

# Software Verification Plan

for the

**<Program Name>**

Document No: <Doc Number>

Revision: -

\_\_\_\_\_  
<Name>, Program Manager

\_\_\_\_\_  
Date

\_\_\_\_\_  
<Name>, Technical Project Lead

\_\_\_\_\_  
Date

\_\_\_\_\_  
<Name>, Engineer

\_\_\_\_\_  
Date

\_\_\_\_\_  
<Name>, Quality Engineer

\_\_\_\_\_  
Date

## Notice

This document and the information contained herein are the property of <Company Name>. Any reproduction, disclosure or use thereof is prohibited except as authorized in writing by <Company Name>. Recipient accepts the responsibility for maintaining the confidentiality of the contents of this document.



---

**Table of Contents**

<b>Section</b>	<b>Page</b>
<b>1.0 INTRODUCTION .....</b>	<b>8</b>
1.1 Purpose .....	8
1.2 Scope .....	8
1.3 Acronyms and Abbreviations .....	9
1.4 Applicable Documents .....	10
1.4.1 External Documents .....	10
1.4.2 Internal Documents .....	10
<b>2.0 ORGANIZATION .....</b>	<b>11</b>
2.1 Team Member Responsibilities .....	11
<b>3.0 INDEPENDENCE .....</b>	<b>15</b>
3.1 Peer Reviews .....	16
3.2 Independence of DO-178C Objectives .....	17
<b>4.0 VERIFICATION METHODS .....</b>	<b>20</b>
4.1 V-Model Verification Approach .....	20
4.2 Analysis of Outputs Methods .....	22
4.2.1 Traceability of Reviews and Analysis Results .....	23
4.2.2 Transition Review Planning .....	24
4.2.3 Peer Review Planning .....	24
4.2.4 Software Planning Process Verification Methods .....	25
4.2.5 Planning Process Verification Activities .....	25
4.2.6 Software Planning Process Inputs .....	25
4.2.7 Software Planning Process Reviews and Analysis .....	25
4.2.7.1 Software Verification Plan Review .....	25
4.2.7.2 Software Planning Review .....	26
4.3 Software Requirements Process Verification Methods .....	28
4.3.1 Software Requirements Process Verification Objectives .....	28
4.3.2 Software Requirements Process Inputs .....	28
4.3.3 Transition Criteria for Entering The Verification of Requirements Process .....	28
4.3.4 Software Requirements Process Reviews and Analysis .....	29
4.3.4.1 Software Requirements Document Review .....	30
4.3.4.2 Software Requirements Review .....	31
4.3.4.3 Analysis of High-Level Software Requirements .....	32
4.3.4.4 System and Software Requirements Trace Analysis .....	32
4.4 Software Design Process Verification Methods .....	33
4.4.1 Software Design Process Verification Objectives .....	33
4.4.2 Software Design Process Inputs .....	33
4.4.3 Transition Criteria for Entering The Verification of Design Process .....	33
4.4.4 Software Design Process Reviews and Analysis .....	34
4.4.4.1 Software Design Description Review .....	34
4.4.4.2 Software Preliminary Design Review .....	35
4.4.4.3 Software Critical Design Review .....	36
4.4.5 Reviews and Analysis of Software Architecture .....	37

---

4.4.6	<i>Reviews and Analysis of Low-Level Software Requirements</i> .....	37
4.5	Software Coding Process Verification Methods .....	38
4.5.1	<i>Software Coding Process Verification Objectives</i> .....	38
4.5.2	<i>Software Verification Process Inputs</i> .....	38
4.5.3	<i>Transition Criteria for Entering The Verification of Software Coding Process</i> ....	38
4.5.4	<i>Software Coding Process Reviews and Analysis</i> .....	38
4.5.4.1	Source Code File Review .....	39
4.5.4.2	Source Code Review.....	39
4.5.5	<i>Reviews and Analysis of Source Code</i> .....	39
4.6	Integration Process Verification Methods.....	41
4.6.1	<i>Integration Process Verification Objectives</i> .....	41
4.6.2	<i>Integration Process Inputs</i> .....	41
4.6.3	<i>Transition Criteria for Entering The Verification of Integration Process</i> .....	41
4.6.4	<i>Integration Process Reviews and Analysis</i> .....	42
4.6.4.1	Executable Object Code Review .....	42
4.6.4.2	System Integration Review .....	42
4.6.4.3	Reviews and Analysis of Executable Object Code .....	42
4.7	Software Testing Process Verification Methods.....	44
4.7.1	<i>Software Testing Process Verification Objectives</i> .....	44
4.7.2	<i>Software Testing Process Inputs</i> .....	44
4.7.3	<i>Transition Criteria for Entering The Testing of Integration Process Outputs</i> ....	44
4.7.4	<i>Transition Criteria for Entering The Verification of Verification Outputs</i> .....	45
4.7.5	<i>Software Testing Process Reviews and Analysis</i> .....	45
4.7.5.1	Software Verification Cases and Procedures Document Review .....	45
4.7.5.2	System Verification Review .....	46
4.7.5.3	Reviews and Analysis of Test Cases, Test Procedures, and Results .....	46
4.7.5.3.1	Review checklists for test cases, procedures, and results .....	47
4.7.6	<i>Software Test Execution</i> .....	49
4.7.6.1	Test Environment .....	50
4.7.6.2	Requirements-Based Test Cases .....	50
4.7.6.3	Normal Range Test Cases .....	50
4.7.6.4	Robustness Test Cases .....	51
4.7.6.4.1	Robustness Test Case Selection Strategy .....	51
4.7.6.5	Requirements-Based System Verification Testing Methods .....	52
4.7.6.5.1	Requirements-Based Software Verification Testing .....	55
4.7.6.5.2	Requirements-Based Low-Level Testing .....	56
4.7.7	<i>Effectiveness of Test Program</i> .....	56
4.7.7.1	Assess results of requirements-based tests .....	56
4.7.7.2	Assess failure explanations and rework.....	57
4.7.7.3	Assess coverage achievement .....	57
4.8	Coverage Analysis Methods.....	58
4.8.1	<i>Requirements Coverage Analysis</i> .....	59
4.8.2	<i>Structural Coverage Analysis</i> .....	59
4.8.2.1	Achieving Coverage .....	60
4.8.2.2	Coverage Analysis Methods.....	62
4.8.2.3	Statement Coverage .....	64
4.8.2.4	Modified Condition Decision Coverage .....	64
4.8.3	<i>Data Coupling and Control Coupling Analysis</i> .....	79
4.8.3.1	Data Coupling Analysis .....	79
4.8.3.2	Control Coupling Analysis .....	82
4.9	Process-Specific Activities .....	84
4.9.1	<i>Test Case Development</i> .....	84

---

4.9.2	Test Case Verification .....	85
4.9.3	Test Procedure Development .....	85
4.9.4	Test Procedure Verification .....	86
4.9.5	Coverage Analysis Verification .....	86
4.9.6	Testing Environment .....	87
4.9.7	Test Execution .....	87
4.9.8	All traceability data is reviewed and under CM control with no outstanding (non-deferrable) PRs Software Testing Process Reviews and Analysis .....	88
4.9.8.1	Software Verification Cases and Procedures Document Review .....	88
4.9.8.2	System Verification Review .....	89
4.9.8.3	Reviews and Analysis of Test Cases, Test Procedures, and Results .....	89
4.9.8.3.1	Review checklists for test cases, procedures, and results .....	90
4.9.9	Software Test Execution .....	92
4.9.9.1	Test Environment .....	93
4.9.9.2	Requirements-Based Test Cases .....	93
4.9.9.3	Normal Range Test Cases .....	93
4.9.9.4	Robustness Test Cases .....	94
4.9.9.4.1	Robustness Test Case Selection Strategy .....	94
4.9.9.5	Requirements-Based System Verification Testing Methods .....	95
4.9.9.5.1	Requirements-Based Software Verification Testing .....	98
4.9.9.5.2	Requirements-Based Low-Level Testing .....	99
4.9.10	Effectiveness of Test Program .....	99
4.9.10.1	Assess results of requirements-based tests .....	99
4.9.10.2	Assess failure explanations and rework .....	100
4.9.10.3	Assess coverage achievement .....	100
4.10	Coverage Analysis Methods .....	101
4.10.1	Requirements Coverage Analysis .....	102
4.10.2	Structural Coverage Analysis .....	102
4.10.2.1	Achieving Coverage .....	103
4.10.2.2	Statement Coverage .....	105
4.10.2.3	Decision Coverage .....	105
4.10.2.4	Modified Condition Decision Coverage .....	105
4.10.2.5	Coverage Analysis Tools .....	120
4.10.3	Source Code to Object Code Traceability .....	122
4.10.4	Data Coupling and Control Coupling Analysis .....	122
4.10.4.1	Structural Coverage Analysis of Data and Control Coupling .....	122
4.10.4.2	Data Coupling Analysis .....	123
4.10.4.3	Control Coupling Analysis .....	124
4.10.4.4	Outputs of Data and Control Coupling Activity .....	126
	Process-Specific Activities .....	127
4.10.5	Test Case Development .....	127
4.10.6	Test Case Verification .....	128
4.10.7	Test Procedure Development .....	128
4.10.8	Test Procedure Verification .....	129
4.10.9	Coverage Analysis Verification .....	129
4.10.10	Testing Environment .....	130
4.10.11	Test Execution .....	130
4.10.12	Test Results Verification .....	131
<b>5.0</b>	<b>VERIFICATION ENVIRONMENT .....</b>	<b>132</b>
5.1	Test Environment Description .....	132
5.1.1	Block Diagram of Test Environment .....	132

5.2	List of Test Equipment Used To Verify Software .....	132
5.3	Testing and Analysis Tools .....	132
5.3.1	<i>Guidelines for Applying the Tools and Hardware Test Environment .....</i>	<i>132</i>
5.4	Test Procedure Structure .....	133
<b>6.0</b>	<b>TRANSITION CRITERIA.....</b>	<b>135</b>
<b>7.0</b>	<b>PARTITIONING CONSIDERATIONS .....</b>	<b>136</b>
7.1	Guidelines for Evaluating Protection .....	136
7.1.1	<i>Time .....</i>	<i>137</i>
7.1.2	<i>Space.....</i>	<i>138</i>
7.2	Project Specific Partitioning.....	138
<b>8.0</b>	<b>COMPILER ASSUMPTIONS.....</b>	<b>139</b>
<b>9.0</b>	<b>REVERIFICATION GUIDELINES .....</b>	<b>140</b>
9.1	Inspect, Review, or Analyze Changes .....	140
9.2	Perform Regression Testing.....	140
9.3	Perform Other Verification .....	141
<b>10.0</b>	<b>PREVIOUSLY DEVELOPED SOFTWARE .....</b>	<b>142</b>
<b>11.0</b>	<b>MULTIPLE VERSION DISSIMILIAR SOFTWARE.....</b>	<b>143</b>
<b>Appendix A:</b>	<b>Software Planning Review Checklist.....</b>	<b>144</b>
<b>Appendix B:</b>	<b>Software Requirements Review Checklist.....</b>	<b>149</b>
<b>Appendix C:</b>	<b>Software Preliminary Design Review Checklist.....</b>	<b>151</b>
<b>Appendix D:</b>	<b>Software Critical Design Review Checklist.....</b>	<b>153</b>
<b>Appendix E:</b>	<b>Software Code Review Checklist.....</b>	<b>155</b>
<b>Appendix F:</b>	<b>Integration Review Checklist.....</b>	<b>157</b>
<b>Appendix G:</b>	<b>Software Verification Review Checklist .....</b>	<b>159</b>
<b>Appendix H:</b>	<b>Software Conformity Review Checklist.....</b>	<b>163</b>
<b>Appendix I:</b>	<b>Peer Review Checklist - Planning .....</b>	<b>165</b>
<b>Appendix J:</b>	<b>Peer Review Checklist - Requirements.....</b>	<b>171</b>
<b>Appendix J:</b>	<b>Peer Review Checklist - Design .....</b>	<b>175</b>
<b>Appendix K:</b>	<b>Peer Review Checklist - Code .....</b>	<b>180</b>
<b>Appendix L:</b>	<b>Peer Review Checklist - Integration.....</b>	<b>183</b>
<b>Appendix M:</b>	<b>Peer Review Checklist – Test Procedures.....</b>	<b>184</b>

<b>Appendix N: Peer Review Checklist – Test Results .....</b>	<b>187</b>
---	------------

## 1.0 INTRODUCTION

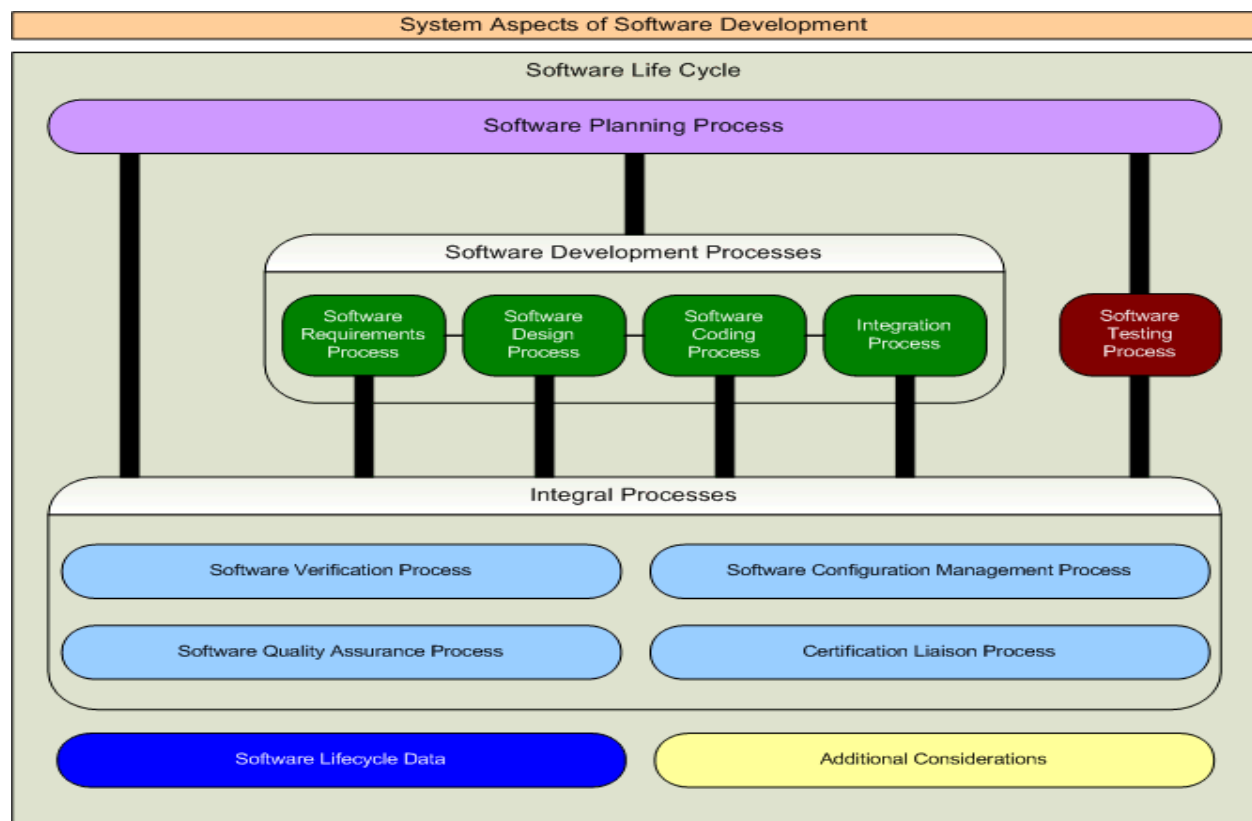
### 1.1 Purpose

This plan describes the Software Verification Process to be implemented on the Program. This plan has been prepared in accordance with the requirements of RTCA/DO-178C. The purpose of the verification process is to detect and report errors that have been introduced in the development processes. The verification process does not produce software; its responsibility is to ensure that the produced software implements its intended function completely and correctly, while avoiding unintended functions. Because each development process may introduce errors, verification is an integral process, which is coupled with every development process. The verification process ensures that the software product is built as designed, with no unexpected functionality. The verification process is also intended to ensure that the software will perform under any foreseeable operating conditions.

### 1.2 Scope

This plan will be used by the certification authority to determine if the Software Life Cycle Process is commensurate with the rigor required for the level of software being developed. Once approved, it is implemented during the development and product life cycle of the deliverable airborne software. This Software Verification Plan complies with the documentation requirements of RTCA/DO-178C, Section 11.3.

The following diagram illustrates the lifecycle process and shows the Verification Process as an integral process associated with all other planning and development processes.



### 1.3 Acronyms and Abbreviations

<PROJ>	<Add Project Acronyms in Alphabetical Order>
RAMS	Reviews and Analysis Management System
CAMS	Coverage Analysis Management System
CC1	DO-178C Control Category 1
CC2	DO-178C Control Category 2
CI	Configuration Item
CM	Configuration Management
COTS	Commercial off the Shelf
CPU	Central Processing Unit
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DER	Designated Engineering Representative
DRMS	Document Review Management System
FAA	Federal Aviation Administration
FHA	Functional Hazard Assessment
IVT	Independent Verification Testing
MC/DC	Modified Condition/Decision Coverage
MISRA	Motor Industries Software Reliability Association
MLCP	Master Load Control Procedure
PEMS	Project Event Management System
PRMS	Problem Reporting Management System
PSAC	Plan for Software Aspects of Certification
PSSA	Preliminary System Safety Assessment
PVCS	Serena PVCS Version Control Software
QA	Quality Assurance
RTCA	Radio Technical Commission for Aeronautics
RTMS	Requirements Traceability Management System
SAS	Status Accounting System
SCI	Software Configuration Index
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCS	Software Coding Standard
SDD	Software Design Description
SDS	Software Design Standard
SDP	Software Development Plan
SECI	Software Environment Configuration Index
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SQE	Software Quality Engineer
SRS	Software Requirements Standard
SSA	System Safety Assessment
SVC&P	Software Verification Cases and Procedures
SVCP	Software Verification Cases and Procedures
SVP	Software Verification Plan
SWRD	Software Requirements Document
VR	Verification Results
VSS	Visual Source Safe

### 1.4 Applicable Documents

The following documents are listed for reference only. Each document is applicable to this plan only to the extent specified herein.

#### 1.4.1 External Documents

RTCA/DO-178C	Software Considerations in Airborne Systems and Equipment Certification
FAA Order 8110.4C	Type Certification
FAA Order 8110.49	FAA, Software Approval Guidelines
AC 20-115C	Advisory Circular, RTCA Inc., Document DO-178C, Software Considerations in Airborne Systems and Equipment Certification
CAST-19	Certification Authorities Software Team (CAST) Position Paper CAST-19: Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling (Rev 2).
<Spec Number>	<Customer Specification>
<Spec Number>	<Regulatory Specification>

#### 1.4.2 Internal Documents

<Ref Doc>	Plan for Software Aspects of Certification (Ref. DO-178C, 11.1)
<Ref Doc>	Software Development Plan (Ref. DO-178C, 11.2)
<Ref Doc>	Software Verification Plan (Ref. DO-178C, 11.3)
<Ref Doc>	Software Configuration Management Plan (Ref. DO-178C, 11.4)
<Ref Doc>	Software Quality Assurance Plan (Ref. DO-178C, 11.5)
<Ref Doc>	Software Design Standards (Ref. DO-178C, 11.7)
<Ref Doc>	Software Code Standards (Ref. DO-178C, 11.8)
<Ref Doc>	Software Requirements Document (Ref. DO-178C, 11.9)
<Ref Doc>	Software Design Description (Ref. DO-178C, 11.10)
<Ref Doc>	Build Procedure for Source Code (Ref. DO-178C, 11.11)
<Ref Doc>	Load Control for Executable Object Code (Ref. DO-178C, 11.12)
<Ref Doc>	Software Verification Cases and Procedures (Ref. DO-178C, 11.13)
<Ref Doc>	Software Verification Results (Ref. DO-178C, 11.14)
<Ref Doc>	Software Environment Configuration Index (Ref. DO-178C, 11.15)
<Ref Doc>	Software Configuration Index (Ref. DO-178C, 11.16)
<Ref Doc>	Software Accomplishment Summary (Ref. DO-178C, 11.20)

## 2.0 ORGANIZATION

Software verification activities will be performed by an individual or individuals other than the developer. The following matrix summarizes the allocation:

Verification Activity	Team	Out Source	Other
Verification of Outputs of Software Requirements Process	✓		
Verification of Outputs of Software Design Process		✓	
Verification of Outputs of Coding & Integration Process		✓	
Testing of Outputs of Integration Process			✓
Verification of Verification Process Results			✓

### 2.1 Team Member Responsibilities

<b>Systems Engineering &amp; Software Engineering</b>
Plan for Software Aspects of Certification
Software Verification Plan
Software Requirements Standards
Analysis of System Requirements (Complete Requirements Peer Review Checklists)
Software Requirements Document
Software Accomplishment Summary
Complete System Requirements Document Checklist
Complete Software Requirements Document Checklist
Complete Software Verification Cases and Procedures
Complete Software Verification Results
Structural Coverage Analysis Results
<b>Software Configuration Management</b>
Software Configuration Management Plan
Software Lifecycle Environment Configuration Index
Software Configuration Index
Release Plan for Software Aspects of Certification
Release Software Development Plan
Release Software Verification Plan
Release Software Configuration Management Plan
Release Software Quality Assurance Plan
Release Software Requirements Standards
Release Software Design Standards

<b>Software Configuration Management</b>
Release Software Code Standards
Release System Requirements Document
Release Software Requirements Document
Release Software Design Description
Establish Software Library
Release Low Level Software Verification Cases and Procedures
Release Low Level Verification Results
Release Source Code
Release Software Verification Cases and Procedures
Release Software Verification Results
Release Structural Coverage Analysis Results
Release Software Lifecycle Environment Configuration Index
Release Software Configuration Index
Release Software Accomplishment Summary

<b>Software Engineering &amp; Independent Verification Engineers</b>
Software Development Plan
Software Design Standards
Software Code Standards
Software Design Description
Source Code
Analysis of Requirements (Complete Requirements Peer Review Checklists)
Analysis of Design (Complete Design Peer Review Checklists)
Analysis of Code (Complete Code Peer Review Checklists)
Analysis of Integration (Complete Integration Peer Review Checklists)
Analysis of Test Cases, Procedures and Results (Complete Test Peer Review Checklists)
Low Level Verification Cases and Procedures
Low Level Verification Results
Executable Object Code
Complete Software Design Description Checklist
Complete Low Level Software Verification Cases and Procedures Checklist
Complete Low Level Software Verification Results Checklist

<b>Transition Review Team</b>
Software Planning Review
Software Requirements Review
Software Preliminary Design Review
Software Critical Design Review
Software Code Review
System Integration Review
System Verification Review
<b>Safety Engineering</b>
Review & Approval of Derived Requirements
Functional Hazard Assessment
Preliminary System Safety Assessment
System Safety Assessment
<b>Software Quality Assurance</b>
Software Quality Assurance Plan
Complete Plan for Software Aspects of Certification Checklist
Complete Software Development Plan Checklist
Complete Software Verification Plan Checklist
Complete Software Configuration Management Plan Checklist
Complete Software Quality Assurance Plan Checklist
Complete Software Requirements Standards Checklist
Complete Software Design Standards Checklist
Complete Software Code Standards Checklist
Complete Software Requirements Document Review Checklist
Complete Software Design Document Review Checklist
Complete Software Verification Cases and Procedures Document Review Checklist
Complete Software Verification Results Review Checklist
Complete Software Lifecycle Environment Configuration Index Checklist
Complete Software Configuration Index Checklist
Complete Software Accomplishment Summary Checklist
Transition Criteria Verification (Planning Review Checklists)
Transition Criteria Verification (Requirements Review Checklists)
Transition Criteria Verification (Preliminary Design Review Checklists)
Transition Criteria Verification (Critical Design Review Checklists)
Transition Criteria Verification (Code Review Checklists)
Transition Criteria Verification (Integration Review Checklists)
Transition Criteria Verification (Verification Review Checklists)
Software Conformity Review
Perform Surveillance and Pre-SOI Audits

<b>FAA Software DER</b>
Plan for Software Aspects of Certification Approval
Stages of Involvement Audit #1 Audit (Planning Review)
Stages of Involvement Audit #2 Audit (Design Review)
Stages of Involvement Audit #3 Audit (Verification Review)
Stages of Involvement Audit #4 Audit (Final Review)
Software Configuration Index Approval
Software Accomplishment Summary Approval
Complete FAA Form 8110-3

### 3.0 INDEPENDENCE

Independence is achieved through the “No Sole Perspective” method. This perspective proposes that there is value in having someone other than the developer of the data review the data, and that it satisfies the criteria for having an “objective evaluation” without requiring organizational independence. In fact, this perspective recommends that there is additional benefit in having multiple other persons involved in each review from different disciplines (such as systems engineers, safety specialists, test engineers, human factors specialists, technical writers, etc.). Also, by having other disciplines involved in the review, one could potentially be getting the greatest possible “objective evaluation” of the data. Independent reviews help prevent a biased perspective since it may be difficult to impartially review one’s own work.

