage for Program Erase/Enable
VRAM - Video Random Access Memory
WAAS - Wide Area Augmentation System

Appendix B - Description of Rate-Monotonic Architecture

Rate-monotonic architecture (RMA) is a technique for scheduling real-time tasks, such that their execution and latency characteristics are deterministic. Each application task is registered to a rate group that is scheduled to execute at a pre-determined rate specified in Hertz (Hz). Each application task has a "run" method that is executed once during every scheduled invocation of its rate group. Under the RMA, faster rate groups enjoy a higher priority than lower ones. Each rate group completes and is suspended before its next scheduled invocation, yielding time to slower rate groups still in need of processor time. This can be seen in the diagram below. At the first major frame, all rate groups are scheduled to execute. The highest, 100 Hz, rate group is scheduled first. Only when it has completed execution does the next, 50 Hz, group run. In addition, upon the next minor frame the next invocation of the 100 Hz group is made, interrupting the 50 Hz group, which has not yet completed execution. For the 50 Hz group to be 'schedulable' it needs only to have completed before its next invocation frame.

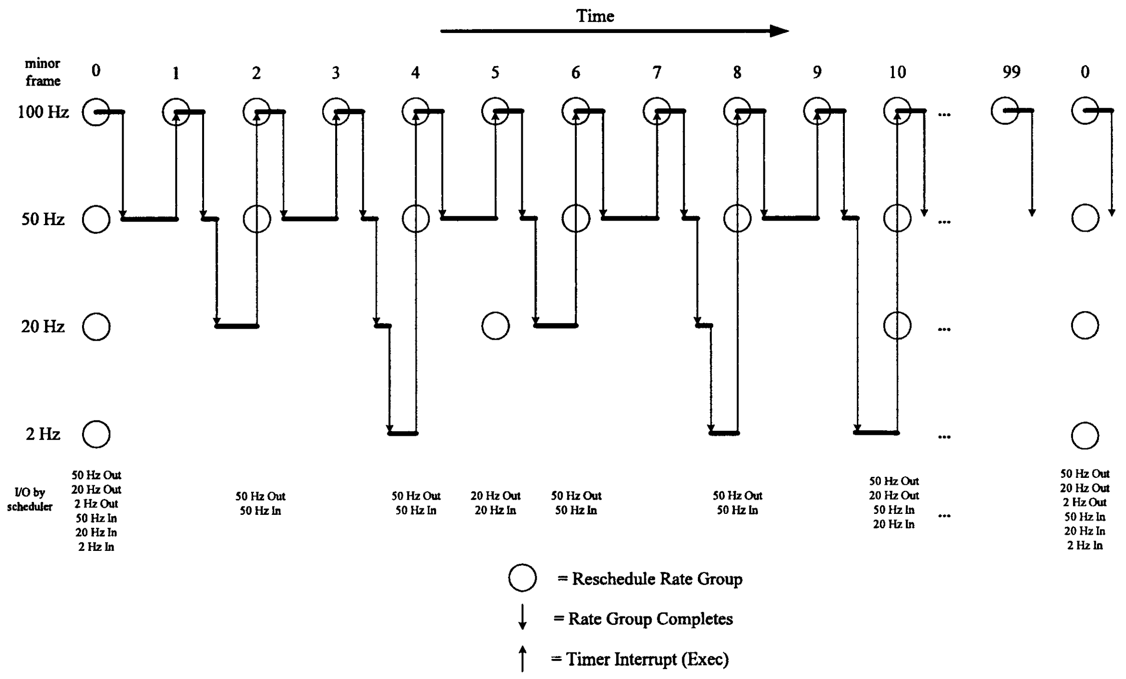

Figure 11 RMA Scheduling Algorithm Timing Diagram

The RMA and Data Interchange

The RMA provides a mechanism whereby tasks can exchange data with guaranteed coherency (i.e., each task is guaranteed that data it receives from another retains a consistent, valid state throughout its execution). For this purpose, each task that is to receive data from another must provide an input method. Each task that is to send data to another must supply an output method. At a rate group schedule point, and before the run methods are released, the tasks registered to a rate group have their output and input methods called. These are called in the context of the scheduler and are therefore not interruptible. In their output methods, tasks publish any data that they produced during the just-completed run method. In their input methods, tasks acquire any data produced by other tasks that they need for processing during their impending run methods. A standardized client/server mechanism is provided in support of these data transfers.

References

Allerton, D. J. (2007). High integrity real-time software. *Proceedings of the Institution of Mechanical Engineers; Part G. Journal of Aerospace Engineering, 221*(G1), 145-161.

Brosgol, B. (2008). Safety and security: Certification issues and technologies. *CrossTalk: the Journal of Defense and Software Engineering, 21(10),* 9-14.

Certification Authorities Software Team (2000, January). *Object-oriented technology (OOT) in civil aviation projects: Certification concerns* (CAST-4).

Hesselink, H. H. (1995). A comparison of standards for software engineering based on DO-178B for certification of avionics systems. *Microprocessors and Microsystems, 19*(10), 559-563.