Additionally, the value of having an independent reviewer involved in the software engineering discipline is supported by extensive research and application. It is also intuitive and reasonable that having someone other than the author or developer of an artifact, review (inspect) that artifact from their different perspectives, disciplines, and experiences will provide for higher quality, safer, easier to maintain, and less expensive (in the long run) products.

This project expands on the “No Sole Perspective,” and proposes the following guidelines:

- a. General Position: To achieve verification independence, the person performing or responsible for the verification activity will not be the same person who developed the data being verified.
- b. Tool Qualification: If a tool is used to eliminate, reduce or automate the activities associated with a DO-178C objective needing verification independence and that tool’s output will not be completely verified with independence, then that tool will be qualified.
- c. Test Case and Procedure Development: The test cases and procedures will not be developed by the same person who developed the low-level requirements or source code to be verified by those test cases and procedures.
- d. Test Case and Procedure Review: The person responsible for performing the test cases and procedures review will not be the same person who developed the test cases and procedures to be verified.
- e. Test Execution: The person responsible for executing the tests will not be the same person who developed the requirements or code being verified by the tests, nor the developer of the test cases and procedures being executed. If the test execution is fully automated (e.g., scripted “batch” run with no need for human intervention or observation), then this guideline would not apply. However, that test “tool” may need to be qualified and the developer of the testing tool (that person setting up the automated test execution and environment) will not be the same person who developed the test cases and procedures.
- f. Test Results Review and Coverage Analysis: The person responsible for performing the test results review or test coverage analysis will not be the same person who developed the test cases and procedures, or the same person who executed the tests.

### 3.1 Peer Reviews

Peer reviews will be used as the primary means to obtain verification independence. Within the scope of this project, peer review is defined as the evaluation of the conceptual and technical soundness of a design by individuals qualified by their education, training and experience in the same discipline, or a closely related field of science, to judge the worthiness of a design or to assess a design for its likelihood of achieving the intended objectives and the anticipated outcomes. A peer review may be conducted on any or all components of a design, conceptual approaches or recommendations, application or interpretation of code requirements or supporting analysis and calculations.

The scope of the peer review may be a complete review of the entire documentation, including compliance with applicable requirements, design, coding and verification standards and the appropriateness of the assumptions, engineering methods and input data used to support the design. Alternatively, the scope of the peer review might be limited to specific aspects of the design documentation, such as specific models or methods and their associated input data and conclusions drawn from the output data. Agreement on the scope of the peer review is achieved between the contracting stakeholder and the peer reviewer and documented in the Reviews and Analysis Management System (see screen shot below). The scope of the review explicitly identified in this tool at the time of execution of the agreement to undertake the peer review. Any changes to the scope must be agreed to by both the contracting stakeholder and the peer reviewer. The peer review is limited to only the technical aspects of the design documentation.

#### Sample Screen Short: Action Item Management System

The screenshot displays the 'Action Item Management System' web application. The browser window shows the address bar with the URL 'https://www.faaconsultants.com/SCMS/Projects/ExampleProject/000/ADP6/Lat.asp'. The application header includes a search bar and a 'Search' button. Below the header, there is a navigation bar with links: 'Program: P1 - Flight Management System | DO-178 |', 'Add Review', 'All Action Items', 'Open Action Items', 'My Action Items', 'DO-178B JobAid', and 'Project List'.

The main content area is titled 'List of Reviews, checklists, action items and sign-in sheets'. It contains a table with the following columns: Review Title, Review Topic, Review Date, Sign-in Sheet / Outlook Meeting Request, Checklist, Action Items, Closed, Edit, and Delete.

Review Title	Review Topic	Review Date	Sign-in Sheet / Outlook Meeting Request	Checklist	Action Items	Closed	Edit	Delete
Software Planning Review	Transition Compliance	8/8/2008	Create Notify Print Upload PDF	View Modify	Add/View			
Software Requirements Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Preliminary Design Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Critical Design Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Code Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Integration Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Verification Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Software Conformity Review	Transition Compliance	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
SQA - Planning	SQA / DER Oversight	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
SQA - Development	SQA / DER Oversight	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
SQA - Verification	SQA / DER Oversight	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
SQA - Final	SQA / DER Oversight	1/1/2009	Create Notify Print Upload PDF	View Modify	Add/View			
Peer Review	Review C2 Requirements	9/26/2008	Create Notify Print Upload PDF	View Modify	Add/View			
Peer Review - Requirements	Look at test string	12/15/2008	Create Notify Print Upload PDF	View Modify	Add/View			

### 3.2 Independence of DO-178C Objectives

The following matrix shows the DO-178C objectives that will be satisfied with independence.

Table	Objective	Verification Activity	Item Being Verified	Interpretation
A-3(1)	Software high-level requirements comply with system requirements.	Reviews and Analyses of the High-Level Requirements	High-level requirements	The reviews and analyses of the high-level requirements will be performed by a person(s) other than the developer of the high-level requirements.
A-3(2)	High-level requirements are accurate and consistent.			
A-3(7)	Algorithms are accurate.			
A-4(1)	Low-level requirements comply with high-level requirements.	Reviews and Analyses of the Low-Level Requirements	Low-level requirements	The reviews and analyses of the low-level requirements will be performed by a person(s) other than the developer of the low-level requirements.
A-4(2)	Low-level requirements are accurate and consistent.			
A-4(7)	Algorithms are accurate.			
A-4(8)	Software architecture is compatible with high-level requirements.	Reviews and Analyses of the Software Architecture	Software architecture	The reviews and analyses of the software architecture will be performed by a person(s) other than the developer of the software architecture.
A-4(9)	Software architecture is consistent.			
A-4(13)	Software partitioning integrity is confirmed.			
A-5(1)	Source Code complies with low-level requirements.	Reviews and Analyses of the Source Code	Source Code	The reviews and analyses of the Source Code will be performed by a person(s) other than the developer of

Table	Objective	Verification Activity	Item Being Verified	Interpretation
A-5(2)	Source Code complies with software architecture.			the Source Code.
A-5(6)	Source Code is accurate and consistent.			
A-6(3)	Executable Object Code complies with low-level requirements.	Requirements-Based Testing	Executable Object Code	<p>The person(s) who created a set of low-level requirements-based test cases should not be the same person(s) who developed the associated Source Code from those low-level requirements. It follows that:</p> <ol style="list-style-type: none"> <li>1. The same person(s) could develop the low-level requirements and the Source Code, provided another person(s) develops the test cases from those low-level requirements, or</li> <li>2. The same person(s) could develop the low-level requirements and their associated test cases, provided another person(s) develops the Source Code.</li> </ol>
A-6(4)	Executable Object Code is robust with low-level requirements.			
A-7(1)	Test procedures and expected results are correct.	Reviews and Analyses of the Test Procedures	Test procedures	The reviews and analyses of the test procedures will be performed by a person(s) other than the developer of the test procedures.
A-7(2)	Test results are correct and discrepancies explained.	Reviews and Analyses of the Test Results	Test results	The reviews and analyses of the test results will be performed by a person(s) other than the person(s) who performed the tests.
A-7(3)	Test coverage of high-level requirements is achieved.	Requirements-Based Test Coverage Analysis	Test cases	The requirements-based test coverage analysis will be performed by a person(s) other than the developer of the test cases.
A-7(4)	Test coverage of low-level requirements is achieved.			

Table	Objective	Verification Activity	Item Being Verified	Interpretation
A-7(5)	Test coverage of software structure (modified condition/decision) is achieved.	Structural Coverage Analysis	Test cases, test procedures, and/or test results	The exact independence required depends on how the structural coverage analysis is carried out. For example, if the structural coverage analysis is performed on the test cases, then the structural coverage analysis will be performed by a person(s) other than the developer of the test cases. Similarly, if the structural coverage analysis is performed on the test procedures and test results, then the structural coverage analysis will be performed by a person(s) other than the developer of the test procedures and test results.
A-7(6)	Test coverage of software structure (decision coverage) is achieved.			
A-7(7)	Test coverage of software structure (statement coverage) is achieved.			
A-7(8)	Test coverage of software structure (data coupling and control coupling) is achieved.			
A-7(9)	Verification of additional code, that cannot be traced to Source Code, is achieved.			

#### 4.0 VERIFICATION METHODS

The Software Verification Process utilizes three methods to verify that the objectives of each process have been satisfied. These methods include review, analysis and test. Reviews and analysis are applied to the results of the software development and software testing processes. Reviews provide a quantitative assessment of correctness and consist of inspection of outputs of the processes guided by checklists. Analysis provides repeatable evidence of correctness and examines in detail the functionality, performance, traceability and safety implications of a software component and its relationship to other components. Testing will be used to exercise the software to verify that it satisfies specific requirements and to detect errors in the software.

##### 4.1 V-Model Verification Approach

A V-Model approach, as detailed in the Software Development Plan, will be used during development and verification. This model is summarized below.

Development Activity	Validation & Verification Activity
Requirements	Validate Requirements & Trace Data
Preliminary Design	Verify Conceptual Design & Trace Data
Detail Design	Verify Detail Design & Trace Data
Integration	Verify Integration & Trace Data
Requirements-Based Test Case Creation	Verify Test Cases & Trace Data
Test Procedure Case Creation	Verify Test Procedures & Trace Data
<b>Implementation</b>	
Requirements-Based Test Execution	Verify Test Results & Trace Data
Structural Coverage Execution	Verify Structural Coverage
Structural Coverage Analysis	Perform Coverage Analysis Resolution

**4-1 V-Model Relationship Table**

Early in the review process, the Software Verification Plan is reviewed to ensure that activities planned for achieving test coverage, if followed, will satisfy the DO-178C objective. Other plans including the Plan for Software Aspects of Certification, Software Configuration Management Plan, Software Quality Assurance Plan, and tool plans (if applicable) may contain additional information related to test coverage.

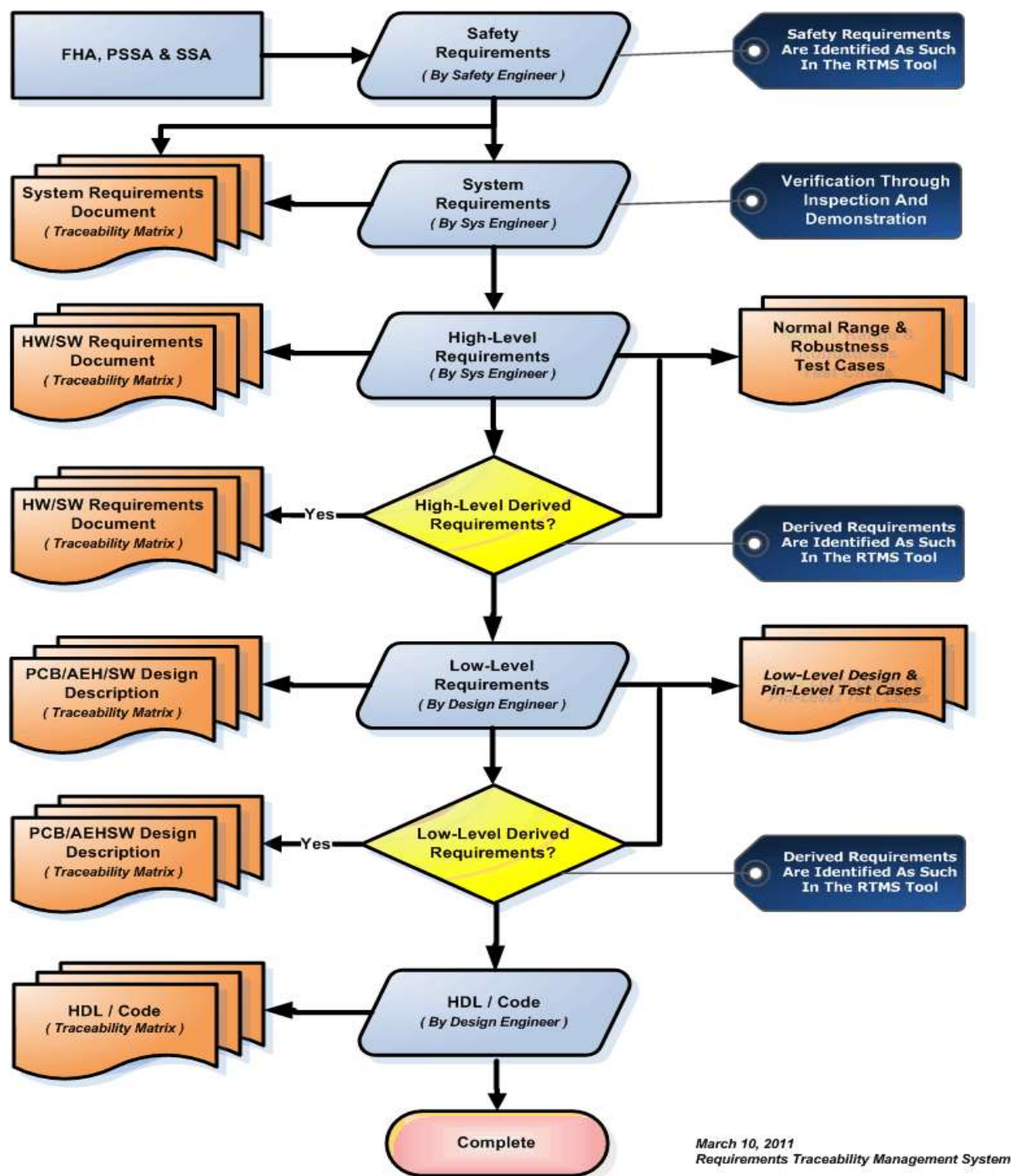
The following questions are considered when reviewing the plans:

- Are the plans sufficiently clear and detailed to allow the development and quality engineers to follow them consistently?
- Do the plans specify who is allowed to perform verification tasks?
- Do the plans specify how each requirement will be tested (e.g., module test, software integration, etc.)?
- Do the plans address all aspects of test coverage analysis? For example, are the following addressed:
  - tools and tool qualification, if tools are used for test coverage
  - the relationship between requirements-based testing and measuring test coverage
  - a process for determining when additional requirements-based tests should be added, if coverage is not achieved as expected
  - a procedure for regression analysis and testing, if necessary
  - the transition criteria to start and end test coverage
- Do the plans address the software change process for the airborne software?
- Do the plans address regression analysis and testing with respect to the unique requirements for test coverage?
- Do the plans address possible reuse of verification tools? For example, is credit being claimed from previous tool qualifications or will the tool qualification data be used in a future program?
- Is there evidence that the plans are being followed (such as, progress against timeframes, staffing, verification records, and SQE records)?

Testing is a method as well as a process, similar to the development processes. The Software Testing Process invokes the integral processes of Verification, Configuration Management, Quality Assurance and Certification Liaison. As such, it will be identified in this Software Verification Plan as both a method and a process, detailing the reviews and analysis that occur during the Software Testing Process.

## 4.2 Analysis of Outputs Methods

The analysis of outputs methods are specific to each analysis being performed. The following diagram, and subsequent paragraphs, details the methods which will be used for each analysis performed as part of the software verification process.

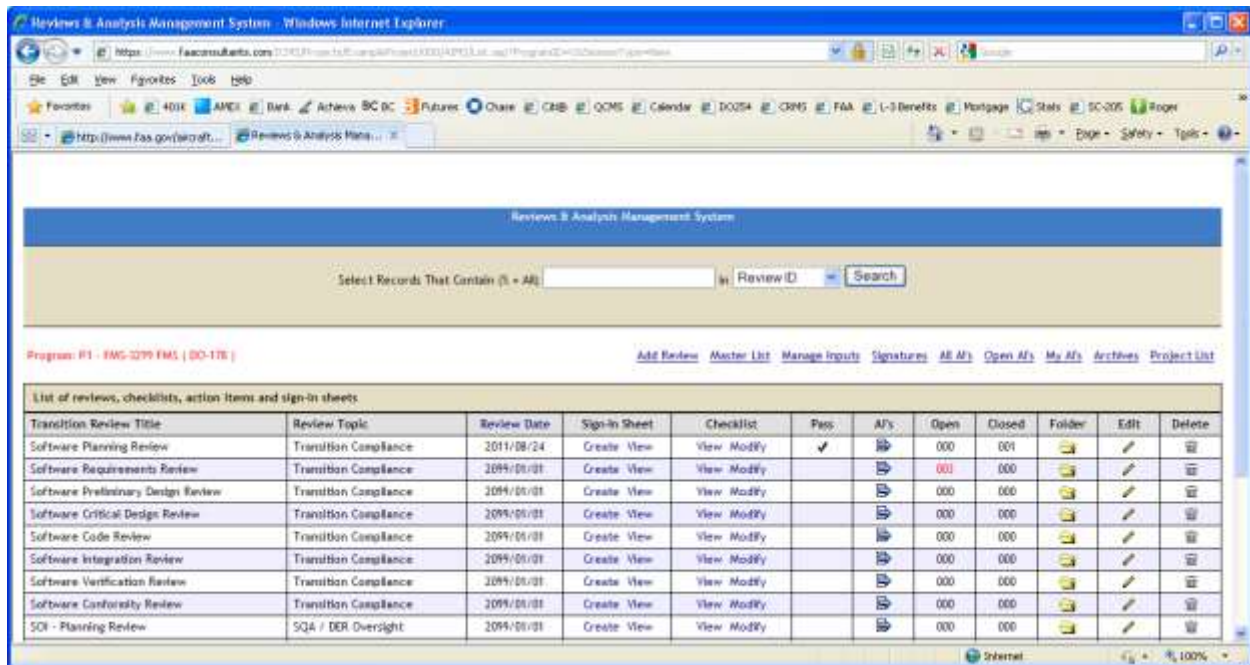


#### 4.2.1 Traceability of Reviews and Analysis Results

Traceability between review artifacts and review and analysis results will be facilitated by applying a unique ID (including the Project ID and Review ID) to each review activity via the Reviews and Analysis Management System. The review item, review sign-in sheet, review checklist and related action items will be traceable to each other based in this ID. In addition, a review folder which contains the review artifacts is maintained using the same ID, ensuring complete traceability and archiving. A final review of all verification evidence and related traceability will be conducted as part of the Software Conformity Reviews.

Review Artifacts	Unique ID <Project> - <Review>	Item
Review Item (What was reviewed)	RI: 01-025	-01, 02, 03, etc.
Review Sign-In Sheet (Who reviewed it)	SS: 01-025	
Review Checklist (Review criteria)	RC: 01-025	
Action Items (Review effectiveness)	AI: 01-025	-01, 02, 03, etc.

#### Reviews and Analysis Management System



#### 4.2.2 Transition Review Planning

Transition reviews will be held for entry into each of the Software Requirements, Software Design, Software Coding and Integration Processes as well as the Software Verification, Software Configuration Management, Software Quality Assurance and Certification Liaison Integral Processes. The number of transition reviews will be based on the number of iterations through the lifecycle and the number of times a process has been re-entered.

The following model illustrates three spirals and a final integration spiral. In all cases, planning is done first and integration is done together. In addition, the final transition criteria where final credit is provided occurs during the final integration spiral. Partial (P) transition criteria are provided for each spiral. Partial transition criteria includes the following as a minimum:

1. The Configuration Item being transitioned is under CC1 control
2. The Configuration Item has been peer reviewed (see Peer Review Planning below)
3. Peer review verification results are under CC2 control
4. SQA has verified that the planned partial transition criteria has been satisfied

Process	PLN	REQ	DES	COD	INT	VER	SCM	SQA	SQA
Spiral 1	✓	P	P	P	✓	P	P	P	P
Spiral 2		P	P	P		P	P	P	P
Spiral 3		P	P	P		P	P	P	P
Integration Spiral		✓	✓	✓		✓	✓	✓	✓

#### 4.2.3 Peer Review Planning

Peer reviews will be held for each of the main functional components identified during the initial analysis of system requirements allocated to software. At least one peer review will be held for each main function for High-Level Requirements (HLR), Architecture (ARCH), Code (CODE), Integration (INT), Test Cases (TC), Test Procedures (TP), Test Results, including Structural Coverage Analysis Results and Results Resolution (TR).

Review Title	HLR	ARCH	LLR	CODE	INT	TC	TP	TR
Main Function 1	✓	✓	✓	✓	✓	✓	✓	✓
Main Function 2	✓	✓	✓	✓	✓	✓	✓	✓
Main Function 3	✓	✓	✓	✓	✓	✓	✓	✓
Functional Interface	✓	✓	✓	✓	✓	✓	✓	✓

### 4.3 Software Planning Process Verification Methods

#### 4.3.1 Planning Process Verification Activities

DO-178C Activities	DO-178C Table Reference	DO-178C Paragraph Reference
Methods are chosen that enable the objectives of DO-178C to be satisfied.	NA	4.6a
Software life cycle processes can be applied consistently.	NA	4.6b
Each process produces evidence that its outputs can be traced to their activity and inputs, showing the degree of independence of the activity, the environment, and the methods to be used.	NA	4.6c
The outputs of the software planning process are consistent and comply with section 11 of DO-178C.	NA	4.6d

#### 4.3.2 Software Planning Process Inputs

Software Planning Process inputs to the Software Verification Process include the Software Verification Plan.

#### 4.3.3 Software Planning Process Reviews and Analysis

##### 4.3.3.1 Software Verification Plan Review

Review of the Software Verification Plan occurs when the document is mature enough to be reviewed. Once prepared, the Software Verification Plan is submitted to Software Configuration Management and entered into the document control system.

The Software Quality Assurance Engineer coordinates the document review process using the Document Review Management System. Each reviewer adds his or her comments in the Document Review Management System. A cycle of comment incorporation and re-review occurs through Configuration Management until all comments are closed. The Project Lead is responsible for closing all document comments prior to formal release.

Once all comments have been closed, the Software Verification Plan is reviewed by the Software Quality Assurance Engineer against the Document Review Checklist and a cross references from each section of the Software Verification Plan to the DO-178C Section 11 Objective to ensure that full compliance is achieved (See sample screenshot below). Once complete, the Software Quality Assurance Engineer signs and dates the checklists, which is maintained by Software Configuration Management as CC2 compliance evidence. The Software Verification Plan is then signed and released.

### Sample Screen Shot: Action Item Detail

The screenshot displays a web browser window titled "Action Item Management - Microsoft Internet Explorer". The address bar shows "https://www.faaconsultants.com". The main content area is titled "Document Review Checklist" and "Software Verification Plan". It contains the following fields:

- ItemNumber: 800-SVP-01
- Program: FMS-3299 Flight Management System
- Evaluation Date: 12/31/2099
- Evaluator / Title: Team Member, Principal Software QA Engineer

Below these fields is a section titled "Checklist Items". The first item is "CI 01:" with a status of "OK" (indicated by a green checkmark) and "NOK" (indicated by a red X). The description for CI 01 is: "The Organization section includes organizational responsibilities within the software verification process and interfaces with the other software life cycle processes." At the bottom, there are fields for "Objective: 11.3a" and "Doc Ref: 2.1".

#### 4.3.3.2 Software Planning Review

The Software Planning Process concludes with a Software Planning Review conducted by the Project Engineer. Transition Criteria from the Software Planning Process to the Software Development Process are discussed at this review.

When the Software Planning Review is held, action items are recorded in the Reviews and Analysis Management System database file associated with that review. The review includes a discussion of the status of the development and integral activities, a review and status of the Planning Documents, and a discussion of any special considerations. The Software Quality Assurance representative steps through the Software Planning Review Checklist. If deficiencies are revealed during the review, action items are generated, and corrective actions to resolve the deficiencies are fed back into the appropriate process.

Lifecycle data to be considered at the Software Planning Review include the following:

- Review and approval of the Plan for Software Aspects of Certification
- Review and approval of the Software Development Plan
- Review and approval of the Software Verification Plan
- Review and approval of the Software Configuration Management Plan
- Review and approval of the Software Quality Assurance Plan
- Review and approval of the Software Requirements Standards
- Review and approval of the Software Design Standards
- Review and approval of the Software Code Standards

A review checklist is used to identify the Review Inputs, Objectives and Activities that must be satisfied in order to transition to the next process (See Checklist Below).

### Sample Screen Shot: Transition Review Checklist

**Action Item Management - Microsoft Internet Explorer**

File Edit View Favorites Tools Help

Back Search Favorites Sign In Address <https://www.faaconsultants.com> Go

RI 09: Software Quality Assurance Plan (CC2)  
Reference: 11.5

RI 10: Software Requirements Standards (CC2)  
Reference: 11.6

RI 11: Software Design Standards (CC2)  
Reference: 11.7

RI 12: Software Code Standards (CC2)  
Reference: 11.8

RI 13: Software Configuration Management Records (CC2)  
Reference: 11.18

RI 14: Software Quality Assurance Records (CC2)  
Reference: 11.19

**Checklist Items**

CI 01: ☐ OK ☐ NOK

The activities of the software development processes and integral processes of the software life cycle that will address the system requirements and software level(s) are defined.

Reference: A-1.1 ( 4.1a, 4.3 )

NA

Done Internet

#### 4.4 Software Requirements Process Verification Methods

##### 4.4.1 Software Requirements Process Verification Objectives

DO-178C Objectives	DO-178C Table Reference	DO-178C Paragraph Reference
Software high-level requirements comply with system requirements.	A-3.1	6.3.1a
High-level requirements are accurate and consistent.	A-3.2	6.3.1b
High-level requirements are compatible with target computer.	A-3.3	6.3.1c
High-level requirements are verifiable.	A-3.4	6.3.1d
High-level requirements conform to standards.	A-3.5	6.3.1e
High-level requirements are traceable to system requirements.	A-3.6	6.3.1f
Algorithms are accurate.	A-3.7	6.3.1g

##### 4.4.2 Software Requirements Process Inputs

Software Requirements Process inputs to the Software Verification Process include the system requirements, high-level software requirements and traceability data.

##### 4.4.3 Transition Criteria for Entering The Verification of Requirements Process

This section includes the conditions necessary to consider the verification closed and successful for the Planning Process which establishes the transition criteria required for entering the Verification of Requirements Process.

- Planning documents are correct, released and under the applicable CC control
- Planning document checklists are complete and are under CC2 control
- Standards checklists are complete and are under CC2 control
- Document comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control
- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required and under CC2 control
- SQA review results have been produced and are under CC2 control
- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

#### 4.4.4 Software Requirements Process Reviews and Analysis

Prior to development of the Software Requirements Document and formal release of the software high-level requirements, peer reviews are held to review and analyze the high-level software requirements to determine if they are compliant with the criteria detailed in the Software Requirements Standards and correctly implement the system requirements. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System.

##### Sample Screen Shot: QCMS Peer Review

The screenshot shows a web browser window titled "Action Item Management - Microsoft Internet Explorer". The address bar displays "https://www.faaconsultants.com". The main content area contains a "Review Criteria Checklist" for "Level C Criticality". Below this, there is a section for "Peer Review - C2 Requirements" with the following details:

- Checklist ID: 0
- Program: FMS-3299 Flight Management System
- Review Topic: Review C2
- Evaluation Date: Friday, September 26, 2008
- Life Cycle Phase: Planning
- Evaluator / Title: Team Member, Quality Engineer

Below these details, there are sections for "Review Inputs", "Checklist Items", and "Meeting Notes". The "Review Inputs" section shows "RI 00: No Review Items Defined (CC-)" and "Reference: NA". The "Checklist Items" and "Meeting Notes" sections are currently empty. At the bottom, there is a text area for "Exit Criteria:" with the text "Compliance with system requirements: Ensure".

During the Software Requirements Process, the high-level software requirements and related traceability data are reviewed and analyzed by the project team based on the objectives identified in the Software Requirements Document Checklist and Software Requirements Review Checklist. The project team participants involved in this peer review must include the signature authority; that is it must include those individuals who are responsible for the release approval of the finalized document. The review checklists are contained in the Software Verification Plan.

The verification review comments for the Software Requirements Document are placed in the Document Review Management System and managed by the Project Engineer through closure. Upon acceptance, each team member signs the document, acknowledging acceptance. The document is then released and controlled through the SCM Process. The released document is provided upon request to the Certification Authority for review. Comments provided by the Certification Authority are added to the Document Review Management System and managed to closure. Re-verification of the Planning Documents occurs until final acceptance by the Certification Authority is reached. On-going change control and change authorization is provided through the SCM Process.

Upon final acceptance of the Software Requirements Documents by the development team and Certification Authority, a formal Software Requirements Review is conducted. The Requirements Review is used to demonstrate that all outstanding issues have been addressed to closure and that the established transition compliance criteria have been satisfied. Final transition criteria assurance and acceptance is obtained by Software Quality Assurance, in conjunction with the Certification Authority.

#### 4.4.4.1 Software Requirements Document Review

Review of the Software Requirements Document occurs when the document is mature enough to be reviewed. Once prepared, the Software Requirements Document is submitted to Software Configuration Management and entered into the document control system.

The software requirements document review is part of the process for developing and verifying the written form of the software requirements for release and further use in the project.

The Software Quality Assurance Engineer coordinates the document review process using the Document Review Management System. Each reviewer adds his or her comments in the Document Review Management System. A cycle of comment incorporation and re-review occurs through Configuration Management until all comments are closed. The Project Lead is responsible for closing all document comments prior to formal release.

Once all comments have been closed, the Software Requirements Document is reviewed by the Software Quality Assurance Engineer against the Document Review Checklist and a cross references from each section of the Software Requirements Document to the DO-178C Section 11 Objective to ensure that full compliance is achieved (See screenshot below). Once complete, the Software Quality Assurance Engineer signs and dates the checklists, which is maintained by Software Configuration Management as CC2 compliance evidence. The Software Requirements Document is then signed and released.

#### 4.4.4.2 Software Requirements Review

The Software Requirements Review follows the Software Requirements Definition Process. The Project Engineer conducts the Software Requirements Review using the Software Review Checklist as an aid. When the Software Requirements Review is held, the Project Engineer records the minutes or assigns someone to do so. The minutes include a discussion of the results, agreements and disagreements reached during the review, updates to the project schedule, resource estimates, and action item assignments with estimated completion dates.

The software requirements review is used to show completion of the software requirements definition process.

The review is conducted to demonstrate compliance with the objectives of the Software Requirements Process. Members of the project team, which includes the Project Engineer, Software Engineer assigned to the project, Hardware Engineer, Software Quality Assurance Engineer, and the Configuration Management representative, will be present at the review. Other concerned individuals, such as Manufacturing Test Engineering, Business Development or Sales, may be invited. The review will include a presentation of the naming conventions used for the software requirements and a review and discussion of each software requirement.

The objective of the Software Requirements Review is to detect and report errors that may have been introduced during the Software Requirements Definition Process.

- The review ensures that the system functions to be performed by the software are completely defined, that the performance and safety requirements have been correctly reflected in the software requirements, and that justification is provided for any derived requirements.
- The review ensures that each requirement is accurate, unambiguous, and sufficiently detailed, and that the requirements do not conflict with each other.
- The review confirms that no conflicts exist between the high-level requirements and the hardware features of the target system. Special attention is given to the use of system resources, system response times, and input/output hardware.
- The review ensures that each requirement can be verified.

The Software Quality Engineer steps through the Software Review Checklist. If deficiencies are revealed during the review, action items are generated, and corrective actions to resolve the deficiencies are fed back into the appropriate process.

Items to be considered at the Software Requirements Review include, but are not limited to, the following:

- Review and approval of the Software Requirements Document
- Review and acceptance of all functional requirements, performance requirements, interface requirements and design constraints.
- High-level and Derived requirements are recorded in the Software Requirements Document.
- The top-level software design is documented in the preliminary release of the Software Design Description.
- The Software Requirements Process includes a Software Preliminary Design Review.

Following the review, the program proceeds to the Software Design process.

#### 4.4.4.3 Analysis of High-Level Software Requirements

The software requirements review is used to show completion of the software requirements definition process.

Peer reviews are conducted to analyze the high-level requirements. The following characteristics are evaluated and form the exit criteria for the peer review:

- Compliance with system requirements: The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the software high-level requirements, and that derived requirements of the system are satisfied by the software high-level requirements, and that derived requirements and the reason for their existence are correctly defined.
- Accuracy and consistency: The objective is to ensure that each high-level requirement is accurate, unambiguous, and sufficiently detailed, and that the requirements do not conflict with each other.
- Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the system features of the target computer, especially, system response times and input/output hardware.
- Verifiability: The objective is to ensure that each high-level requirement can be verified.
- Conformance to standards: The objective is to ensure the Software Requirements Standards were followed during the software requirements process and that deviations from the standards are justified.
- Traceability: The objective is to ensure that the functional, performance, and safety-related requirements of the system that are allocated to software were developed into the software high-level requirements.
- Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

#### 4.4.4.4 System and Software Requirements Trace Analysis

The results of this analysis are contained in the system requirements trace matrix. This requirements trace matrix is constructed as follows:

- The requirement identifier for each system requirement allocated to software will be entered into one field of the matrix.
- The requirement identifier for each software requirement that satisfies the system requirement will be entered into the other field of the matrix.
- When multiple software requirements satisfy one system requirement, an entry with the duplicate system requirement identifier field will be entered.
- When multiple system requirements are satisfied by one software requirement, an entry with the duplicate software requirement identifier field will be entered.
- All software requirements derived due to implementation will be designated as "Derived" in the system requirement identifier field.

## 4.5 Software Design Process Verification Methods

### 4.5.1 Software Design Process Verification Objectives

DO-178C Objectives	DO-178C Table Reference	DO-178C Paragraph Reference
Low-level requirements comply with high-level requirements.	A-4.1	6.3.2a
Low-level requirements are accurate and consistent.	A-4.2	6.3.2b
Low-level requirements are compatible with target computer.	A-4.3	6.3.2c
Low-level requirements are verifiable.	A-4.4	6.3.2d
Low-level requirements conform to standards.	A-4.5	6.3.2e
Low-level requirements are traceable to high-level requirements.	A-4.6	6.3.2f
Algorithms are accurate.	A-4.7	6.3.2g
Software architecture is compatible with high-level requirements.	A-4.8	6.3.3a
Software architecture is consistent.	A-4.9	6.3.3b
Software architecture is compatible with target computer.	A-4.10	6.3.3c
Software architecture is verifiable.	A-4.11	6.3.3d
Software architecture conforms to standards.	A-4.12	6.3.3e
Software partitioning integrity is confirmed.	A-4.13	6.3.3f

### 4.5.2 Software Design Process Inputs

Software Design Process inputs to the Software Verification Process include the software architecture, low-level software requirements and traceability data.

### 4.5.3 Transition Criteria for Entering The Verification of Design Process

This section includes the conditions necessary to consider the verification closed and successful for the Requirements Process which establishes the transition criteria required for entering the Verification of Design Process.

- Requirements document is correct, released and under the applicable CC control
- Requirements document checklists are complete and are under CC2 control
- Document comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control

- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required and under CC2 control
- SQA review results have been produced and are under CC2 control
- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

### 4.5.4 Software Design Process Reviews and Analysis

Prior to development of the Software Design Description and formal release of the software architecture and software low-level requirements, peer reviews are held to review and analyze the proposed architecture and software low-level requirements to determine if they are compliant with the criteria detailed in the Software Requirements Standards and Software Design Standards and correctly implement the high-level software requirements. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System.

During the Software Design Process, the software architecture, software low-level requirements and related traceability data are reviewed and analyzed by the project team based on the objectives identified in the Software Design Description Checklist and Software Preliminary Design Review and Software Critical Design Review Checklists. The project team participants involved in this peer review must include the signature authority; that is it must include those individuals who are responsible for the release approval of the finalized document. The review checklists are contained in the Software Verification Plan.

The Verification review comments for the Software Design Description are placed in the Document Review Management System and managed by the Project Engineer through closure. Upon acceptance, each team member signs the document, acknowledging acceptance. The document is then released and controlled through the SCM Process. The released document is provided to the Certification Authority for acceptance. Comments provided by the Certification Authority are added to the Document Review Management System and managed to closure. Re-verification of the Planning Documents occurs until final acceptance by the Certification Authority is reached. On-going change control and change authorization is provided through the SCM Process.

Upon final acceptance of the Software Design Descriptions by the development team and Certification Authority, formal Software Preliminary Design and Software Critical Design Reviews are conducted. These reviews are used to demonstrate that all outstanding issues have been addressed to closure and that the established transition compliance criteria have been satisfied. Final transition criteria assurance and acceptance is obtained by Software Quality Assurance, in conjunction with the Certification Authority.

#### 4.5.4.1 Software Design Description Review

Review of the Software Design Description occurs when the document is mature enough to be reviewed. Once prepared, the Software Design Description is submitted to Software Configuration Management and entered into the document control system.

The Software Quality Assurance Engineer coordinates the document review process using the Document Review Management System. Each reviewer adds his or her comments in the

Document Review Management System. A cycle of comment incorporation and re-review occurs through Configuration Management until all comments are closed. The Project Lead is responsible for closing all document comments prior to formal release.

Once all comments have been closed, the Software Design Description is reviewed by the Software Quality Assurance Engineer against the Document Review Checklist and a cross reference from each section of the Software Design Description to the DO-178C Section 11 Objective to ensure that full compliance is achieved (See screenshot below). Once complete, the Software Quality Assurance Engineer signs and dates the checklists, which is maintained by Software Configuration Management as CC2 compliance evidence. The Software Design Description is then signed and released.

#### 4.5.4.2 Software Preliminary Design Review

The Software Preliminary Design Review (PDR) follows the Software Requirements Review. The Software PDR Checklist is used during the review.

Representatives from Electrical, Mechanical, Software, Quality Assurance, Manufacturing Engineering, and Manufacturing Test Engineering are invited to the Software PDR.

The Project Engineer conducts the Software PDR. The review includes a presentation of the overall software design structure, module design structure, relationships of the design elements and modules, and rationale for the software design.

The interfaces between the software modules and interfaces between the software and hardware devices are presented and discussed.

If deficiencies are revealed during the review, action items are generated, and corrective actions to resolve the deficiencies are fed back into the appropriate process.

Where applicable, the following items considered at the PDR include:

- Software Architecture
- Rationale for the Software Design
- Review and approval of the Context Level Data Flow Diagram
- Review and approval of the Software Block Diagrams
- Review and approval of Control Flows and Data Flows
- Review and approval of State Transition Diagrams

#### 4.5.4.3 Software Critical Design Review

The Software Critical Design Review (CDR) is integrated into the Software Design process. The Software CDR Checklist will be used during the review.

The Project Engineer conducts the Software CDR. Representatives from Electrical, Software, Mechanical, Business Development, Quality Assurance, Manufacturing, and Test Departments are invited to the Software CDR.

The review includes a presentation by the software engineer of the overall detailed design structure, module design structure, relationships of the design elements and modules, and rationale for the software design.

The interfaces between the software modules and interfaces between the software and hardware devices will be presented and discussed.

The Software CDR ensures that the components of the software architecture are accurate and consistent. The review will confirm that no conflicts exist between the software architecture and the hardware features of the target system.

If deficiencies are revealed during the review, corrective actions to resolve the deficiencies are fed back into the appropriate engineering process or document. This may include the Business Development Requirements, Software Design Description, the Electrical Design, Mechanical Design, or the Software Design.

One or more of the following items will be considered at the Software CDR:

- Software requirements are complete
- The Software Detailed Design conforms to the requirements
- Review and approval of the Context Level Data Flow Diagram
- Review and approval of the Software Block Diagram
- Review and approval of Control Flow and Data Flow

When the Software CDR is held, the Project Engineer records the Action Items or assigns someone to do so. The Action Items include action item assignments.

#### 4.5.5 Reviews and Analysis of Software Architecture

Peer reviews are conducted to analyze the software architecture. The following characteristics are evaluated and form the exit criteria for the peer review:

- The compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.
- Consistency: The objective is to ensure that a correct relationship exists via data flow and control flow.
- Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization and interrupts, between the software architecture and the system features of the target computer.
- Verifiability: The objective is to ensure that the software architecture can be verified; there are no unbounded recursive algorithms, for example.
- Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, especially complexity restrictions and design constructs that would not comply with the system safety objectives.
- Partitioning integrity: The objective is to ensure that partitioning breaches are prevented.

#### 4.5.6 Reviews and Analysis of Low-Level Software Requirements

Peer reviews are conducted to analyze the low-level software requirements. The following characteristics are evaluated and form the exit criteria for the peer review:

- Compliance with high-level requirements: The objective is to ensure that the software low-level requirements satisfy the software high-level requirements and that derived requirements and the design basis for their existence are correctly defined.
- Accuracy and consistency: The objective is to ensure that each low-level requirement is accurate and unambiguous and that the low-level requirements do not conflict with each other.
- Compatibility with the target computer: The objective is to ensure that no conflicts exist between the software requirements and the system features of the target computer, especially, the use of resources (such as bus loading), system response times, and input/output hardware.
- Verifiability: The objective is to ensure that each low-level requirement can be verified.
- Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process, and that deviations from the standards are justified.
- Traceability: The objective is to ensure that the high-level requirements and derived requirements were developed into the low-level requirements.
- Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

#### 4.6 Software Coding Process Verification Methods

##### 4.6.1 Software Coding Process Verification Objectives

DO-178C Objectives	DO-178C Table Reference	DO-178C Paragraph Reference
Source Code complies with low-level requirements.	A-5.1	6.3.4a
Source Code complies with software architecture.	A-5.2	6.3.4b
Source Code is verifiable.	A-5.3	6.3.4c
Source Code conforms to standards.	A-5.4	6.3.4d
Source Code is traceable to low-level requirements.	A-5.5	6.3.4e
Source Code is accurate and consistent.	A-5.6	6.3.4f

##### 4.6.2 Software Verification Process Inputs

Software Coding Process inputs to the Software Verification Process include the software source code and traceability data.

##### 4.6.3 Transition Criteria for Entering The Verification of Software Coding Process

This section includes the conditions necessary to consider the verification closed and successful for the Design Process which establishes the transition criteria required for entering the Verification of Software Coding Process.

- Design document is correct, released and under the applicable CC control
- Design document checklists are complete and are under CC2 control
- Document comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control
- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required and under CC2 control
- SQA review results have been produced and are under CC2 control
- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

##### 4.6.4 Software Coding Process Reviews and Analysis

Throughout the coding process, peer reviews are held to review and analyze the source

code to determine that it is in compliance with the Software Design Standards and Software Coding Standards and correctly implements the low-level software requirements. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System.

#### 4.6.4.1 Source Code File Review

Review of the Source Code Files occurs when the code is mature enough to be reviewed. Once developed, the code is added to the Software Library and turned over to Software Configuration Management for control.

#### 4.6.4.2 Source Code Review

Following the peer reviews, where each of the source files has been reviewed and analyzed, a Software Code Review is conducted. The Software Code Review Checklist and the Software Verification Plan are used during the review to verify code completion and adherence to standards.

The Software Engineer and Independent Verification Engineer conduct the Software Code Review or assigns someone with the same authority to do so. Action items are recorded in the Reviews and Analysis Management System database file associated with that review. The Action Items include the action item assignments and space for the completion date.

Representatives from Software, Electrical and Mechanical Engineering, Quality Assurance, Manufacturing, and Test Departments are invited to the Software Code Review.

If deficiencies are revealed during the review, corrective actions to resolve the deficiencies are documented in the Reviews and Analysis Management System and fed back into the appropriate software development process.

The following items will be reviewed at the Software Code Review:

- Compliance with low level requirements
- Compliance with software architecture
- Verifiability
- Conformance to standards
- Traceability
- Accuracy and consistency

#### 4.6.5 Reviews and Analysis of Source Code

Peer reviews are conducted to analyze the source code. The following characteristics are evaluated and form the exit criteria for the peer review:

- Compliance with the low-level requirements: The objective is to ensure that the Source Code is accurate and complete with respect to the software low-level requirements, and that no Source Code implements an undocumented function.
- Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow defined in the software architecture.
- Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the source code does not have to be altered to test it.

Peer reviews are also conducted to verify conformance to standards. The intent is to ensure that the Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives. The following characteristics are evaluated and form the exit criteria for the peer review:

- Traceability: The objective is to ensure that the software low-level requirements were developed into Source Code.
- Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of uninitialized variables or constants, and data corruption due to task or interrupt conflicts.
  - NOTE: Worst-case execution timing is evaluated during code review by the examination of looping constructs for execution length, and by examining the code for timing related construction, including excessive call stack depth and successive or stacked interrupts.

#### 4.7 Integration Process Verification Methods

##### 4.7.1 Integration Process Verification Objectives

DO-178C Objectives	DO-178C Table Reference	DO-178C Paragraph Reference
Output of software integration process is complete and correct.	A-5.7	6.3.5
Parameter Data Item File is correct and complete	A-5.8	6.6a
Verification of Parameter Data Item File is achieved.	A-5.9	6.6b
Executable Object Code complies with high-level requirements.	A-6.1	6.4a
Executable Object Code is robust with high-level requirements.	A-6.2	6.4b
Executable Object Code complies with low-level requirements.	A-6.3	6.4c
Executable Object Code is robust with low-level requirements.	A-6.4	6.4d
Executable Object Code is compatible with target computer.	A-6.5	6.4e

##### 4.7.2 Integration Process Inputs

Software Integration Process inputs to the Software Verification Process include the Executable Object Code and traceability data.

##### 4.7.3 Transition Criteria for Entering The Verification of Integration Process

This section includes the conditions necessary to consider the verification closed and successful for the Software Coding Process which establishes the transition criteria required for entering the Verification of Integration Process.

- Source Code is correct, released and under the applicable CC control
- Code review checklists are complete and are under CC2 control
- Comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control
- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required and under CC2 control
- SQA review results have been produced and are under CC2 control

- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

#### 4.7.4 Integration Process Reviews and Analysis

During the Integration Process, a peer review is held to review and analyze the Executable Object Code to determine that it is compatible with the target computer. Issues addressed at the peer reviews include, but may not be limited to, incorrect hardware addresses; memory overlays and missing software components. If deactivated code is integrated into the Executable Object Code, these peer reviews produce the evidence that the deactivated code will remain deactivated during normal operation. Objective evidence is also produced which addresses analysis of the verification activities that need to occur when unintended activation occurs due to abnormal conditions. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System.

##### 4.7.4.1 Executable Object Code Handling

When the source code and object code are mature enough to be built and installed on the target computer, the Executable Object Code is entered into configuration management. Once built and target compatibility is assured, the Executable Object Code is added to the Software Library and turned over to Software Configuration Management for control. This is a prerequisite for formal review of the integration data.

##### 4.7.4.2 System Integration Review

The System Integration Review is conducted at the conclusion of the System Integration process. The System Integration Review Checklist will be used during the review.

The Project Engineer conducts the System Integration Review. When the System Integration Review is held, the Project Engineer records the minutes or assigns someone to do so. The minutes will include a discussion of the results, agreements and disagreements reached during the review, updates to the project schedule and resource estimates, and action item assignments with estimated completion dates.