Hicks, P. (2006). Adapting legacy systems for DO-178B certification. *CrossTalk: The Journal of Defense Software Engineering, 19*(8), 27-30.

Jackson, D., Thomas, M., & Millett, L. (2007). *Software for Dependable Systems, Sufficient Evidence*. Washington D.C.: Academic Press.

Johnson, L. A. (1998, October). DO-178B Software considerations in airborne systems and equipment certification [Online exclusive]. *CrossTalk: the Journal of Defense Software Engineering, http://www.stsc.hill.af.mil/CrossTalk/1998/10/schad.asp.*

Kim, M., Kim, S., & Choi, M. (2005). Practical design recovery techniques for embedded operating system on complying with RTCA/DO-178B and ISO/IEC15408. *Knowledge-Based Intelligent Information and Engineering Systems, 3683,* 621-627. doi:10.1007/11553939_89

Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Boston: Addison-Wesley.

Leveson, N. G. (2004). A systems-theoretic approach to safety in software-intensive systems. *Dependable and Secure Computing, IEEE Transactions on, 1*(1), 68-86.

Lions, J. L. (1996, July 19). *Ariane 501 failure: Report by the inquiry board*. Paris: European Space Agency.

Matharu, J. (2006, February). Reusing safety-critical. *IEE Electronics Systems and Software, 4,* 32-35.

Maxey, B. (2003). COTS integration in safety critical systems using RTCA/DO-178B guidelines. *COTS-Based Software Systems, 2580,* 134-142. doi:10.1007/3-540-36465-X_13

Myers, G. J. (1976). *Software Reliability: Principles and Practices*. New York: John Wiley & Sons.

National Transportation Safety Board. (n.d.). Retrieved from United States Department of Transportation Web site: http://www.ntsb.gov/events/2006/SR0601/April%2025%20Board%20Meeting%20no%20scripts.pdf

Ramsey, J. (2005, August 1). *Special Report: Avoiding NFF*. Retrieved from Avionics Magazine Web site: http://www.aviationtoday.com/av/categories/bga/Special-Report-Avoiding-NFF_1046.html

RTCA (1982, January). *Software considerations in airborne systems and equipment certification* (DO-178). Washington, D.C.: RTCA.

RTCA (1985, March). *Software considerations in airborne systems and equipment certification* (DO-178A). Washington, D.C.: RTCA.

RTCA (1992, December). S*oftware considerations in airborne systems and equipment certification* (DO-178B). Washington, D.C.: RTCA.

RTCA (2001, October). *Final report for clarification of DO-178B* (DO-248B). Washington, D.C.: RTCA.

Sakugawa, B., Cury, E., & Yano, E. (2005). Airborne software concerns in civil aviation certification. *Dependable Computing, 3747,* 52-60. doi:10.1007/11572329

Society Of Automotive Engineers (1996). *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* (Aerospace Recommended Practice 4761). Warrendale, PA: SAE International.

Society Of Automotive Engineers (1996, November 1). *Certification considerations for highly-integrated or complex aircraft systems* (Aerospace Recommended Practice 4754). Warrendale, PA: SAW International.

Tuohey, W. G. (2002). Benefits and effective application of software engineering standards. *Software Quality Journal, 10*(1), 47-68.

United States Department of Defense (1988, February 28). *DoD-STD-2167A Defense system software development* (AMSC No. N4327). Washington, D.C.

United States Department of Defense (1994, December 5). *DoD-STD-498 Software development and documentation* (AMSC No. N7069). Washington, D.C.

United States Department of Transportation Federal Aviation Administration. (2005, March 10). *What we do*. Retrieved April 18, 2009, from http://www.faa.gov/about/mission/activities

United States Department Of Transportation, Federal Aviation Administration (2003, June 2). *Software approval guidelines* (Advisory Circular 8110.49). Washington, D.C.:

United States Department of Transportation, Federal Aviation Administration (2009, January 16). *System safety analysis and assessment for part 23 airplanes* (Advisory Circular No. 23.1309-1D). Washington, D.C.

United States Department of Transportation, Federal Aviation Administration (1999, March 12). *Equipment, systems, and installations in part 23 airplanes* (Advisory Circular No. 23.1309-1C). Washington, D.C.

United States Department of Transportation, Federal Aviation Administration (2004, December 7). *Reusable software components* (Advisory Circular No. 20-148). Washington, D.C.

United States Department of Transportation, Federal Aviation Administration (1993). *Radio Technical Commission for Aeronautic, Inc. Document RTCA/DO-178B* (Advisory Circular No. 20-115B). Washington, DC

United States National Aeronautics and Space Administration (2004, March 31). *NASA Software Safety Guidebook* (NASA-GB-8719.13).

Wils, A., Van Baelen, S., Holvoet, T., & De Vlaminck, K. (2006). Agility in the avionics software world. *Extreme Programming and Agile Processes in Software Engineering, 4044,* 123-132. doi:10.1007/11774129