Representatives from Quality Assurance, Test Engineering, Manufacturing Engineering, Mechanical Engineering, and Software Engineering are invited to the System Integration Review.

The review ensures the results of the integration process are complete and correct. If deficiencies are revealed during the review, corrective actions to resolve the deficiencies are fed back into the appropriate process.

The system integration review is used to show the satisfaction of transition criteria from the integration process to the follow-on verification processes.

##### 4.7.4.3 Reviews and Analysis of Executable Object Code

Peer reviews are conducted to analyze the Executable Object Code. The following characteristics are evaluated and form the exit criteria for the peer review:

- Proper resource usage.
- Incorrect hardware addresses.
- Memory overlaps and Missing software components.

This summary review is used to determine that the software integration process

completeness criteria have been evaluated and found to be satisfied during the Executable Object Code reviews and System Integration Review.

## 4.8 Software Testing Process Verification Methods

### 4.8.1 Software Testing Process Verification Objectives

DO-178C Objectives	DO-178C Table Reference	DO-178C Paragraph Reference
Test procedures are correct.	A-7.1	6.4.5b
Test results are correct and discrepancies explained.	A-7.2	6.4.5c
Test coverage of high-level requirements is achieved.	A-7.3	6.4.4a
Test coverage of low-level requirements is achieved.	A-7.4	6.4.4b
Test coverage of software structure (modified condition/decision) is achieved.	A-7.5	6.4.4c
Test coverage of software structure (decision coverage) is achieved.	A-7.6	6.4.4c
Test coverage of software structure (statement coverage) is achieved.	A-7.7	6.4.4c
Test coverage of software structure (data coupling and control coupling) is achieved.	A-7.8	6.4.4d
Verification of additional code, that cannot be traced to Source Code, is achieved.	A-7.9	6.4.4c

### 4.8.2 Software Testing Process Inputs

Software Testing Process inputs to the Software Verification Process include the test cases, test procedures, test results and traceability data.

### 4.8.3 Transition Criteria for Entering The Testing of Integration Process Outputs

This section includes the conditions necessary to consider the verification closed and successful for the Integration Process which establishes the transition criteria required for entering the Testing of Integration Process Outputs.

- EOC is correct, released and under the applicable CC control
- Integration review checklists are complete and are under CC2 control
- Comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control
- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required and under CC2 control

- SQA review results have been produced and are under CC2 control
- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

### 4.8.4 Transition Criteria for Entering The Verification of Verification Outputs

This section includes the conditions necessary to consider the verification closed and successful for the Verification Process.

- All verification evidence is correct, released and under the applicable CC control
- Verification review checklists are complete and are under CC2 control
- Comments have been implemented, verified and are under CC2 control
- Peer review checklists are complete and under CC2 control
- Transition review checklist is complete and under CC2 control
- Action items have been recorded, implemented, closed and under CC2 control
- Signature sheets have been produced and are under CC2 control
- Verification Independence has been shown where required, with records under CC2 control
- SQA review results have been produced and are under CC2 control
- Other artifacts (i.e., customer comments) are recorded and are under CC2 control

### 4.8.5 Software Testing Process Reviews and Analysis

Throughout the Software Testing Process, **peer** reviews are held to review and analyze the Test Cases and Test Procedures to determine that they are complete and fully verify the high-level and low-level requirements. In addition, peer reviews are used to brainstorm methods for robustness testing. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System. The resulting Robustness test cases are reviewed for their ability to reveal vulnerabilities in the software.

Throughout the Software Testing Process, **peer** reviews are held to review and analyze the Test Cases and Test Procedures to determine that they are complete comply with the Software Verification Plan and cover the software high-level and low-level requirements.

Peer reviews are held to analyze the coverage achieved as a result of requirements-based testing. Where code structures are not covered, an analysis is performed to determine the cause. If the cause is determined to be untraceable code as a result of dead code, the code is removed. If the cause is inadequate requirements or test cases and procedures, the peer review results include action items to resolve this. If the result is unreachable "required" code that is traceable, an analysis of each line of uncovered code is documented in the Structural Coverage Analysis Results document.

#### 4.8.5.1 Software Verification Cases and Procedures Document Review

Review of the Software Verification Cases and Procedures occurs when the document is mature enough to be reviewed. Once prepared, the Software Verification Cases and Procedures is submitted to Software Configuration Management and entered into the document control system.

The Software Quality Assurance Engineer coordinates the document review process using

the Document Review Management System. Each reviewer adds his or her comments in the Document Review Management System. A cycle of comment incorporation and re-review occurs through Configuration Management until all comments are closed. The Project Lead is responsible for closing all document comments prior to formal release.

Once all comments have been closed, the Software Verification Cases and Procedures is reviewed by the Software Quality Assurance Engineer against the Document Review Checklist and a cross reference from each section of the Software Verification Cases and Procedures to the DO-178C Section 11 Objective to ensure that full compliance is achieved. Once complete, the Software Quality Assurance Engineer signs and dates the checklists, which is maintained by Software Configuration Management as CC2 compliance evidence. The Software Verification Cases and Procedures is then signed and released.

#### 4.8.5.2 System Verification Review

The System Verification Review is conducted at the conclusion of the Software Testing Process. The System Verification Review Checklist will be used during the review.

The Project Engineer conducts the System Verification Review. When the System Verification Review is held, the Project Engineer records the minutes or assigns someone to do so. The minutes will include a discussion of the results, agreements and disagreements reached during the review, updates to the project schedule and resource estimates, and action item assignments with estimated completion dates.

Representatives from Quality Assurance, Test Engineering, Manufacturing Engineering, Mechanical Engineering, and Software Engineering are invited to the System Integration Review.

The review ensures the results of the integration process are complete and correct. If deficiencies are revealed during the review, corrective actions to resolve the deficiencies are fed back into the appropriate process.

#### 4.8.5.3 Reviews and Analysis of Test Cases, Test Procedures, and Results

Peer reviews are conducted to analyze Test Cases, Test Procedures, and Results. The following characteristics are evaluated and form the exit criteria for the peer review:

- Test cases: Independent verification of test cases is presented later in this document.
- Test procedures: The objective is to verify that the test cases were accurately developed into test procedures and expected results.
- Test results: The objective is to ensure that the test results are correct and that discrepancies between actual and expected results are explained.

During the verification process, the Independent Verification Engineer (along with select members of the development team) reviews the requirements-based test cases to assure that all requirements are adequately covered. If the requirements-based tests are not adequate to achieve test coverage, then additional requirements-based tests or analysis may be needed.

The following questions are considered when evaluating test cases and procedures:

- Do the test cases and procedures adhere to the relevant plans and standards?
- If plans or standards have not been followed, is there documented rationale for deviations from stated plans and standards?
- Is the rationale for each test case clearly explained?
- Are the test cases and procedures appropriately commented to allow future updates?
- Have the test cases and procedures been subjected to appropriate change and configuration control?
- Is the separation between test cases clear? For example, are test starts and stops identified?
- Do the test cases and procedures specify required input data and expected output data?
- Were the inputs for each test case derived from the requirements?
- Are the test cases and procedures sufficient to cover all the relevant requirements? That is, do the traceability matrices provide clear association between test cases and requirements?
- Are the test cases and procedures sufficient to achieve test coverage?
- Are sufficient tests identified to provide test coverage for each logic construct?
- Are there sufficient robustness test cases and procedures?
- Are test cases and procedures correct?

### 4.8.5.3.1 Review checklists for test cases, procedures, and results

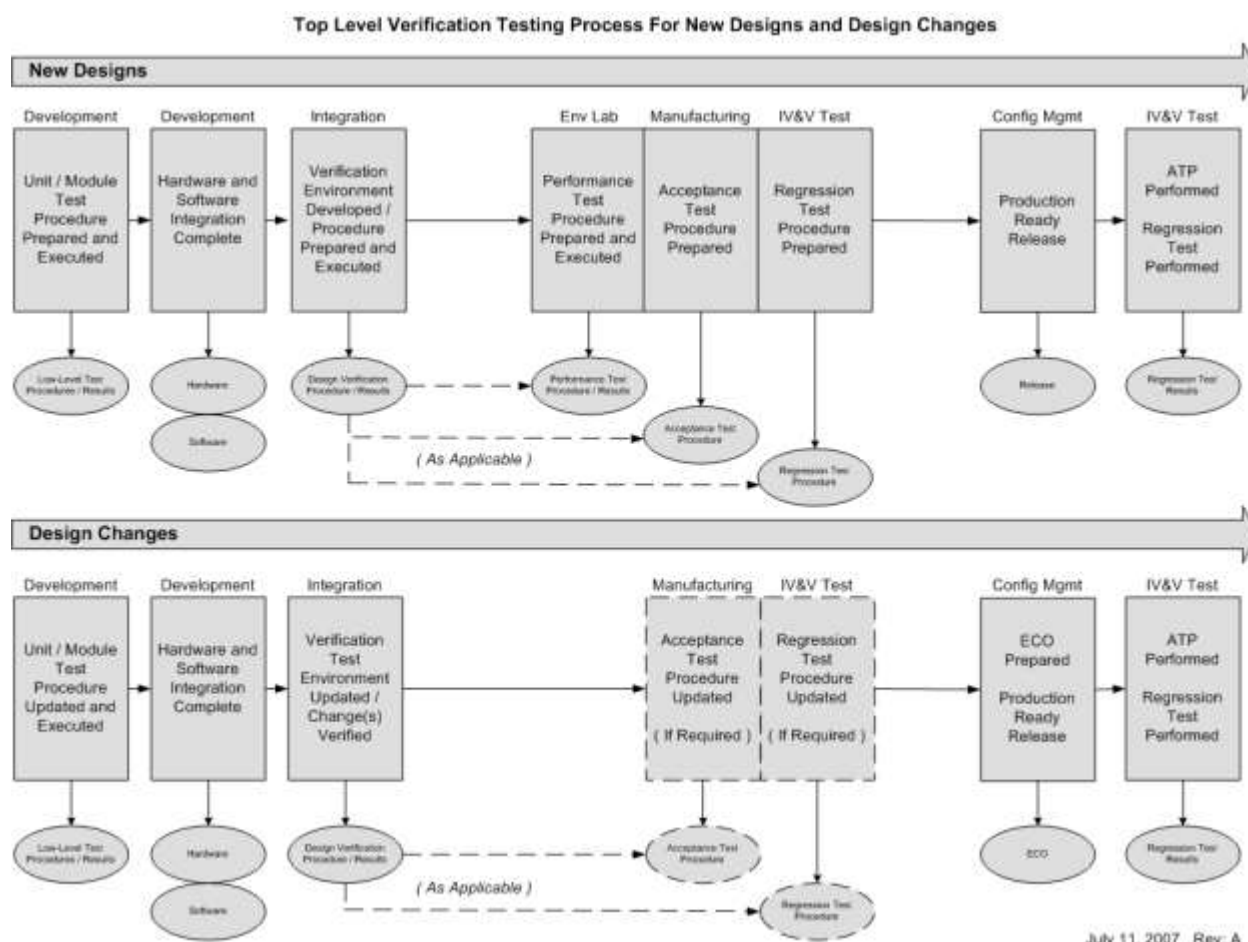
A checklist is used during review of test procedures and results (See Document Review Management System in the Software Quality Assurance Plan). During this review, the checklists themselves are assessed, considering the following questions for test coverage:

- Are the checklists sufficient to determine that the requirements-based test cases, procedures, and results meet the test coverage objective?
- Have the checklists been prepared and/or reviewed by quality?
- Do the checklists specify:
  - who performed the review?
  - what data was reviewed (with revision)?
  - when it was reviewed?
  - what was found?
  - what corrective actions were taken, if necessary?
- Do the checklists require evaluation of tolerances specified in the requirements?
- Do the checklists ensure that results of the test cases can be visually verified? (e.g., can the SQE, or other reviewer, visually determine when requirements-based tests have passed or failed?)
- Will the checklists reveal whether the results of the test cases that are counted for credit towards test coverage are observable?
- Will the checklists address limitations of the structural coverage analysis tool as documented in the tool qualification?
- Will the checklists reveal test cases that violate project standards?

## 4.8.6 Software Test Execution

Verification testing of software has two objectives. One objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors that could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed. The following three types of testing are used:

- Hardware Software Integration Testing: To verify correct operation of the software in the target computer environment.
- Software Integration Testing: To verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.
- Low-level Testing: To verify the implementation of software and low-level requirements.



**Peer** reviews are conducted to ensure that software testing objectives have been satisfied. The following characteristics are evaluated and form the exit criteria for the peer review:

- Test cases are developed based primarily on the software requirements.
- Test cases are developed to verify correct functionality and to establish conditions that reveal potential errors.
- Software requirements coverage and traceability analysis are used to determine what software requirements were not tested.
- Structural coverage analysis techniques are used to determine what software structures were not exercised (for Levels A, B, and C software).

#### 4.8.6.1 Test Environment

More than one test environment may be needed to satisfy the objectives for software testing. The test environment includes the (independently built) software loaded into the target computer and tested in the target computer environment.

*Note: This section must be tailored to specify the actual test environments in use.*

#### 4.8.6.2 Requirements-Based Test Cases

Requirements-based testing is emphasized because this strategy has been found to be the most effective at revealing errors. Requirements-based test case selection includes the following:

- Implementation of both normal range and robustness (abnormal range) test cases. The specific test cases should be developed from the software requirements and the error sources inherent in the software development process.

#### 4.8.6.3 Normal Range Test Cases

Normal Range test cases are developed to demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases include:

- Real and integer input variables are exercised using valid equivalence classes and boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code are performed to check the characteristics of the function in context.
- For state transitions, test cases are developed to exercise the transitions possible during normal operation.
- For software requirements expressed by logical equations, the normal range test cases verify the variable usage and the Boolean operators.

#### 4.8.6.4 Robustness Test Cases

Robustness test cases are developed to demonstrate the ability of the software to respond to abnormal inputs and conditions. Robustness test cases include:

- Real and integer variables are exercised using equivalence class selection of invalid values.
- System initialization is exercised during abnormal conditions.
- The possible failure modes of the incoming data are determined, especially complex, digital data strings from an external system.
- For loops where the loop count is a computed value, test cases may be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
- For time-related functions, such as filters, integrators, and delays, test cases may be developed for arithmetic overflow protection mechanisms.
- For state transitions, test cases may be developed to provoke transitions that are not allowed by the software requirements.

##### 4.8.6.4.1 Robustness Test Case Selection Strategy

The following criteria will be used, at a minimum, to select robustness test cases:

1. Starting with the Functional Hazard Assessment (FHA), produce each hazard conditions and verify expected result.
2. Test multiple combinations of hazard conditions. Combine hazards.
3. Identify all range / boundary requirements. Test outside the boundaries of each range requirement.
4. Identify all conditions where a fault is asserted. Test all of those conditions.
5. Combine fault conditions (without reset) and verify expected results.
6. Perform negative testing. If A AND B THEN X, test If NOT A and B THEN NOT X.
7. If not required by DO-160 Testing, test critical functionality and other key functionality over temperature.
8. Perform testing over non-standard electrical conditions (i.e., power glitching, power up, power down, brown out)

#### 4.8.6.5 Requirements-Based System Verification Testing Methods

Requirements-based system verification testing methods concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The objective of requirements-based testing is to ensure that the software in the target computer will satisfy the high-level requirements.

Typical errors revealed by this level of testing include:

- Incorrect input handling.
- Failure to satisfy execution requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.
- Errors in system interfaces.
- Incorrect behavior of feedback loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.

Peer reviews are conducted to ensure that common software errors were not introduced into the design. The review includes the focus on the following most common error conditions:

- Implementation Error Source (Data Bugs)
  - 1) Logic bugs → ( $x = 0$ ;  $x \leq 10$ ;  $x++$ ) → Expect a result of 10
  - 2) Parameter Passing → Incorrect arguments passed
  - 3) Return Codes → Unexpected return codes passed
  - 4) Math Overflow / Underflow → Exceeding integer value
  - 5) Logic Processing Error → Too many nested conditions or calculations
  - 6) Reentrance Problem → If a section of code can be interrupted before it completes its execution, and can be called again before the first execution has completed, the code must be designed to be reentrant. This typically requires that all variables referenced by the reentrant routine exist on the stack and not in static memory.
  - 7) Incorrect Control Flow → The intended sequence of operations can be corrupted by incorrectly designed conditional loops. This may cause problems such as missing execution paths, unreachable code, and incorrect control logic.
  - 8) Pointer Errors → Pointing to a NULL pointer in a linked list, improperly incrementing pointer used to step through look-up tables or lists, bad function pointers.
  - 9) Indexing Problems → Improper use of Index Registers in assembly language have similar problems to those identified with pointers. Provides the same type of indirection useful for table look-up, walking through lists, trees, and other data structures.
  - 10) Variable Scope Errors → Using the same name and applying it to different data items that exist in different scopes.
  - 11) Improper Data Usage → Using an uninitialized variable or using the same variable for more than one purpose.
  - 12) Incorrect Flag Usage → Flags are usually global in scope and are almost always static (stored in a fixed memory location). Flag may inadvertently be used for more than one purpose or used to indicate more than one condition. Every flag should be SET, CLEARED and tested at some point in the program.
  - 13) Incorrect Address → Usually the result of an incorrect pointer. It's possible to code a bad address into the code. This generally happens when the memory subsystem changes (i.e., Reduce memory size).
  - 14) Data / Range Overflow / Underflow → May result in passing a parameter that is out of bounds or storing a data type not large enough to hold the data.
  - 15) Signed / Unsigned data errors → Mixed sign arithmetic can easily lead to calculations that overflow the data types. Assembly languages have different branch instructions used after comparing signed and unsigned data. Using the wrong branch instruction may cause a critical error.
  - 16) Incorrect Conversion / Type-Casting / Scaling → Converting a data value from one representation to another is common and may cause bugs. Conversion from signed to unsigned or string to numeric type is common. Typecasts are useful to get data into whatever representation is needed, but circumvent compiler type-checking, increasing the risk of making a mistake.

17) Data Synchronization Errors → Embedded systems share data among separate threads of execution. An operation that uses a number of different data inputs must be synchronized in order to perform its processing. If the data values are updated asynchronously, the processing may be using some "new" data items with some "old" data items, and compute the wrong result.

➤ Implementation Error Source (Real-Time Bugs)

1) Interrupt Handling → It is critical to handle all interrupts that the system will ever receive. Receiving an unexpected interrupt without being able to handle it will likely cause failures.

2) Task Synchronization → Tasks must be synchronized correctly. One task may acquire raw data; another may process this data as a set; still another may make control decisions on the processed data values. Proper synchronization usually is implemented by relying on flags or semaphores to control task regular intervals.

➤ Implementation Error Source (System Bugs)

1) Stack Overflow / Underflow → Pushing more data into the stack than it can hold is referred to as a stack overflow. Pulling more data from the stack than was put on the stack is referred to as a stack underflow. Both result in using bad data and can cause an unintended jump to an arbitrary address, resulting in a failure.

2) Race Conditions → A race condition occurs when two or more independent threads each access the same resource at the same time. The effects of a race condition vary widely; they're dependent on the specifics of the situation.

3) Deadlock → When race conditions are avoided by "locking" a resource, preventing any other thread from accessing it, the design must be evaluated to ensure that deadlock will never occur. Testing for deadlock is generally ineffective, since only a particular order of resource locking may produce it, and that ordering may not result from the most common tests.

Deadlock is only a problem in multi-threading environments that lock resources. The following four conditions must be present in order for a deadlock to occur. Breaking any one of these conditions eliminates deadlock:

- a. Mutual exclusion—only one thread can use a locked resource at a time
- b. Nonpreemption—threads cannot force another thread to release a resource
- c. Hold-and-wait—threads hold resources that they have locked while waiting for any additional needed resources
- d. Circular wait—a circular chain of threads exist, such that each thread holds a resource needed by the next thread in the chain

4) Resource Sharing Problems → In the case where a peripheral such as an analog multiplexer may be used to direct one of a number of different inputs to a single A/D converter; If one task alters the mux setting to measure a given signal and another preempts it and sets the mux to pass a different signal, when control returns to the first task, it will be measuring the wrong signal, likely causing a failure condition.

- Implementation Error Source (Other Bugs)
  - 1) Syntax / Typing → Compilers do a good job of syntax checking; however, special attention needs to be placed on coding standards.
  - 2) Interface → Complex interfaces are a common source of failures. Interface problem may include incorrect EEPROM erase / write sequence, improper use of LCD controller chip commands, wrong sequence in reading / writing serial communication interface registers, etc.
  - 3) Memory Allocation / Deallocation → Using memory management routines can greatly simplify the efficient use of available memory. It can also be an added source of errors. For example, not checking for successful allocation before using the memory, not freeing memory when it is no longer needed (memory leak).
  - 4) Peripheral Register Initialization → Peripherals typically have different modes of operation, increasing the number of applications for which they're useful. This can complicate the initialization and use of these devices producing another source for errors.
  - 5) Watchdog Servicing → Watchdog timers help ensure that if something in the system goes exceptionally wrong, it will fail in a safe, or at least a predictable, manner. Servicing the watchdog timer must be done properly and at the right time. The watchdog must be enabled, and set to timeout at the correct interval.

#### 4.8.6.5.1 Requirements-Based Software Verification Testing

This testing method is used and concentrates on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. The objective of the requirements-based Software Verification Testing is to ensure that the software components interact correctly with each other and satisfy the software requirements through successive integration of code components with a corresponding expansion of the scope of the test cases.

Typical errors revealed by this testing method include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

#### 4.8.6.5.2 Requirements-Based Low-Level Testing

This testing method is used and concentrates on demonstrating that each software component complies with its low-level requirements. The objective of requirements-based low-level testing is to ensure that the software components satisfy their low-level requirements:

Typical errors revealed by this testing method include:

- Failure of an algorithm to satisfy a software requirement.
- Incorrect loop operations.
- Incorrect logic decisions.
- Failure to process correctly legitimate combinations of input conditions.
- Incorrect responses to missing or corrupted input data.
- Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
- Incorrect computation sequence.
- Inadequate algorithm precision, accuracy, or performance.

#### 4.8.7 Effectiveness of Test Program

The following tasks are performed to determine the effectiveness of the test program.

##### 4.8.7.1 Assess results of requirements-based tests

The first step after test execution is to determine whether all requirements-based tests pass. In addition to checking the final pass/fail results, the test cases and results for some randomly selected requirements should be examined to ensure that the results reflect the given inputs for those cases. Test results are also checked carefully with respect to any specified tolerances.

The following questions are considered to assess the requirements-based test results:

- Are the test result files clearly linked to the test procedures and codes?
- Are failed test cases obvious from the test results?
- Do the test results indicate whether each procedure passed or failed and the final pass/fail results?
- Do the test results adhere to the relevant plans, standards, and procedures?
- Have the test results been subjected to appropriate configuration control?

#### 4.8.7.2 Assess failure explanations and rework

Each failed test case is documented with an explanation for why it failed, including references to applicable Action Request. In some cases, rework of some life cycle data will be required; in other cases, only an explanation for the failed test cases is needed. If rework is required, the impact of changes should be carefully evaluated and the changed items should be subjected to the appropriate change and configuration control.

Once all rework is complete, test cases should be rerun in compliance with plans for regression testing. Note: there may be cases where failed requirements-based tests are acceptable; however, it is typical for them to be fixed and rerun.

The following questions are considered to assess failures and rework:

- Is there an acceptable rationale for deviations from expected results, standards, or plans?
- Are explanations for the failed test cases technically sound and accurate?
- Do explanations for failed test cases contain accurate references to relevant problem reports?
- Are explanations for code or test rework suitable to address the failure?
- Have test cases been re-executed in compliance with plans for regression testing?
- Have the test results from regression testing been documented appropriately?

#### 4.8.7.3 Assess coverage achievement

The Verification Engineer produces test cases that are expected to achieve 100% test coverage (i.e., the purpose of test documentation is to show compliance with all of the requirements). If all the requirements have been covered by tests without achieving full test coverage, dead code, unintended functionality, or incorrectly documented de-activated code may be indicated. It is the policy to remove all dead code.

The following questions are considered when assessing coverage achievement:

- Has the test coverage criteria been correctly applied?
- Is 100% structural coverage achieved through requirements-based testing?
- If 100% structural coverage is not achieved through requirements-based testing, is there an explanation detailing which parts of the code were not executed, and why? Have additional test cases been added?
- Are explanations for drops in coverage sufficiently detailed and acceptable?
- Are there problem reports associated with dead code?
- Has dead code been analyzed and/or removed?

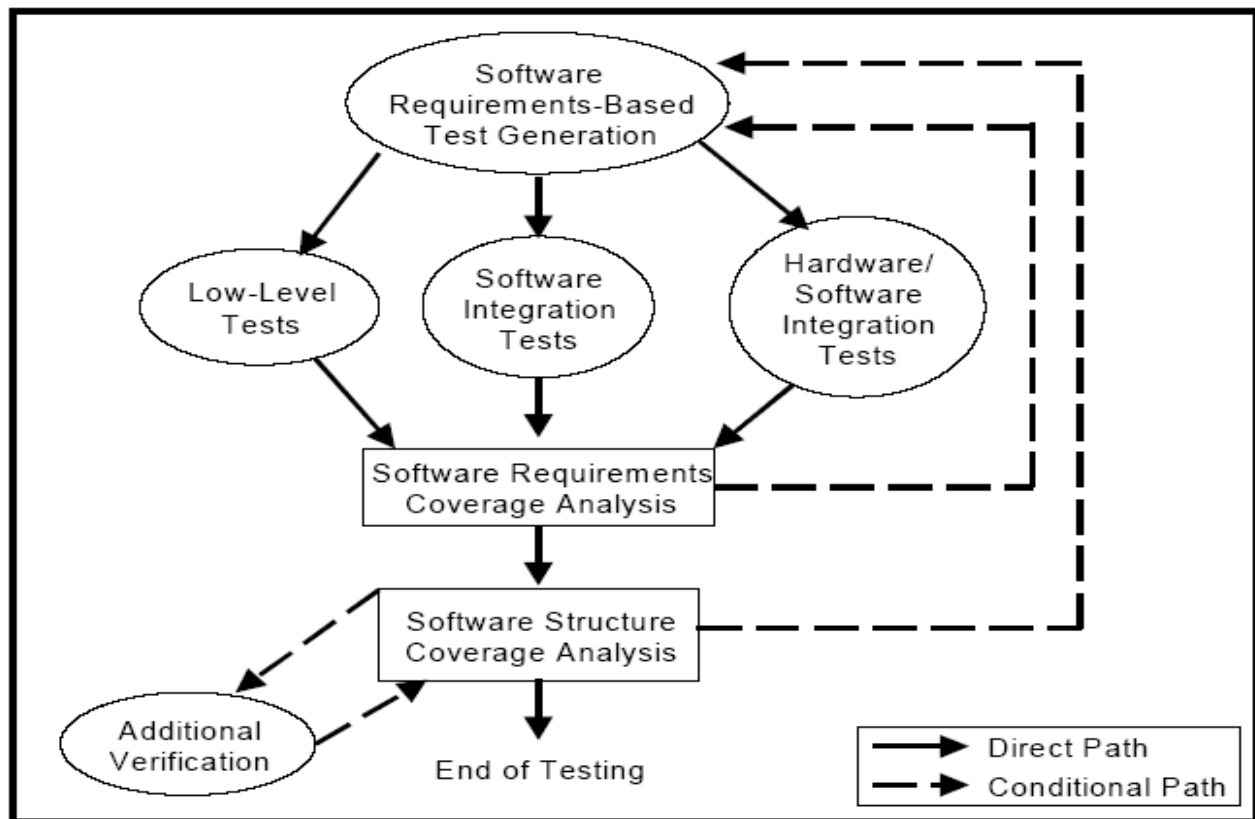
#### 4.9 Coverage Analysis Methods

The subsequent paragraphs detail the methods that will be used for coverage analysis as part of the software verification process.

Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage analysis measures will be applied to both requirements definitions and testing activities. Appropriate coverage measures will be used by SQA to audit verification activities. This will aid in determining the adequacy of the verification accomplished.

Coverage is viewed as a measure, not a method or a test. As such, results will be expressed as the percentage of an activity that is accomplished. Two specific measures of test coverage are identified in the following figure: requirements coverage and software structure coverage.

Requirements coverage analysis will be used to determine how well the requirements-based testing verifies the implementation of the software requirements and establishes traceability between the software requirements and the test cases. Structural coverage analysis will be used to determine how much of the code structure will be executed by the requirements-based tests and establishes traceability between the code structure and the test cases.



#### 4.9.1 Requirements Coverage Analysis

Each software requirement contains a finite list of behaviors and features, and each requirement is written to be verifiable. Testing based on requirements will be performed from the perspective of the user (providing a demonstration of intended function), and will provide a means for the development of test cases concurrently with development of the requirements.

Peer reviews will go beyond requirements coverage in evaluating the project. Reasons include:

- The software requirements and the design description (used as the basis for the test set) may not contain a complete and accurate specification of all the behavior represented in the executable code.
- The software requirements may not be written with sufficient granularity to assure that all the functional behaviors implemented in the source code are tested.
- Requirements-based testing *alone* cannot confirm that the code does not include unintended functionality.

In addition, software structure may be created that cannot be determined from top-level software specifications. Derived requirements, as described in DO-178C, will be used for this reason. Derived requirements will be tested as part of requirements-based testing.

#### 4.9.2 Structural Coverage Analysis

The purpose of structural coverage analysis with the associated structural coverage analysis resolution is to complement requirements-based testing as follows:

- Provide evidence that the code structure was verified to the degree required for the applicable software level.
- Provide a means to support demonstration of absence of unintended functions.
- Establish the thoroughness of requirements-based testing.

With respect to intended function, evidence that testing was rigorous and completed is provided by the combination of requirements-based testing (both normal range testing and robustness testing) and requirements-based test coverage analysis.

Requirements-based testing cannot completely provide this kind of evidence with respect to unintended functions. Code that is implemented without being linked to requirements may not be exercised by requirements-based tests. Such code could result in unintended functions. In this case, it will be designated this "Dead Code" or require that a requirement be written for the code. Should a new requirement be added, the applicable lifecycle artifacts (i.e., the Software Requirements Document) will be updated and the required processes will be repeated.

If requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there are no unintended functions. Structural coverage analysis will:

- Indicate to what extent the requirements-based test procedures exercise the code structure.
- Reveal code structure that was not exercised during testing.

Run-time libraries are subject to the same coverage requirements as the rest of the application code.

It should be noted that the structural coverage tools employed on the project must support resolution of overloaded operators and/or functions to the extent overloading is used on the project.

#### 4.9.2.1 Achieving Coverage

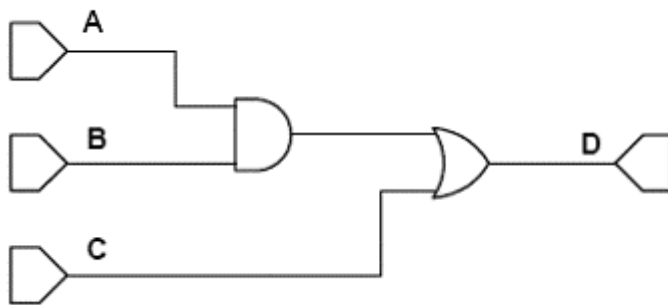
To achieve test coverage, a structural coverage analysis tool or a code instrumentation method will be used to monitor statements, entry and exit points, decision and branching statements, and Boolean conditions. Some tools do not support all of the coverage points required for test coverage. For example, not all structural coverage tools support coverage of entry and exit points. Such a tool can support part of the structural coverage analysis if other means are used to cover entry and exit points.

The structural coverage analysis tool will monitor a statement for multiple coverage points, as illustrated below:

Return (A and B) or C;

This statement will be monitored for the following coverage points:

- Statement—must be invoked at least once
- Exit Point—must be invoked at least once
- Decision—must take all possible outcomes (*false*, *true*) at least once



Test Case Number	1	2	3	4	5
Input A	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input B	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Input C	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output D	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

---

#### 4.9.2.2 Coverage Analysis Methods

A structural coverage analysis tool will be used to provide visibility into testing by either instrumenting code or providing other intervention techniques to gain visibility. The tool will be capable of instrumenting the code, provide flags, or other monitoring mechanisms to the original source code or object code. This enables the analysis tool to determine exactly what parts of the code are exercised. Once the code is instrumented, test cases are executed and the coverage analysis tool tracks which parts of the code are exercised by the test cases and, where complex analysis is required, how they are exercised. Pass/fail criteria for structural coverage are specified and tool analyzes the code against these criteria. If the pass/fail criteria are not specified, the tool will report the level of structural coverage the test cases achieve.

The Coverage Analysis Management System will be used to obtain both Statement and Decision Coverage.

#### Coverage Analysis Management System Screen Shot

Coverage Analysis Management System - Microsoft Internet Explorer provided by Roadrunner

http://www.faeconsultants.com/Projects/ExampleProject/000/CAMS/Just.asp?ProgramID=100&SessionType=New

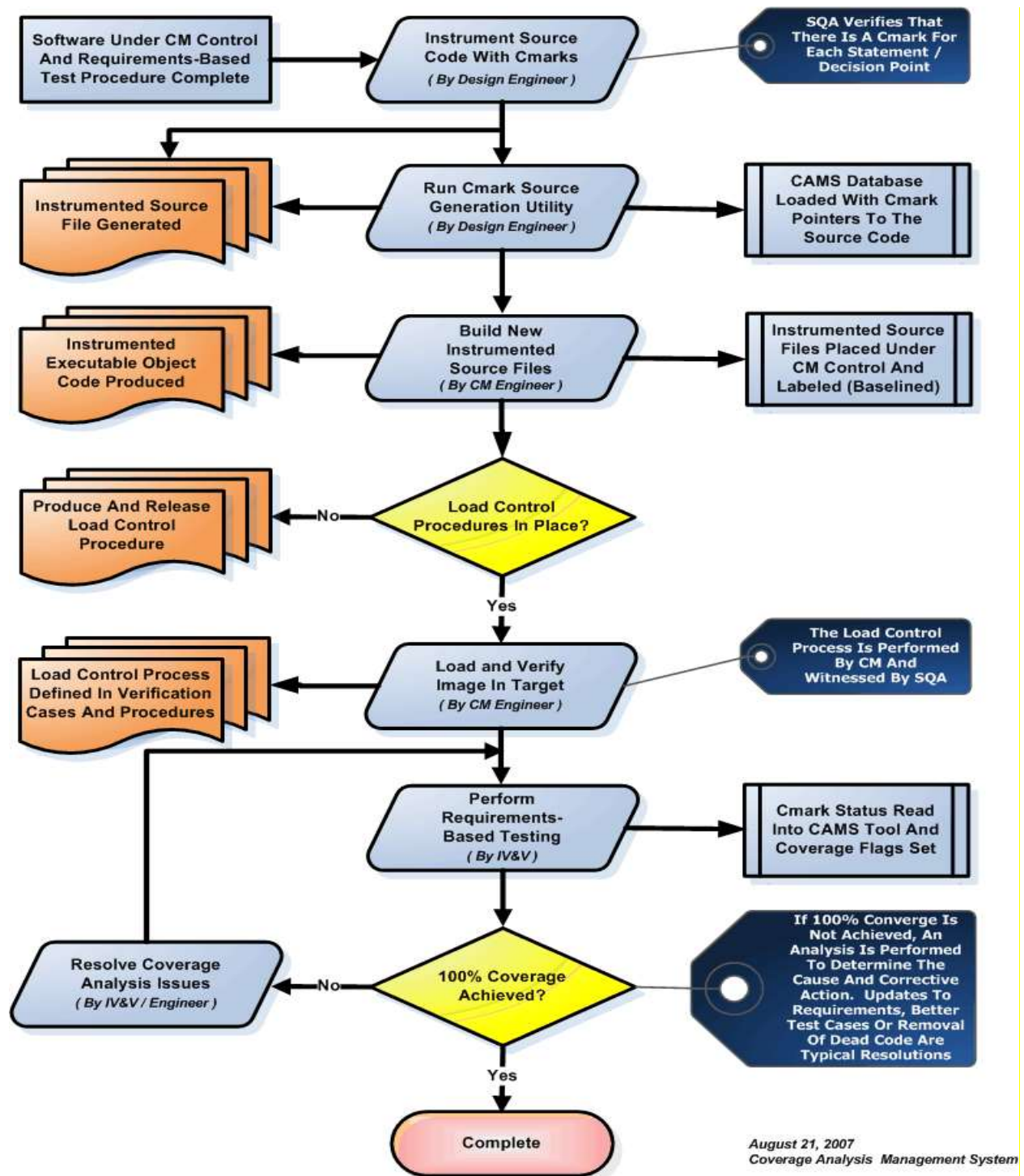
Select Software Release To Display:   Current Software Release Displayed

Program: Flight Management System

[Manage Source Files](#) [Instrument Source Code](#) [Perform Coverage Analysis](#) [Clear Database](#) [Home Page](#)

Release	File	Version	Line No	Function Name	Code Element	Marker	Coverage
1.000	XYZ.c	4.2	038	void DisplayFunction (void)	Simple Statement	CMARK(0)	✓
1.000	XYZ.c	4.2	044	void TestFunction (void)	Simple Statement	CMARK(1)	✓
1.000	XYZ.c	4.2	116	word ADC_GetAvg (byte Chan)	If Statement (1)	CMARK(2)	✓
1.000	XYZ.c	4.2	120	word ADC_GetAvg (byte Chan)	Simple Statement	CMARK(3)	✓
1.000	XYZ.c	4.2	151	static void ADC_RunThresh (void)	Simple Statement	CMARK(4)	✓
1.000	XYZ.c	4.2	159	static void ADC_RunThresh (void)	If Statement (1)	CMARK(5)	✓
1.000	XYZ.c	4.2	175	static void ADC_RunThresh (void)	Case Statement (1)	CMARK(6)	✓
1.000	XYZ.c	4.2	179	static void ADC_RunThresh (void)	If Statement (2)	CMARK(7)	✓
1.000	XYZ.c	4.2	196	static void ADC_RunThresh (void)	Case Statement (2)	CMARK(8)	✓
1.000	XYZ.c	4.2	200	static void ADC_RunThresh (void)	If Statement (3)	CMARK(9)	✓
1.000	XYZ.c	4.2	209	static void ADC_RunThresh (void)	Else If Statement (1)	CMARK(10)	✓
1.000	XYZ.c	4.2	222	static void ADC_RunThresh (void)	Case Statement (3)	CMARK(11)	✓
1.000	XYZ.c	4.2	226	static void ADC_RunThresh (void)	If Statement (4)	CMARK(12)	✓
1.000	XYZ.c	4.2	243	static void ADC_RunThresh (void)	Case Statement (4)	CMARK(13)	✓
1.000	XYZ.c	4.2	247	static void ADC_RunThresh (void)	Complex Path	CNOTE(0)	Analysis

The Coverage Analysis Management System process is as follows. Specifics of this process and this tool are described in the CAMS Tool Qualification Accomplishment Summary.



#### 4.9.2.3 Statement Coverage

To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. Achieving statement coverage shows that all code statements are reachable (in the context of DO-178C, reachable based on test cases developed from the requirements). Note that statement coverage is considered a weak criterion because it is insensitive to some control structures. Consider the following code segment:

```
If (  $x > 1$  ) and (  $y = 0$  ) then  $z := z / x$ ; end if;
```

By choosing  $x = 2$ ,  $y = 0$ , and  $z = 4$  as input to this code segment, every statement is executed at least once. However, if an “or” is coded by mistake (see code segment below) in the first statement instead of an “and”, the test case will not detect a problem. This makes sense because analysis of logic expressions is not part of the statement coverage criterion.

```
If (  $z = 2$  ) or (  $y > 1$  ) then  $z := z + 1$ ; end if;
```

#### 4.9.2.4 Modified Condition Decision Coverage

Decision coverage requires two test cases: one for a *true* outcome and another for a *false* outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. But, not all decisions are simple. For the decision (**A or B**), test cases (*TF*) and (*FF*) will toggle the decision outcome between *true* and *false*. However, the effect of **B** is not tested; that is, those test cases cannot distinguish between the decision (**A or B**) and the decision **A**.



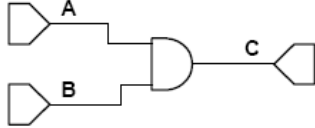
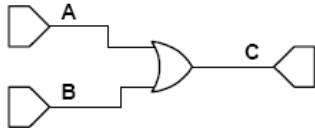
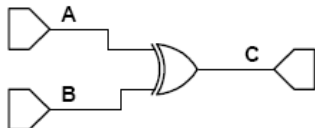
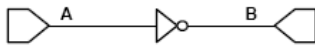
Condition coverage requires that each condition in a decision take on all possible outcomes at least once (to overcome the problem in the previous example), but does not require that the decision take on all possible outcomes at least once. In this case, for the decision (**A or B**) test cases (*TF*) and (*FT*) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two test cases is required for each decision.

Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between *true* and *false* and to toggle each condition value between *true* and *false*. Hence, a minimum of two test cases are necessary for each decision. Using the example (**A or B**), test cases (*TT*) and (*FF*) would meet the coverage requirement. However, these two tests do not distinguish the correct expression (**A or B**) from the expression **A** or from the expression **B** or from the expression (**A and B**).

MC/DC enhances the condition/decision coverage criterion by requiring that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, as will be discussed further in chapter 3, and, in general, a minimum of  $n+1$  test cases for a decision with  $n$  inputs. For the example (**A or B**), test cases (*TF*), (*FT*), and (*FF*) provide MC/DC. For decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the coverage measures discussed above.

Multiple Condition Coverage requires test cases that ensure each possible combination of inputs to a decision is executed at least once. Thus, multiple condition coverage requires exhaustive testing of the input combinations to a decision. In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with  $n$  inputs, multiple condition coverage requires 2 to the  $n$ th tests.

### Representations for Elementary Logical Expressions

Name	Schematic Representation	Code example	Truth Table															
Input																		
Output																		
and Gate		<b>C := A and B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>F</td></tr><tr><td>F</td><td>T</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	T	T	F	F	F	T	F	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	T																
T	F	F																
F	T	F																
F	F	F																
or Gate		<b>C := A or B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	T	T	F	T	F	T	T	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	T																
T	F	T																
F	T	T																
F	F	F																
xor Gate		<b>C := A xor B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>F</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	F	T	F	T	F	T	T	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	F																
T	F	T																
F	T	T																
F	F	F																
not Gate		<b>B := not A;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th></tr><tr><td>T</td><td>F</td></tr><tr><td>F</td><td>T</td></tr></table>	<u>A</u>	<u>B</u>	T	F	F	T									
<u>A</u>	<u>B</u>																	
T	F																	
F	T																	

### AND Gate

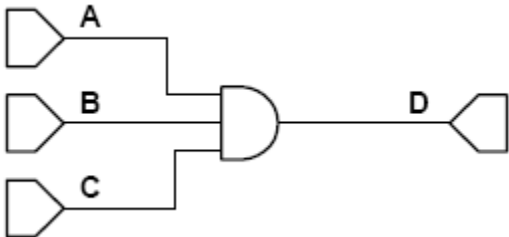
The following tests will be performed to achieve test coverage for an "and" gate:

- All inputs are set *true* with the output observed to be *true*. This requires one test case for each  $n$ -input "and" gate.
- Each and every input is set exclusively *false* with the output observed to be *false*. This requires  $n$  test cases for each  $n$ -input "and" gate.

Changing a single condition starting from a state where all inputs are *true* will change the outcome; that is, an "and" gate is sensitive to any *false* input. Hence, a specific set of  $n+1$  test cases is needed for an  $n$ -input "and" gate. These specific  $n+1$  test cases meet the intent of test coverage by demonstrating that the "and" gate is correctly implemented.

The following is an example of the minimum testing required for a three-input "and" gate. In this case, it takes four test cases to show that each input "independently" affects the output.

If ( A = 1 ) and ( B = 1 ) and ( C = 1 ) then D := 1; end if;



Test Case Number	1	2	3	4
Input A	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input B	<i>T</i>	<i>T</i>	<i>F</i>	<i>T</i>
Input C	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output D	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>

### OR Gate

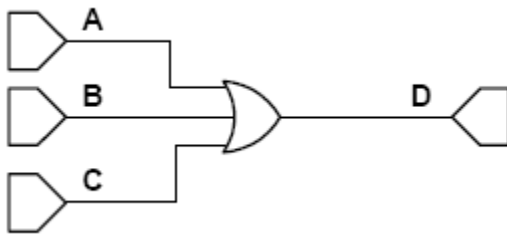
The following tests will be performed to achieve test coverage for an "or" gate:

- All inputs are set *false* with the output observed to be *false*. This requires one test case for each  $n$ -input "or" gate.
- Each and every input is set exclusively *true* with the output observed to be *true*. This requires  $n$  test cases for each  $n$ -input "or" gate.

These requirements are based on an "or" gate's sensitivity to a *true* input. Here again,  $n+1$  specific test cases are needed to test an  $n$ -input "or" gate. These specific  $n+1$  test cases meet the intent of test coverage by demonstrating that the "or" gate is correctly implemented.

The following is an example of the minimum testing required for a three-input "or" gate. In this case, it takes four test cases to show that each input "independently" affects the output.

If (  $A = 1$  ) or (  $B = 1$  ) or (  $C = 1$  ) then  $D := 1$ ; end if;



Test Case Number	1	2	3	4
Input A	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>
Input B	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
Input C	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>
Output D	<i>F</i>	<i>T</i>	<i>T</i>	<i>T</i>

### XOR Gate

The "**xor**" gate differs from both the "**and**" and the "**or**" gates with respect to test coverage in that there are multiple minimum test sets for an "**xor**". Consider the two-input "**xor**" gate. All of the possible test cases for this "**xor**" gate are shown below. For a two-input "**xor**" gate, any combination of three test cases will provide test coverage.

The following is an example of the minimum testing required for a two-input "**xor**" gate. Minimum testing to meet test coverage requires one of the following sets of test cases:

- test cases 1, 2, and 3
- test cases 1, 2, and 4
- test cases 1, 3, and 4
- test cases 2, 3, and 4

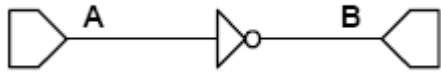
If ( A = 1 ) xor ( B = 1 ) then C := 1; end if;

Test Case Number	1	2	3	4
Input A	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>
Input B	<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i>
Output C	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

Note that for a test set to distinguish between an "**or**" and an "**xor**" gate it must contain test case 1. Test sets 1, 2, and 3 above can detect when an "**or**" is coded incorrectly for an "**xor**", and vice versa. While not explicitly required by test coverage, elimination of test set 4 as a valid test set is worth considering. Note also that minimum tests to achieve test coverage for an "**xor**" gate with more than two inputs are implementation dependent. Hence, no single set of rules applies universally to an "**xor**" gate with more than two inputs.

### Not Gate

The logical “**not**” works differently from the previous gates: the “**not**” works only on a single operand. That operand may be a single condition or a logical expression. But, with respect to a gate level representation, there is a single input to the “**not**” gate as shown below.



Minimum testing to achieve test coverage for a logical “**not**” requires the following:

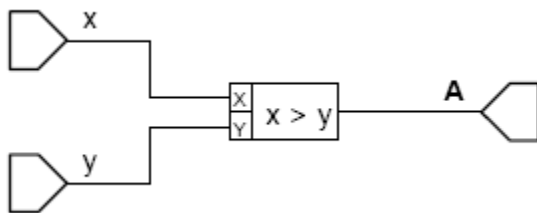
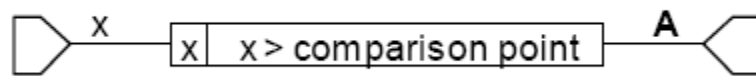
- The input is set *false* with the output observed to be *true*.
- The input is set *true* with the output observed to be *false*.

## Comparator

A comparator evaluates two numerical inputs and returns a Boolean based on the comparison criteria. Within the context of DO-178C, a comparator is a condition and also a simple decision. The following comparison criteria are considered in this tutorial:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to
- != not equal to

In general, the comparison point can be a constant or another variable.



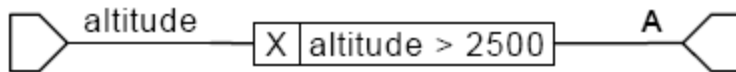
In either case, two test cases will be used to confirm test coverage for a comparator—one test case with a *true* outcome, and one test case with a *false* outcome. Minimum testing for a comparator requires the following:

- Input x set at a value above the comparison point (or y)
- Input x set at a value below the comparison point (or y)

Typically, three test cases will be used to assure that simple coding errors have not been made; that is, that the correct relational operator and comparison point are used in the code. So, while test coverage only requires two tests, minimum good requirements-based testing for a comparator requires:

- Input x set at a value slightly above the comparison point
- Input x set at a value slightly below the comparison point
- Input x set at a value equal to the comparison point

The definition of “slightly” is determined by engineering judgment based on the numerical resolution of the data type and/or target computer, the test equipment driving the inputs, and the resolution of the output device. Consider for example, the following set of test cases for a design that sets the output **A** *true* when altitude is greater than 2500.



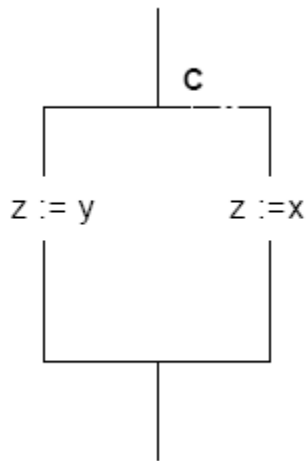
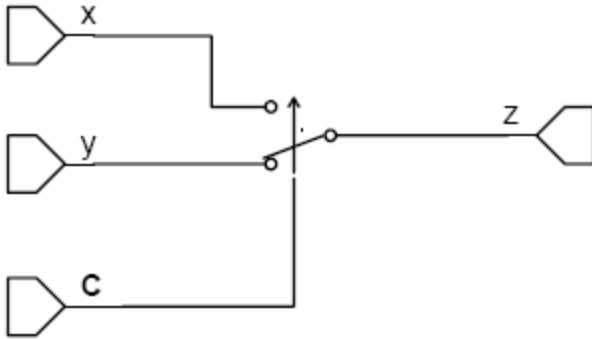
Test Case Number	1	2	3	4	5
Input altitude	25	32000	2500	2499	2501
Output A	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>

Test cases 1 and 2 give the desired test coverage output. However, those test cases do not confirm that the toggle occurred at 2500, and not elsewhere. Even adding test case 3 does not improve the test suite much. The design could have been implemented with a comparison point anywhere between 2501 and 32000, and give the same result for test cases 1, 2, and 3. Test cases 3, 4, and 5 are a better set, because this set confirms that the transition occurs at 2500.

**If Then Else:**

The *if-then-else* statement is a switch that controls the execution of the software. Consider the following example where  $x$ ,  $y$ , and  $z$  are integers and  $C$  is a Boolean:

If  $C$  then  $z := x$  else  $z := y$ ;



The following tests will be performed for the *if-then-else* statement:

- Inputs that force the execution of the *then* path (that is, the decision evaluates to *true*)
- Inputs that force the execution of the *else* path (that is, the decision evaluates to *false*)
- Inputs to exercise any logical gates in the decision

Note that the decision must evaluate to *false* with confirmation that the *then* path did not execute, even if there is no *else* path.

For example, for a single condition **Z**, the statement *if Z then...else...* requires only two test cases to achieve test coverage. The decision in *if X or Y or Z then... else...* requires four test cases to achieve test coverage.

A minimal test set for the statement *if Z then a := x else a := y* is shown in Table 9. Note that a *case* statement may be handled similarly to the *if-then-else* statement.

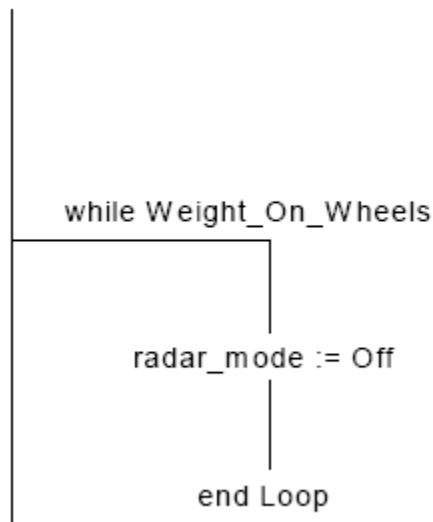
Test Case Number	1 Traverse the <i>then</i> path	2 Traverse the <i>else</i> path
Input x	12	18
Input y	50	34
Input <b>Z</b>	<i>T</i>	<i>F</i>
Output a	12	34

### While Loop:

Consider the following example where **Weight\_On\_Wheels** is a Boolean:

While **Weight\_On\_Wheels** loop radar\_mode := Off; end loop;

A schematic representation of this code is shown in Figure 10. In this case, **Weight\_On\_Wheels** is the decision for the *while loop* construct.



The following tests will be performed for the *while loop*:

- Inputs to force the execution of the statements in the loop (that is, the decision evaluates to *true*)
- Inputs to force the exit of the loop (that is, the decision evaluates to *false*)
- Inputs to exercise any logical gates in the decision

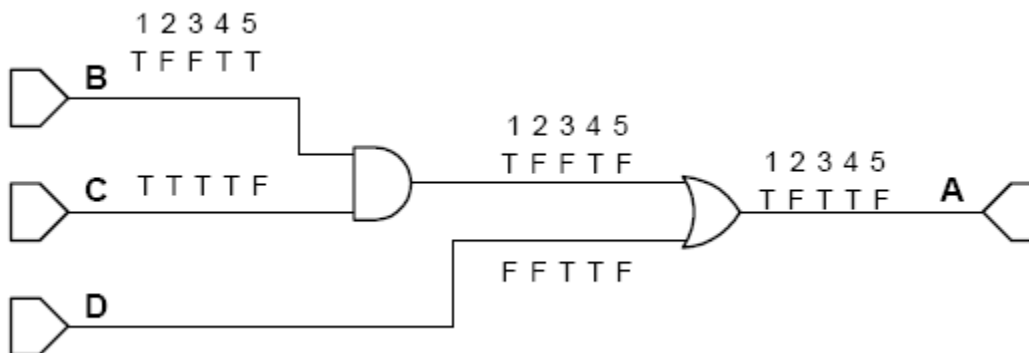
Two test cases may be used to achieve test coverage. One test case confirms that radar\_mode remains off as long as Weight\_On\_Wheels is true. The second test case confirms that radar\_mode could be set to something other than off when Weight\_On\_Wheels is false. In the case where Weight\_On\_Wheels is replaced by a Boolean expression, the Boolean expression would also need to be evaluated, and the setting of radar\_mode to off confirmed.

Applying Boolean Logic to Requirements-Based Testing

This process takes the inputs from the requirements-based test cases and maps them to the schematic representation. This provides a view of the test cases and the source code in a convenient format. Inputs and expected observable outputs for the requirements-based test cases for example 1 are given.

Test Case Number	1	2	3	4	5
Input <b>B</b>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input <b>C</b>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Input <b>D</b>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output <b>A</b>	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

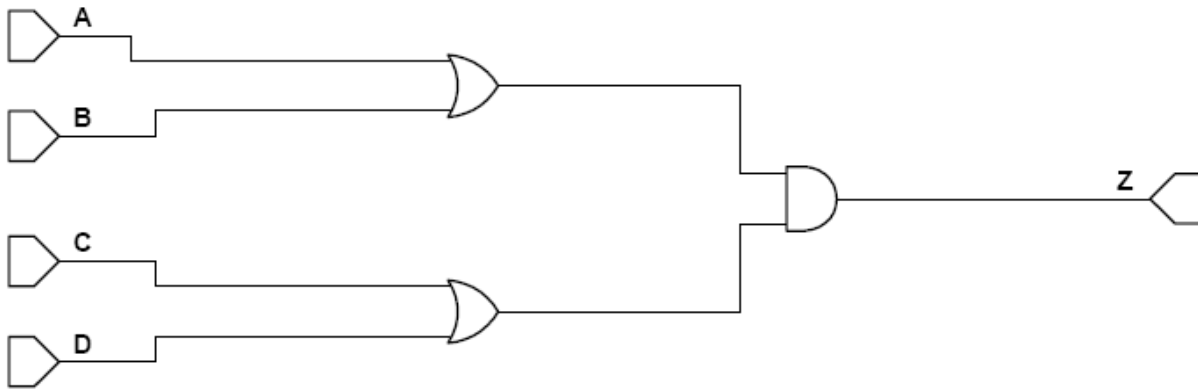
This example shows the test cases annotated on the schematic representation. Note that intermediate results are also determined from the test inputs and shown on the schematic representation.



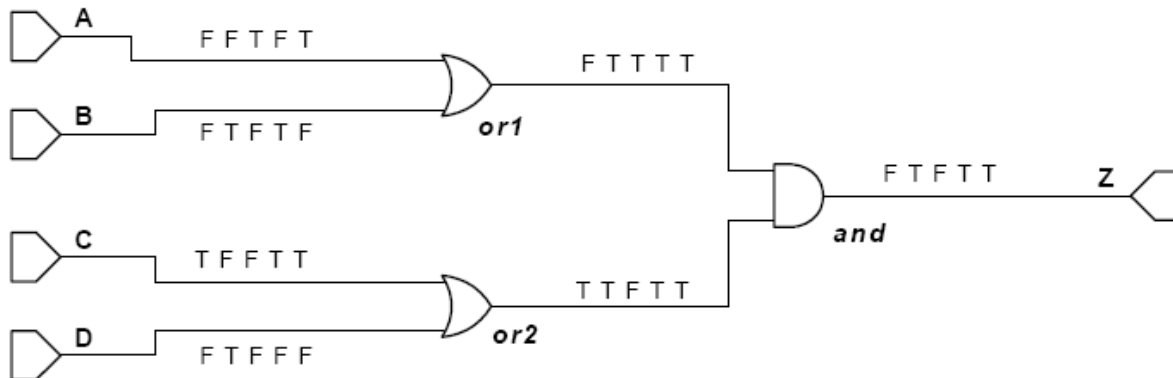
Knowing the intermediate results is important because some inputs may mask the effect of other inputs when two or more logic constructs are evaluated together. Test cases where the output is masked do not contribute to achieving test coverage. Using the annotated figure, the requirements-based tests cases that do not contribute (or count for credit) towards achieving test coverage can be identified. Once those test cases are eliminated from consideration, the remaining test cases can be compared to the building blocks to determine if they are sufficient to meet the test coverage criteria.

Expression:  $Z := (A \text{ or } B) \text{ and } (C \text{ or } D);$

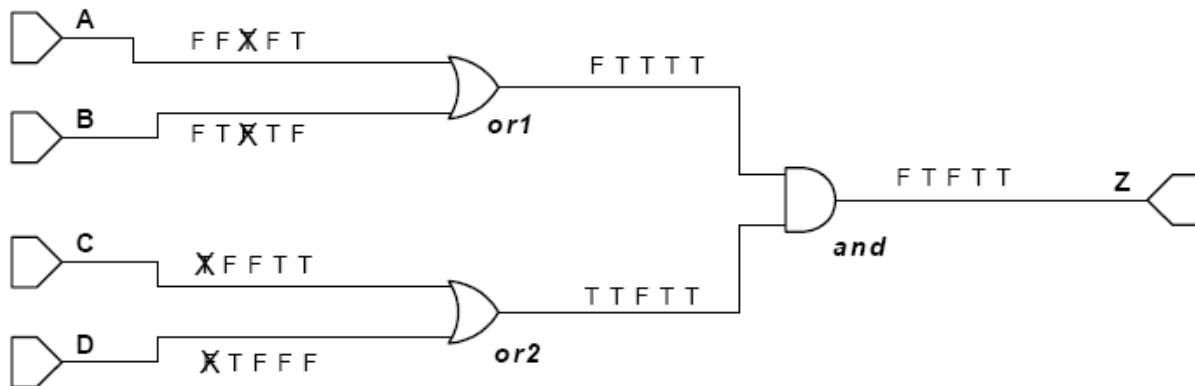
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture.



Step 3: Eliminate masked tests. In this case, any *false* input to the "and" gate will mask the other input. In this case, the *false* outcome of "or1" will mask test case 1 for the "or2" gate. Similarly, the *false* outcome of "or2" will mask test case 3 for the "or1" gate.


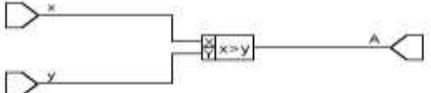
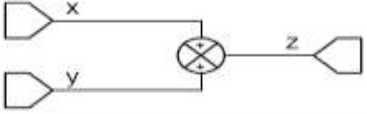
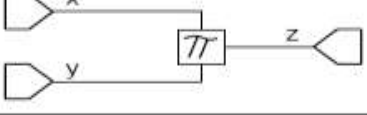
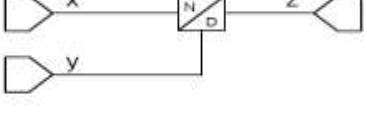
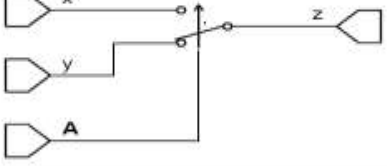
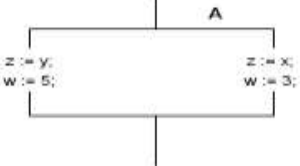
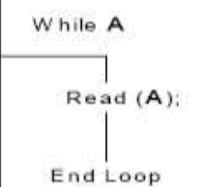


Step 4: Determine test coverage.

Gate	Valid Test Inputs	Missing Test Cases
<b>or1</b>	FF Case 1 FT Case 2 or 4 TF Case 5	None
<b>or2</b>	FF Case 3 FT Case 2 TF Case 4 or 5	None
<b>and</b>	TT Case 2, 4, or 5 TF Case 3 FT Case 1	None

Step 5: Confirm output. The outputs computed match those provided.

### Symbols for Source Code Representation

Name	Schematic Representation	Code example
Comparator (x with constant)		<b>A</b> := x > constant;
Comparator (x with y)		<b>A</b> := x > y;
Summer (addition or subtraction may be shown)		<b>z</b> := x + y;
Multiplier		<b>z</b> := x * y;
Divider		<b>z</b> := x / y;
If-then-else		If <b>A</b> then <b>z</b> := x; Else <b>z</b> := y; End if;
If-then-else		If <b>A</b> then <b>z</b> := x; <b>w</b> := 3; Else <b>z</b> := y; <b>w</b> := 5; End if;
While Loop		While <b>A</b> Loop Read ( <b>A</b> ); End loop;

#### 4.9.3 Data Coupling and Control Coupling Analysis

Analysis of data coupling and control coupling will be performed to ensure the adequacy of integration testing. This objective will be achieved in conjunction with hardware/software integration testing or software integration testing. A structural coverage analysis will be used to confirm that the requirements based testing has exercised the data and control coupling between code components. A separate peer review will be to verify that the source code matches the data flow and control flow defined in the software architecture.

##### Structural Coverage Analysis of Data and Control Coupling

Structural coverage analyses of data coupling and control coupling will be used to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies. The intent of this analysis is to show that the software modules/components affect one another in the ways in which the software designer intended and do not affect one another in ways in which they were not intended, thus resulting in unplanned, anomalous, or erroneous behavior. Typically, the measurements and assurance should be conducted on R-BT of the integrated components (that is, on the final software program build) in order to ensure that the interactions and dependencies are correct, the coverage is complete, and the objective is satisfied.

Satisfaction of this objective will be based on the detailed high and low level requirements of the modules/components' interfaces and the thorough requirements-based normal range and robustness tests of the software program. The interfaces and dependencies will be specified in the design requirements, and if those requirements are tested for both normal functioning and robustness. Satisfaction of the data and control coupling objective becomes a by-product of the design and verification processes.

The sections below identify the areas that are applicable and the means with which verification will occur.

##### 4.9.3.1 Data Coupling Analysis

Data coupling manifests as:

(1) Parameters passed to a function.

In the case of parameters passed to the function (case 1); statement coverage is sufficient to determine whether all control paths through the function that might be influenced by the parameter set have been exercised.

(2) Global data set or used by the function whose value is determined at compile-time or as part of system configuration.

In the case of global configuration data (case 2); analysis should determine the equivalency classes of all potential configurations. Structural coverage analysis should be executed under all equivalency classes.

(3) Global data set or used by the function which represents the current state of execution of the system.

In the case of global state data (case 3); analysis should determine the potential states (or their equivalency classes). Structural coverage analysis through instrumentation should determine if all states have been entered and all legal transitions between states have been exercised.

Note 1: Sub-functions exist where a function parameter determines which of multiple independent execution paths is taken through a function. Usually the parameter is used to determine which case of a large switch statement is executed.

### An Approach

Perform a review of the flight software to confirm data coupling and control coupling among the software components.

To satisfy the control coupling objective, use the structural coverage results to provide evidence that all functions were executed through high-level test cases. For functions that could not be exercised by high-level tests, develop additional functional analyses and add to the Software Verification Cases and Procedures (SVCP). The intent is to provide confidence that the requirements-based testing has completely exercised the code structure.

To satisfy the data coupling objective, this analysis includes functional parameters, global variables, external data, stored data, and resource contention. Analyze the SVCP and associated test code to confirm the verification coverage of the data coupling in the code. As with the control coupling, structural coverage results can be used to provide evidence that the data coupling through parameters was covered.

Although Certification Authority Software Team (CAST) Position Papers are not considered guidance, the approach outlined in CAST-19 is voluntarily adopted as a reasonable method for the demonstration of data coupling and control coupling coverage analysis. The table below examines the objectives of the data coupling coverage approach discussed in CAST-19.

<b>CAST-19 Objective</b>	<b>Where and how the objective is met</b>
Identify data dependencies.	This objective is met by defining the data items in the requirements and during the software requirements and code reviews ensuring proper setting and using of the data.
Identify inappropriate data dependencies.	This objective is met by the performance of the software requirements and code reviews.
Define and evaluate the extent of interface depth	This objective is met by the simplicity and small size of the project and verified by the code review.
Determine and minimize coupling interdependencies.	This objective is met by the simplicity and small size of the project. There will be no specific review test or analysis to verify this objective.
Evaluate accurate use of global data	This objective is met by code review and requirements base testing. The requirements based tests will ensure the software performs as required. The combination of these verifications adequately verifies the use of global data.
Evaluate input/output data buffers	This objective will be met by the accumulation of all the requirements based tests being executed, with passed results. The Software Verification Review checklist addresses this objective.

#### 4.9.3.2 Control Coupling Analysis

In the C language control coupling manifests in one of three ways:

(1) Static function calls.

In the case of static function calls (case 1); statement coverage is sufficient to determine if all possible calling points for a function have been executed by the test procedures.

(2) Sub-functions (See Note 1)

- In the case of sub-functions (case 2); analysis should reveal if the controlling parameter a constant determined at compile-time or whether the controlling parameter may be dynamically modified during execution.
- If the controlling parameter is a constant determined at compile-time, this case is equivalent to case 1.
- If the controlling parameter may be dynamically modified during execution, this case is equivalent to case 3.

(3) Dynamic function calls (i.e. function called through a pointer.)

- Points where a function is called through a pointer (case 3); it is necessary to determine whether (a) the function pointer has been initialized before use, (b) what the range of possible values for the function pointer is, and (c) that all possible values of the function pointer within that range have been executed.
- In the case of function pointers which belong to a jump table which is initialized at compile-time, this case is reduces to case 1.
- In the case of function pointers that are initialized at powerup, the calling point must be exercised in all potential configurations of the jump table. (Also see Data Coupling case 2.)

Although Certification Authority Software Team (CAST) Position Papers are not considered guidance, the approach outlined in CAST-19 is voluntarily adopted as a reasonable method for the demonstration of data coupling and control coupling coverage analysis. The table below examines the objectives of the control coupling coverage approach discussed in CAST-19.

CAST-19 Objective	Where and how the objective is met
Identify control dependencies.	This objective is met by defining the data items in the requirements and during the software requirements and code reviews ensuring proper setting and usage of the data
Identify inappropriate control dependencies.	Inappropriate control dependencies will be removed. This objective will be verified by the performance of the software requirements and code reviews.
Verify correct execution call sequence, including startup sequences.	This objective is met by reviewing the code against the requirements and by testing execution related requirements, with passed results.
Define and evaluate the extent of interface depth	This objective is met by the simplicity and small size of the project and by the code review.
Verifying scheduling	This objective is met by reviewing the code against the requirements and by testing execution related requirements, with passed results.
Worst-case execution time analysis	This analysis will be part of the Software Integration Analysis.

#### 4.10 Process-Specific Activities

The following sections detail the planned process-specific activities of the Testing Process.

##### 4.10.1 Test Case Development

- Test cases will be developed by a person other than the author of the software.
- Test case development can start after the software requirements have been formally reviewed. An iterative process for updating the test cases works in conjunction with any PRs processed to necessary changes in the software requirements.
- Test cases will be developed using software requirements, any certification document, as required for the function being tested, and information from the software detailed design that indicates additional boundary and robustness test steps are required. Additionally, test steps will be iteratively modified when preliminary coverage data is available to address any coverage deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.
- Test case tools will be chosen based on the verification needs identified. Software Simulation tools, specific lab equipment used in validation, and on-target testing tools (script processing tools, external interface stimulation tools) determine the specific steps developed. Refer to the PSAC for a list of verification tools.
- Test cases will be developed that capture test environment setup and parameters, versions of CM controlled Software, versions of CM controlled test documentation (including test cases) and industry interface ICDs for verification of external interfaces.
- Test cases will be developed based on functional interfaces and components. Where applicable, a test case may be used to verify multiple requirements concerning the same function or functions. The software trace matrix supports tracing from test case to software requirement. A test case may cover more than one software requirement, and the test case and trace matrix will indicate all software requirements covered during the test. Each instance of a core function must employ a separate test case with the appropriate tracing to the requirement. All iterative work in the lifecycle is completed using PRs and CM controls.
- Test cases will be developed to include positive path testing, plus additional testing as warranted for robustness. Robustness testing includes boundary conditions, obscure event mitigation, failure compensation, negative path testing, default case verification and more. Developed test cases indicate when test steps are for robustness testing, and may not trace to a specific software requirement. Additionally, test steps are iteratively modified when preliminary coverage data is available to address robustness deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.

### 4.10.2 Test Case Verification

Test cases will be formally reviewed by an independent party against the software requirements claimed in each test step. The trace matrix will be validated during the review to insure proper credit is taken for the software requirements listed. The software development life cycle steps will be followed to insure any discrepancies found in the review are addressed. All iterative work in the lifecycle will be completed using PRs and CM controls. Refer to Peer Review Process for the test cases.

### 4.10.3 Test Procedure Development

- Test procedures will be developed by a person other than the author of the Software.
- Test procedure development can start after the software requirements have been formally reviewed. Test procedures may be developed in conjunction with the test case. An iterative process for updating the test procedures works in conjunction with any PRs processed to necessary changes in the software requirements or related test cases.
- Test procedures will be developed using software requirements, test cases any certification documents as required for the function being tested, and information from the software detailed design that indicates additional boundary and robustness test steps are required. Additionally, test procedures are iteratively modified when preliminary coverage data is available to address any coverage deficiencies. All iterative work in the life cycle is completed using PRs and CM controls.
- Test tools will be chosen based on the verification needs identified.
- Test procedures will be developed using proven test templates that capture test environment setup and parameters, versions of CM controlled software, versions of CM controlled test documentation (including test cases) and industry interface ICDs for verification of external interfaces.
- When test procedure gaps are discovered during testing, the PR process will be used to address the gaps.
- Test procedures will be developed based on functional interfaces and components. Where applicable, a test procedure may be used to verify multiple requirements concerning the same function or functions. Test procedures will be tied directly to a test case – one for one. The software trace matrix will support tracing from test case to software requirement. The test procedure will be an integral part of the test case trace. As discrepancies in test procedures are identified, iterative changes will be made as necessary to resolve the discrepancy. All iterative work in the lifecycle will be completed using PRs and CM controls.

- Test procedures will be developed to include positive path testing, plus additional testing as warranted for robustness. Robustness testing will include boundary conditions, obscure event mitigation, failure compensation, negative path testing, default case verification and more. Test procedures will be developed to indicate when test steps are for robustness testing, and may not trace to a specific software requirement. Additionally, test procedures will be iteratively modified when preliminary coverage data is available to address robustness deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.

### 4.10.4 Test Procedure Verification

Test procedures will be formally reviewed by an independent party against the respective test case and software requirements claimed in each test step. The trace matrix will be validated during the review to insure proper credit is taken for the software requirements listed. The software development life cycle steps will be followed to insure any discrepancies found in the review are addressed. All iterative work in the lifecycle will be completed using PRs and CM controls. Refer to Peer Review Process of test cases.

### 4.10.5 Coverage Analysis Verification

Structural coverage analysis results will be formally reviewed by an independent party. Where code structures are not covered by requirements-based testing, the review will ensure that:

- If the uncovered code is dead code (extraneous code for which no requirements exist), it is removed.
- If the uncovered code contains intended functionality not included in the requirements and/or design, additional requirements (and related test procedures) are added to address the undocumented functionality.
- If the uncovered code is deactivated code used only in certain configurations of the airborne product, the code is reachable in the appropriate configuration of the product, and that the deactivation mechanism prevents inadvertent activation of the code.
- If the uncovered code is deactivated code not used in any approved configuration (such as test related code), the deactivated code structure is specifically identified in the Coverage Results and that the behavior of the code structure is deterministic and would not cause unintended behavior (determined by analysis).

#### 4.10.6 Testing Environment

- Each test case will include the following information:
  - Test Description
  - Tester Name
  - Test Date
  - Software Version tested
  - Test Method used
  - Tool(s) Version(s) used (if applicable)
- If appropriate (i.e., special equipment required) the test procedure will describe the specific bench configuration, test tool configuration, and any special instruction required to insure the tester sets up the correct environment.
- If appropriate (i.e., conformed unit, or special test rig) the test procedure will describe the following to insure the proper equipment and rig configuration is achieved before testing
  - P/N of test unit
  - S/N of test unit
  - Identification of special test rig components and gear
- SQA person will audit the test setup before testing.
- Once a test rig or environment has been conformed, the apparatus will be "Locked Down" for the time required to complete the test procedure. ("Locked Down" means the equipment and test gear involved in the test setup is physically or electronically secured from other personnel changing the environment.)

#### 4.10.7 Test Execution

- On-Target testing consists of normal system level test such as TSO, normal flight test simulation and DO-178C requirements based test. Additionally, special test cases will be created to exercise areas of the software where normal system level tests do not obtain full coverage, or configured options on the standard product may not be enabled. All system level testing will be identified in the software trace matrix for evaluation and review.
- Specific test procedures will be designed to exercise timing interfaces, critical data functions and configured options. Validation of the software at the low level will be achieved by capturing artifacts using lab equipment with electronic output. These resultant artifacts will be formally reviewed by an independent source and put under CM control. Data from these tests will also be used in verification by analysis efforts as required based on total test coverage analysis.

- Each test case will include the following test run information:
  - Test Description
  - Tester Name
  - Test Date
  - software Version tested
  - Test Method used
  - Tool(s) Version(s) used (if applicable)
- Testing will commence once the following are complete:
  - All software requirements are reviewed and under CM control with no outstanding (non-deferrable) PRs
  - All Test Cases/Procedures are reviewed and under CM control with no outstanding (non-deferrable) PRs
  - All software source files are reviewed and under CM control with no outstanding (non-deferrable) PRs

### 4.10.8 Test Results Verification

All traceability data is reviewed and under CM control with no outstanding (non-deferrable) PRs Software Testing Process Reviews and Analysis

Throughout the Software Testing Process, **peer** reviews are held to review and analyze the Test Cases and Test Procedures to determine that they are complete and fully verify the high-level and low-level requirements. In addition, peer reviews are used to brainstorm methods for robustness testing. Peer review entry and exit criteria, along with signature sheets and action items are recorded in the Reviews and Analysis Management System. The resulting Robustness test cases are reviewed for their ability to reveal vulnerabilities in the software.

Throughout the Software Testing Process, **peer** reviews are held to review and analyze the Test Cases and Test Procedures to determine that they are complete comply with the Software Verification Plan and cover the software high-level and low-level requirements.

Peer reviews are held to analyze the coverage achieved as a result of requirements-based testing. Where code structures are not covered, an analysis is performed to determine the cause. If the cause is determined to be untraceable code as a result of dead code, the code is removed. If the cause is inadequate requirements or test cases and procedures, the peer review results include action items to resolve this. If the result is unreachable "required" code that is traceable, an analysis of each line of uncovered code is documented in the Structural Coverage Analysis Results document.

#### 4.10.8.1 Software Verification Cases and Procedures Document Review

Review of the Software Verification Cases and Procedures occurs when the document is mature enough to be reviewed. Once prepared, the Software Verification Cases and Procedures is submitted to Software Configuration Management and entered into the document control system.

The Software Quality Assurance Engineer coordinates the document review process using the Document Review Management System. Each reviewer adds his or her comments in the Document Review Management System. A cycle of comment incorporation and re-review occurs through Configuration Management until all comments are closed. The Project Lead is responsible for closing all document comments prior to formal release.

Once all comments have been closed, the Software Verification Cases and Procedures is reviewed by the Software Quality Assurance Engineer against the Document Review Checklist and a cross references from each section of the Software Verification Cases and Procedures to the DO-178B Section 11 Objective to ensure that full compliance is achieved (See screenshot below). Once complete, the Software Quality Assurance Engineer signs and dates the checklists, which is maintained by Software Configuration Management as CC2 compliance evidence. The Software Verification Cases and Procedures is then signed and released.

#### 4.10.8.2 System Verification Review

The System Verification Review is conducted at the conclusion of the Software Testing Process. The System Verification Review Checklist will be used during the review.

The Project Engineer conducts the System Verification Review. When the System Verification Review is held, the Project Engineer records the minutes or assigns someone to do so. The minutes will include a discussion of the results, agreements and disagreements reached during the review, updates to the project schedule and resource estimates, and action item assignments with estimated completion dates.

Representatives from Quality Assurance, Test Engineering, Manufacturing Engineering, Mechanical Engineering, and Software Engineering are invited to the System Integration Review.

The review ensures the results of the integration process are complete and correct. If deficiencies are revealed during the review, corrective actions to resolve the deficiencies are fed back into the appropriate process.

#### 4.10.8.3 Reviews and Analysis of Test Cases, Test Procedures, and Results

Peer reviews are conducted to analyze Test Cases, Test Procedures, and Results. The following characteristics are evaluated and form the exit criteria for the peer review:

- Test cases: Independent verification of test cases is presented later in this document.
- Test procedures: The objective is to verify that the test cases were accurately developed into test procedures and expected results.
- Test results: The objective is to ensure that the test results are correct and that discrepancies between actual and expected results are explained.

During the verification process, the Independent Verification Engineer (along with select members of the development team) reviews the requirements-based test cases to assure that all requirements are adequately covered. If the requirements-based tests are not adequate to achieve test coverage, then additional requirements-based tests or analysis may be needed.

The following questions are considered when evaluating test cases and procedures:

- Do the test cases and procedures adhere to the relevant plans and standards?
- If plans or standards have not been followed, is there documented rationale for deviations from stated plans and standards?
- Is the rationale for each test case clearly explained?
- Are the test cases and procedures appropriately commented to allow future updates?
- Have the test cases and procedures been subjected to appropriate change and configuration control?
- Is the separation between test cases clear? For example, are test starts and stops identified?
- Do the test cases and procedures specify required input data and expected output data?
- Were the inputs for each test case derived from the requirements?
- Are the test cases and procedures sufficient to cover all the relevant requirements? That is, do the traceability matrices provide clear association between test cases and requirements?
- Are the test cases and procedures sufficient to achieve test coverage?
- Are sufficient tests identified to provide test coverage for each logic construct?
- Are there sufficient robustness test cases and procedures?
- Are test cases and procedures correct?

### 4.10.8.3.1 Review checklists for test cases, procedures, and results

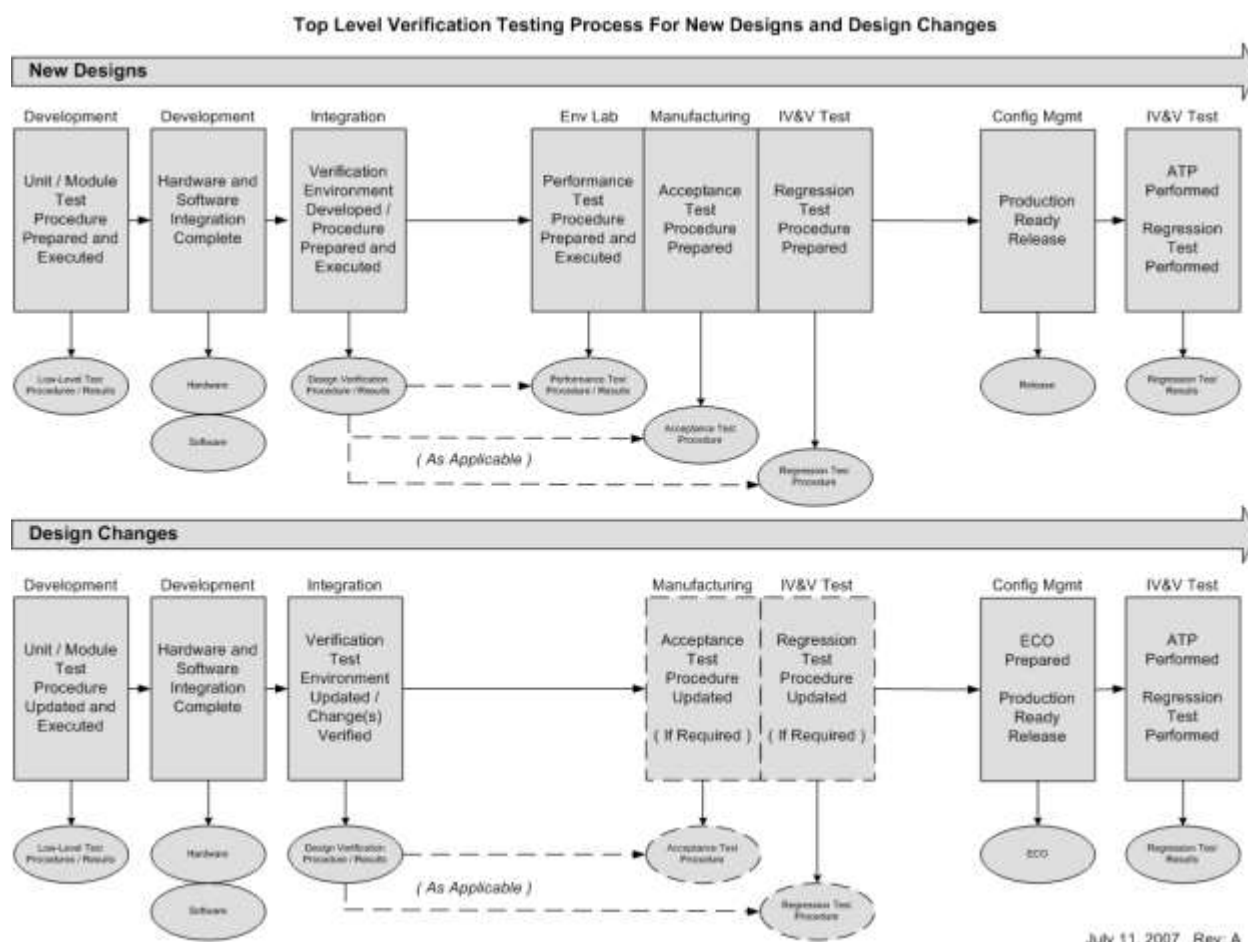
A checklist is used during review of test procedures and results (See Document Review Management System in the Software Quality Assurance Plan). During this review, the checklists themselves are assessed, considering the following questions for test coverage:

- Are the checklists sufficient to determine that the requirements-based test cases, procedures, and results meet the test coverage objective?
- Have the checklists been prepared and/or reviewed by quality?
- Do the checklists specify:
  - who performed the review?
  - what data was reviewed (with revision)?
  - when it was reviewed?
  - what was found?
  - what corrective actions were taken, if necessary?
- Do the checklists require evaluation of tolerances specified in the requirements?
- Do the checklists ensure that results of the test cases can be visually verified? (e.g., can the SQE, or other reviewer, visually determine when requirements-based tests have passed or failed?)
- Will the checklists reveal whether the results of the test cases that are counted for credit towards test coverage are observable?
- Will the checklists address limitations of the structural coverage analysis tool as documented in the tool qualification?
- Will the checklists reveal test cases that violate project standards?

## 4.10.9 Software Test Execution

Verification testing of software has two objectives. One objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors that could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed. The following three types of testing are used:

- Hardware Software Integration Testing: To verify correct operation of the software in the target computer environment.
- Software Integration Testing: To verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.
- Low-level Testing: To verify the implementation of software and low-level requirements.



**Peer** reviews are conducted to ensure that software testing objectives have been satisfied. The following characteristics are evaluated and form the exit criteria for the peer review:

- Test cases are developed based primarily on the software requirements.
- Test cases are developed to verify correct functionality and to establish conditions that reveal potential errors.
- Software requirements coverage and traceability analysis are used to determine what software requirements were not tested.
- Although not required, structural coverage analysis techniques may be used to determine what software structures were not exercised.

#### 4.10.9.1 Test Environment

More than one test environment may be needed to satisfy the objectives for software testing. The test environment includes the (independently built) software loaded into the target computer and tested in the target computer environment.

*Note: This section must be tailored to specify the actual test environments in use.*

#### 4.10.9.2 Requirements-Based Test Cases

Requirements-based testing is emphasized because this strategy has been found to be the most effective at revealing errors. Requirements-based test case selection includes the following:

- Implementation of both normal range and robustness (abnormal range) test cases. The specific test cases should be developed from the software requirements and the error sources inherent in the software development process.

#### 4.10.9.3 Normal Range Test Cases

Normal Range test cases are developed to demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases include:

- Real and integer input variables are exercised using valid equivalence classes and boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code are performed to check the characteristics of the function in context.
- For state transitions, test cases are developed to exercise the transitions possible during normal operation.
- For software requirements expressed by logical equations, the normal range test cases verify the variable usage and the Boolean operators.

#### 4.10.9.4 Robustness Test Cases

Robustness test cases are developed to demonstrate the ability of the software to respond to abnormal inputs and conditions. Robustness test cases include:

- Real and integer variables are exercised using equivalence class selection of invalid values.
- System initialization is exercised during abnormal conditions.
- The possible failure modes of the incoming data are determined, especially complex, digital data strings from an external system.
- For loops where the loop count is a computed value, test cases may be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
- For time-related functions, such as filters, integrators, and delays, test cases may be developed for arithmetic overflow protection mechanisms.
- For state transitions, test cases may be developed to provoke transitions that are not allowed by the software requirements.

##### 4.10.9.4.1 Robustness Test Case Selection Strategy

The following criteria will be used, at a minimum, to select robustness test cases:

9. Starting with the Functional Hazard Assessment (FHA), produce each hazard conditions and verify expected result.
10. Test multiple combinations of hazard conditions. Combine hazards.
11. Identify all range / boundary requirements. Test outside the boundaries of each range requirement.
12. Identify all conditions where a fault is asserted. Test all of those conditions.
13. Combine fault conditions (without reset) and verify expected results.
14. Perform negative testing. If A AND B THEN X, test If NOT A and B THEN NOT X.
15. If not required by DO-160 Testing, test critical functionality and other key functionality over temperature.
16. Perform testing over non-standard electrical conditions (i.e., power glitching, power up, power down, brown out)

#### 4.10.9.5 Requirements-Based System Verification Testing Methods

Requirements-based system verification testing methods concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The objective of requirements-based testing is to ensure that the software in the target computer will satisfy the high-level requirements.

Typical errors revealed by this level of testing include:

- Incorrect input handling.
- Failure to satisfy execution requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.
- Errors in system interfaces.
- Incorrect behavior of feedback loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.

Peer reviews are conducted to ensure that common software errors were not introduced into the design. The review includes the focus on the following most common error conditions:

➤ Implementation Error Source (Data Bugs)

18) Logic bugs → ( $x = 0$ ;  $x \leq 10$ ;  $x++$ ) → Expect a result of 10

19) Parameter Passing → Incorrect arguments passed

20) Return Codes → Unexpected return codes passed

21) Math Overflow / Underflow → Exceeding integer value

22) Logic Processing Error → Too many nested conditions or calculations

23) Reentrance Problem → If a section of code can be interrupted before it completes its execution, and can be called again before the first execution has completed, the code must be designed to be reentrant. This typically requires that all variables referenced by the reentrant routine exist on the stack and not in static memory.

24) Incorrect Control Flow → The intended sequence of operations can be corrupted by incorrectly designed conditional loops. This may cause problems such as missing execution paths, unreachable code, and incorrect control logic.

25) Pointer Errors → Pointing to a NULL pointer in a linked list, improperly incrementing pointer used to step through look-up tables or lists, bad function pointers.

26) Indexing Problems → Improper use of Index Registers in assembly language have similar problems to those identified with pointers. Provides the same type of indirection useful for table look-up, walking through lists, trees, and other data structures.

27) Variable Scope Errors → Using the same name and applying it to different data items that exist in different scopes.

28) Improper Data Usage → Using an uninitialized variable or using the same variable for more than one purpose.

29) Incorrect Flag Usage → Flags are usually global in scope and are almost always static (stored in a fixed memory location). Flag may inadvertently be used for more than one purpose or used to indicate more than one condition. Every flag should be SET, CLEARED and tested at some point in the program.

30) Incorrect Address → Usually the result of an incorrect pointer. It's possible to code a bad address into the code. This generally happens when the memory subsystem changes (i.e., Reduce memory size).

31) Data / Range Overflow / Underflow → May result in passing a parameter that is out of bounds or storing a data type not large enough to hold the data.

32) Signed / Unsigned data errors → Mixed sign arithmetic can easily lead to calculations that overflow the data types. Assembly languages have different branch instructions used after comparing signed and unsigned data. Using the wrong branch instruction may cause a critical error.

33) Incorrect Conversion / Type-Casting / Scaling → Converting a data value from one representation to another is common and may cause bugs. Conversion from signed to unsigned or string to numeric type is common. Typecasts are useful to get data into whatever representation is needed, but circumvent compiler type-checking, increasing the risk of making a mistake.

34) Data Synchronization Errors → Embedded systems share data among separate threads of execution. An operation that uses a number of different data inputs must be synchronized in order to perform its processing. If the data values are updated asynchronously, the processing may be using some "new" data items with some "old" data items, and compute the wrong result.

➤ Implementation Error Source (Real-Time Bugs)

3) Interrupt Handling → It is critical to handle all interrupts that the system will ever receive. Receiving an unexpected interrupt without being able to handle it will likely cause failures.

4) Task Synchronization → Tasks must be synchronized correctly. One task may acquire raw data; another may process this data as a set; still another may make control decisions on the processed data values. Proper synchronization usually is implemented by relying on flags or semaphores to control task regular intervals.

➤ Implementation Error Source (System Bugs)

5) Stack Overflow / Underflow → Pushing more data into the stack than it can hold is referred to as a stack overflow. Pulling more data from the stack than was put on the stack is referred to as a stack underflow. Both result in using bad data and can cause an unintended jump to an arbitrary address, resulting in a failure.

6) Race Conditions → A race condition occurs when two or more independent threads each access the same resource at the same time. The effects of a race condition vary widely; they're dependent on the specifics of the situation.

7) Deadlock → When race conditions are avoided by "locking" a resource, preventing any other thread from accessing it, the design must be evaluated to ensure that deadlock will never occur. Testing for deadlock is generally ineffective, since only a particular order of resource locking may produce it, and that ordering may not result from the most common tests.

Deadlock is only a problem in multi-threading environments that lock resources. The following four conditions must be present in order for a deadlock to occur. Breaking any one of these conditions eliminates deadlock:

- a. Mutual exclusion—only one thread can use a locked resource at a time
- b. Nonpreemption—threads cannot force another thread to release a resource
- c. Hold-and-wait—threads hold resources that they have locked while waiting for any additional needed resources
- d. Circular wait—a circular chain of threads exist, such that each thread holds a resource needed by the next thread in the chain

8) Resource Sharing Problems → In the case where a peripheral such as an analog multiplexer may be used to direct one of a number of different inputs to a single A/D converter; If one task alters the mux setting to measure a given signal and another preempts it and sets the mux to pass a different signal, when control returns to the first task, it will be measuring the wrong signal, likely causing a failure condition.

- Implementation Error Source (Other Bugs)
- 6) Syntax / Typing → Compilers do a good job of syntax checking; however, special attention needs to be placed on coding standards.
- 7) Interface → Complex interfaces are a common source of failures. Interface problem may include incorrect EEPROM erase / write sequence, improper use of LCD controller chip commands, wrong sequence in reading / writing serial communication interface registers, etc.
- 8) Memory Allocation / Deallocation → Using memory management routines can greatly simplify the efficient use of available memory. It can also be an added source of errors. For example, not checking for successful allocation before using the memory, not freeing memory when it is no longer needed (memory leak).
- 9) Peripheral Register Initialization → Peripherals typically have different modes of operation, increasing the number of applications for which they're useful. This can complicate the initialization and use of these devices producing another source for errors.
- 10) Watchdog Servicing → Watchdog timers help ensure that if something in the system goes exceptionally wrong, it will fail in a safe, or at least a predictable, manner. Servicing the watchdog timer must be done properly and at the right time. The watchdog must be enabled, and set to timeout at the correct interval.

#### 4.10.9.5.1 Requirements-Based Software Verification Testing

This testing method is used and concentrates on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. The objective of the requirements-based Software Verification Testing is to ensure that the software components interact correctly with each other and satisfy the software requirements through successive integration of code components with a corresponding expansion of the scope of the test cases.

Typical errors revealed by this testing method include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

#### 4.10.9.5.2 Requirements-Based Low-Level Testing

This testing method is used and concentrates on demonstrating that each software component complies with its low-level requirements. The objective of requirements-based low-level testing is to ensure that the software components satisfy their low-level requirements:

Typical errors revealed by this testing method include:

- Failure of an algorithm to satisfy a software requirement.
- Incorrect loop operations.
- Incorrect logic decisions.
- Failure to process correctly legitimate combinations of input conditions.
- Incorrect responses to missing or corrupted input data.
- Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
- Incorrect computation sequence.
- Inadequate algorithm precision, accuracy, or performance.

#### 4.10.10 Effectiveness of Test Program

The following tasks are performed to determine the effectiveness of the test program.

##### 4.10.10.1 Assess results of requirements-based tests

The first step after test execution is to determine whether all requirements-based tests pass. In addition to checking the final pass/fail results, the test cases and results for some randomly selected requirements should be examined to ensure that the results reflect the given inputs for those cases. Test results are also checked carefully with respect to any specified tolerances.

The following questions are considered to assess the requirements-based test results:

- Are the test result files clearly linked to the test procedures and codes?
- Are failed test cases obvious from the test results?
- Do the test results indicate whether each procedure passed or failed and the final pass/fail results?
- Do the test results adhere to the relevant plans, standards, and procedures?
- Have the test results been subjected to appropriate configuration control?

#### 4.10.10.2 Assess failure explanations and rework

Each failed test case is documented with an explanation for why it failed, including references to applicable Action Request. In some cases, rework of some life cycle data will be required; in other cases, only an explanation for the failed test cases is needed. If rework is required, the impact of changes should be carefully evaluated and the changed items should be subjected to the appropriate change and configuration control.

Once all rework is complete, test cases should be rerun in compliance with plans for regression testing. Note: there may be cases where failed requirements-based tests are acceptable; however, it is typical for them to be fixed and rerun.

The following questions are considered to assess failures and rework:

- Is there an acceptable rationale for deviations from expected results, standards, or plans?
- Are explanations for the failed test cases technically sound and accurate?
- Do explanations for failed test cases contain accurate references to relevant problem reports?
- Are explanations for code or test rework suitable to address the failure?
- Have test cases been re-executed in compliance with plans for regression testing?
- Have the test results from regression testing been documented appropriately?

#### 4.10.10.3 Assess coverage achievement

The Verification Engineer produces test cases that are expected to achieve 100% test coverage (i.e., the purpose of test documentation is to show compliance with all of the requirements). If all the requirements have been covered by tests without achieving full test coverage, dead code, unintended functionality, or incorrectly documented de-activated code may be indicated. It is the policy to remove all dead code.

The following questions are considered when assessing coverage achievement:

- Has the test coverage criteria been correctly applied?
- Is 100% structural coverage achieved through requirements-based testing?
- If 100% structural coverage is not achieved through requirements-based testing, is there an explanation detailing which parts of the code were not executed, and why? Have additional test cases been added?
- Are explanations for drops in coverage sufficiently detailed and acceptable?
- Are there problem reports associated with dead code?
- Has dead code been analyzed and/or removed?

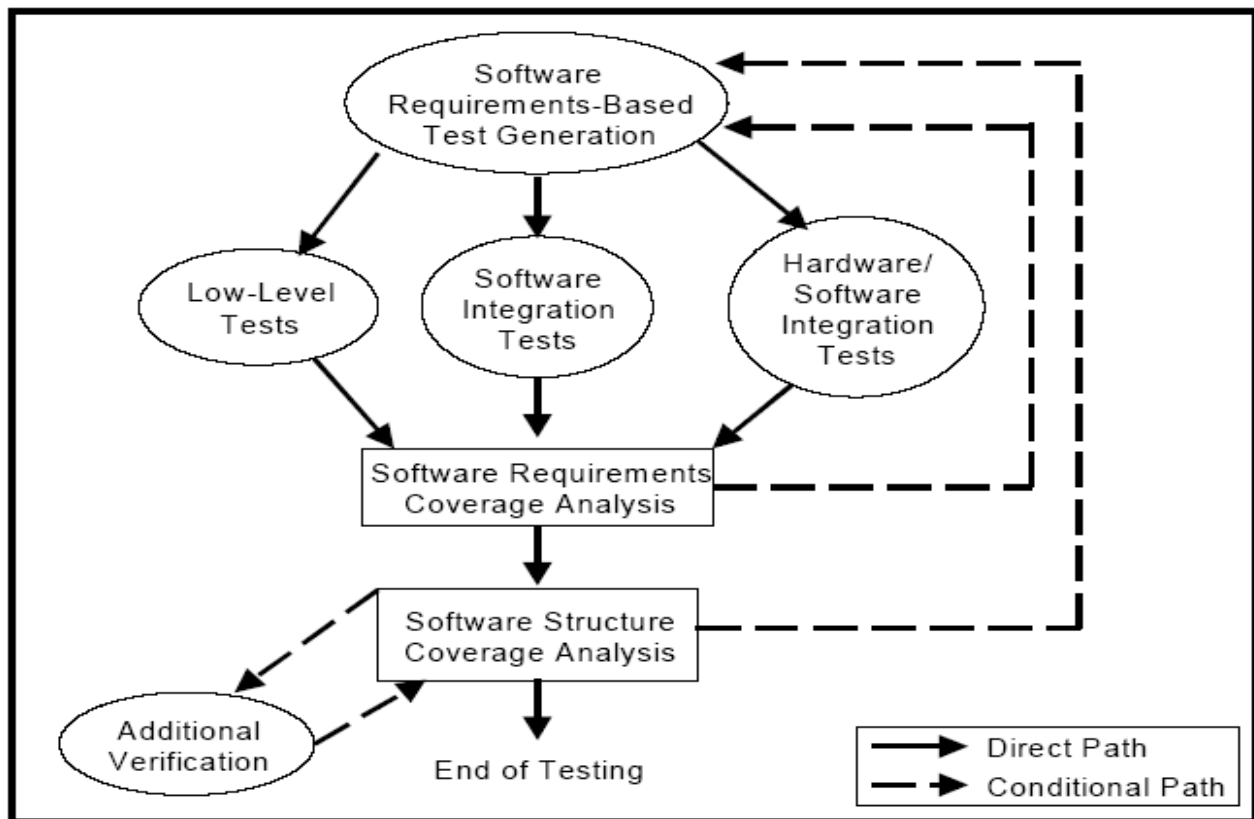
#### 4.11 Coverage Analysis Methods

The subsequent paragraphs detail the methods that will be used for coverage analysis as part of the software verification process.

Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage analysis measures will be applied to both requirements definitions and testing activities. Appropriate coverage measures will be used by SQA to audit verification activities. This will aid in determining the adequacy of the verification accomplished.

Coverage is viewed as a measure, not a method or a test. As such, results will be expressed as the percentage of an activity that is accomplished. Two specific measures of test coverage are identified in the following figure: requirements coverage and software structure coverage.

Requirements coverage analysis will be used to determine how well the requirements-based testing verifies the implementation of the software requirements and establishes traceability between the software requirements and the test cases. Structural coverage analysis will be used to determine how much of the code structure will be executed by the requirements-based tests and establishes traceability between the code structure and the test cases.



#### 4.11.1 Requirements Coverage Analysis

Each software requirement contains a finite list of behaviors and features, and that each requirement is written to be verifiable. Testing based on requirements will be performed from the perspective of the user (providing a demonstration of intended function), and will provide a means for the development of test cases concurrently with development of the requirements.

Peer reviews will go beyond requirements coverage in evaluating the project. Reasons include:

- The software requirements and the design description (used as the basis for the test set) may not contain a complete and accurate specification of all the behavior represented in the executable code.
- The software requirements may not be written with sufficient granularity to assure that all the functional behaviors implemented in the source code are tested.
- Requirements-based testing *alone* cannot confirm that the code does not include unintended functionality.

In addition, software structure may be created that cannot be determined from top-level software specifications. Derived requirements, as described in DO-178B, will be used for this reason. Derived requirements will be tested as part of requirements-based testing.

#### 4.11.2 Structural Coverage Analysis

The purpose of structural coverage analysis with the associated structural coverage analysis resolution is to complement requirements-based testing as follows:

- Provide evidence that the code structure was verified to the degree required for the applicable software level.
- Provide a means to support demonstration of absence of unintended functions.
- Establish the thoroughness of requirements-based testing.

With respect to intended function, evidence that testing was rigorous and completed is provided by the combination of requirements-based testing (both normal range testing and robustness testing) and requirements-based test coverage analysis.

Requirements-based testing cannot completely provide this kind of evidence with respect to unintended functions. Code that is implemented without being linked to requirements may not be exercised by requirements-based tests. Such code could result in unintended functions. In this case, it will be designated this "Dead Code" or require that a requirement be written for the code. Should a new requirement be added, the applicable lifecycle artifacts (i.e., the Software Requirements Document) will be updated and the required processes will be repeated.

If requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a greater level of confidence that there are no unintended functions. Structural coverage analysis will:

- Indicate to what extent the requirements-based test procedures exercise the code structure.
- Reveal code structure that was not exercised during testing.

Run-time libraries are subject to the same coverage requirements as the rest of the application code.

It should be noted that the structural coverage tools employed on the project must support resolution of overloaded operators and/or functions to the extent overloading is used on the project.

#### 4.11.2.1 Achieving Coverage

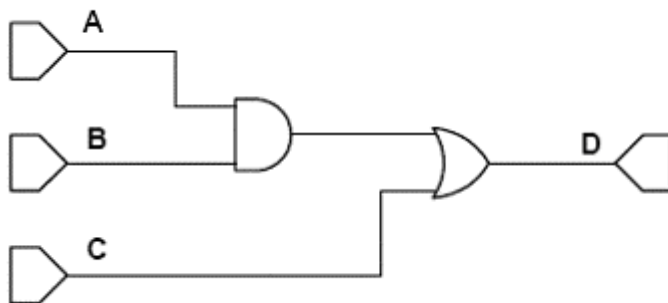
To achieve test coverage, a structural coverage analysis tool or a code instrumentation method will be used to monitor statements, entry and exit points, decision and branching statements, and Boolean conditions. Some tools do not support all of the coverage points required for test coverage. For example, not all structural coverage tools support coverage of entry and exit points. Such a tool can support part of the structural coverage analysis if other means are used to cover entry and exit points.

The structural coverage analysis tool will monitor a statement for multiple coverage points, as illustrated below:

Return (A and B) or C;

This statement will be monitored for the following coverage points:

- Statement—must be invoked at least once
- Exit Point—must be invoked at least once
- Decision—must take all possible outcomes (*false*, *true*) at least once



Test Case Number	1	2	3	4	5
Input A	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input B	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Input C	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output D	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

---

#### 4.11.2.2 Statement Coverage

To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. Achieving statement coverage shows that all code statements are reachable (in the context of DO-178B, reachable based on test cases developed from the requirements). Note that statement coverage is considered a weak criterion because it is insensitive to some control structures. Consider the following code segment:

```
If ( x > 1 ) and ( y = 0 ) then z := z / x; end if;
```

By choosing  $x = 2$ ,  $y = 0$ , and  $z = 4$  as input to this code segment, every statement is executed at least once. However, if an “**or**” is coded by mistake (see code segment below) in the first statement instead of an “**and**”, the test case will not detect a problem. This makes sense because analysis of logic expressions is not part of the statement coverage criterion.

```
If ( z = 2 ) or ( y > 1 ) then z := z + 1; end if;
```

#### 4.11.2.3 Decision Coverage

Decision coverage requires two test cases: one for a *true* outcome and another for a *false* outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. But, not all decisions are simple. For the decision (**A or B**), test cases (*TF*) and (*FF*) will toggle the decision outcome between *true* and *false*. However, the effect of **B** is not tested; that is, those test cases cannot distinguish between the decision (**A or B**) and the decision **A**.

#### 4.11.2.4 Modified Condition Decision Coverage

Condition coverage requires that each condition in a decision take on all possible outcomes at least once (to overcome the problem in the previous example), but does not require that the decision take on all possible outcomes at least once. In this case, for the decision (**A or B**) test cases (*TF*) and (*FT*) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two test cases is required for each decision.



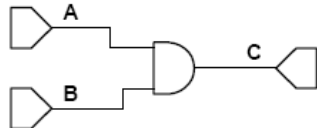
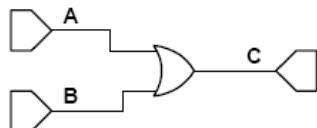
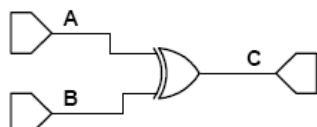

Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between *true* and *false* and to toggle each condition value between *true* and *false*. Hence, a minimum of two test cases are necessary for each decision. Using the example (**A or B**), test cases (*TT*) and (*FF*) would meet the coverage requirement. However, these two tests do not distinguish the correct expression (**A or B**) from the expression **A** or from the expression **B** or from the expression (**A and B**).

MC/DC enhances the condition/decision coverage criterion by requiring that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, as will be discussed further in chapter 3, and, in general, a minimum of  $n+1$  test cases for a decision with  $n$  inputs. For the example (**A or B**), test cases (*TF*), (*FT*), and (*FF*) provide MC/DC. For decisions with a large number of inputs, MC/DC requires considerably more test

cases than any of the coverage measures discussed above.

Multiple Condition Coverage requires test cases that ensure each possible combination of inputs to a decision is executed at least once. Thus, multiple condition coverage requires exhaustive testing of the input combinations to a decision. In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with  $n$  inputs, multiple condition coverage requires 2 to the  $n^{th}$  tests.

### Representations for Elementary Logical Expressions

Name	Schematic Representation	Code example	Truth Table															
Input																		
Output																		
and Gate		<b>C := A and B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>F</td></tr><tr><td>F</td><td>T</td><td>F</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	T	T	F	F	F	T	F	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	T																
T	F	F																
F	T	F																
F	F	F																
or Gate		<b>C := A or B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>T</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	T	T	F	T	F	T	T	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	T																
T	F	T																
F	T	T																
F	F	F																
xor Gate		<b>C := A xor B;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th><th><u>C</u></th></tr><tr><td>T</td><td>T</td><td>F</td></tr><tr><td>T</td><td>F</td><td>T</td></tr><tr><td>F</td><td>T</td><td>T</td></tr><tr><td>F</td><td>F</td><td>F</td></tr></table>	<u>A</u>	<u>B</u>	<u>C</u>	T	T	F	T	F	T	F	T	T	F	F	F
<u>A</u>	<u>B</u>	<u>C</u>																
T	T	F																
T	F	T																
F	T	T																
F	F	F																
not Gate		<b>B := not A;</b>	<table><tr><th><u>A</u></th><th><u>B</u></th></tr><tr><td>T</td><td>F</td></tr><tr><td>F</td><td>T</td></tr></table>	<u>A</u>	<u>B</u>	T	F	F	T									
<u>A</u>	<u>B</u>																	
T	F																	
F	T																	

### AND Gate

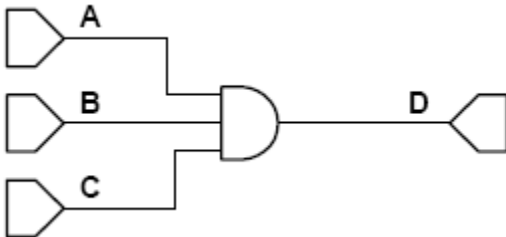
The following tests will be performed to achieve test coverage for an **"and"** gate:

- All inputs are set *true* with the output observed to be *true*. This requires one test case for each  $n$ -input **"and"** gate.
- Each and every input is set exclusively *false* with the output observed to be *false*. This requires  $n$  test cases for each  $n$ -input **"and"** gate.

Changing a single condition starting from a state where all inputs are *true* will change the outcome; that is, an **"and"** gate is sensitive to any *false* input. Hence, a specific set of  $n+1$  test cases is needed for an  $n$ -input **"and"** gate. These specific  $n+1$  test cases meet the intent of test coverage by demonstrating that the **"and"** gate is correctly implemented.

The following is an example of the minimum testing required for a three-input **"and"** gate. In this case, it takes four test cases to show that each input "independently" affects the output.

If ( A = 1 ) and ( B = 1 ) and ( C = 1 ) then D := 1; end if;



Test Case Number	1	2	3	4
Input A	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input B	<i>T</i>	<i>T</i>	<i>F</i>	<i>T</i>
Input C	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output D	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>

### OR Gate

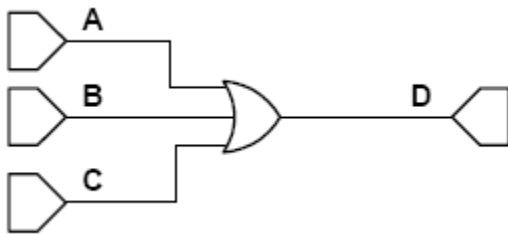
The following tests will be performed to achieve test coverage for an "or" gate:

- All inputs are set *false* with the output observed to be *false*. This requires one test case for each  $n$ -input "or" gate.
- Each and every input is set exclusively *true* with the output observed to be *true*. This requires  $n$  test cases for each  $n$ -input "or" gate.

These requirements are based on an "or" gate's sensitivity to a *true* input. Here again,  $n+1$  specific test cases are needed to test an  $n$ -input "or" gate. These specific  $n+1$  test cases meet the intent of test coverage by demonstrating that the "or" gate is correctly implemented.

The following is an example of the minimum testing required for a three-input "or" gate. In this case, it takes four test cases to show that each input "independently" affects the output.

If (  $A = 1$  ) or (  $B = 1$  ) or (  $C = 1$  ) then  $D := 1$ ; end if;



Test Case Number	1	2	3	4
Input A	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>
Input B	<i>F</i>	<i>F</i>	<i>T</i>	<i>F</i>
Input C	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>
Output D	<i>F</i>	<i>T</i>	<i>T</i>	<i>T</i>

### XOR Gate

The "**xor**" gate differs from both the "**and**" and the "**or**" gates with respect to test coverage in that there are multiple minimum test sets for an "**xor**". Consider the two-input "**xor**" gate. All of the possible test cases for this "**xor**" gate are shown below. For a two-input "**xor**" gate, any combination of three test cases will provide test coverage.

The following is an example of the minimum testing required for a two-input "**xor**" gate. Minimum testing to meet test coverage requires one of the following sets of test cases:

- test cases 1, 2, and 3
- test cases 1, 2, and 4
- test cases 1, 3, and 4
- test cases 2, 3, and 4

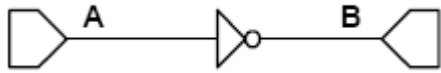
If ( A = 1 ) xor ( B = 1 ) then C := 1; end if;

Test Case Number	1	2	3	4
Input A	<i>T</i>	<i>T</i>	<i>F</i>	<i>F</i>
Input B	<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i>
Output C	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

Note that for a test set to distinguish between an "**or**" and an "**xor**" gate it must contain test case 1. Test sets 1, 2, and 3 above can detect when an "**or**" is coded incorrectly for an "**xor**", and vice versa. While not explicitly required by test coverage, elimination of test set 4 as a valid test set is worth considering. Note also that minimum tests to achieve test coverage for an "**xor**" gate with more than two inputs are implementation dependent. Hence, no single set of rules applies universally to an "**xor**" gate with more than two inputs.

### Not Gate

The logical “**not**” works differently from the previous gates: the “**not**” works only on a single operand. That operand may be a single condition or a logical expression. But, with respect to a gate level representation, there is a single input to the “**not**” gate as shown below.



Minimum testing to achieve test coverage for a logical “**not**” requires the following:

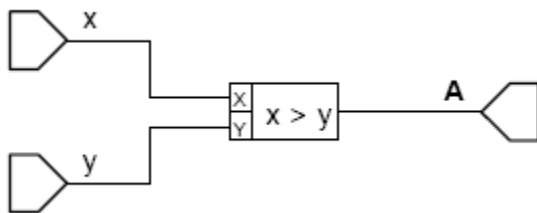
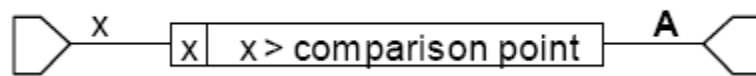
- The input is set *false* with the output observed to be *true*.
- The input is set *true* with the output observed to be *false*.

## Comparator

A comparator evaluates two numerical inputs and returns a Boolean based on the comparison criteria. Within the context of DO-178B, a comparator is a condition and also a simple decision. The following comparison criteria are considered in this tutorial:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equal to
- != not equal to

In general, the comparison point can be a constant or another variable.



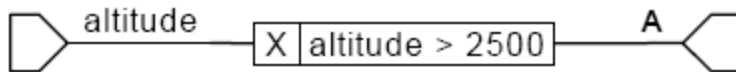
In either case, two test cases will be used to confirm test coverage for a comparator—one test case with a *true* outcome, and one test case with a *false* outcome. Minimum testing for a comparator requires the following:

- Input x set at a value above the comparison point (or y)
- Input x set at a value below the comparison point (or y)

Typically, three test cases will be used to assure that simple coding errors have not been made; that is, that the correct relational operator and comparison point are used in the code. So, while test coverage only requires two tests, minimum good requirements-based testing for a comparator requires:

- Input x set at a value slightly above the comparison point
- Input x set at a value slightly below the comparison point
- Input x set at a value equal to the comparison point

The definition of “slightly” is determined by engineering judgment based on the numerical resolution of the data type and/or target computer, the test equipment driving the inputs, and the resolution of the output device. Consider for example, the following set of test cases for a design that sets the output **A** *true* when altitude is greater than 2500.



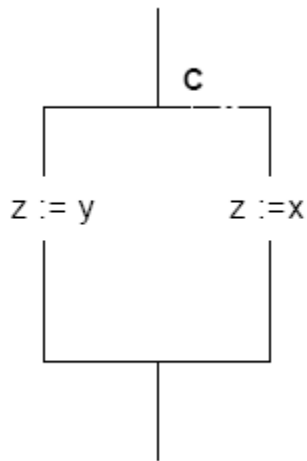
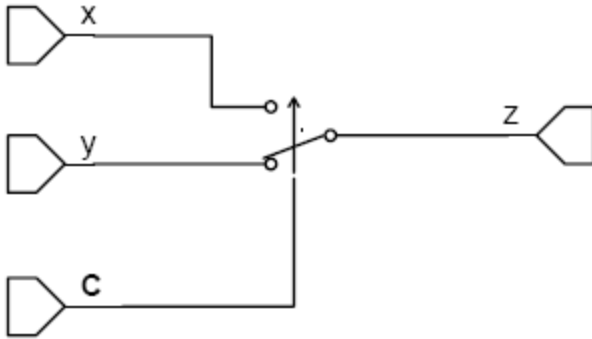
Test Case Number	1	2	3	4	5
Input altitude	25	32000	2500	2499	2501
Output A	<i>F</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>

Test cases 1 and 2 give the desired test coverage output. However, those test cases do not confirm that the toggle occurred at 2500, and not elsewhere. Even adding test case 3 does not improve the test suite much. The design could have been implemented with a comparison point anywhere between 2501 and 32000, and give the same result for test cases 1, 2, and 3. Test cases 3, 4, and 5 are a better set, because this set confirms that the transition occurs at 2500.

**If Then Else:**

The *if-then-else* statement is a switch that controls the execution of the software. Consider the following example where  $x$ ,  $y$ , and  $z$  are integers and  $C$  is a Boolean:

If  $C$  then  $z := x$  else  $z := y$ ;



The following tests will be performed for the *if-then-else* statement:

- Inputs that force the execution of the *then* path (that is, the decision evaluates to *true*)
- Inputs that force the execution of the *else* path (that is, the decision evaluates to *false*)
- Inputs to exercise any logical gates in the decision

Note that the decision must evaluate to *false* with confirmation that the *then* path did not execute, even if there is no *else* path.

For example, for a single condition **Z**, the statement *if Z then...else...* requires only two test cases to achieve test coverage. The decision in *if X or Y or Z then... else...* requires four test cases to achieve test coverage.

A minimal test set for the statement *if Z then a := x else a := y* is shown in Table 9. Note that a *case* statement may be handled similarly to the *if-then-else* statement.

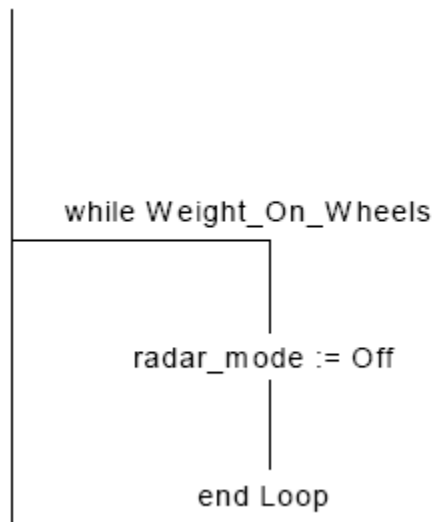
Test Case Number	1 Traverse the <i>then</i> path	2 Traverse the <i>else</i> path
Input x	12	18
Input y	50	34
Input <b>Z</b>	<i>T</i>	<i>F</i>
Output a	12	34

**While Loop:**

Consider the following example where **Weight\_On\_Wheels** is a Boolean:

While **Weight\_On\_Wheels** loop radar\_mode := Off; end loop;

A schematic representation of this code is shown in Figure 10. In this case, **Weight\_On\_Wheels** is the decision for the *while loop* construct.



The following tests will be performed for the *while loop*:

- Inputs to force the execution of the statements in the loop (that is, the decision evaluates to *true*)
- Inputs to force the exit of the loop (that is, the decision evaluates to *false*)
- Inputs to exercise any logical gates in the decision

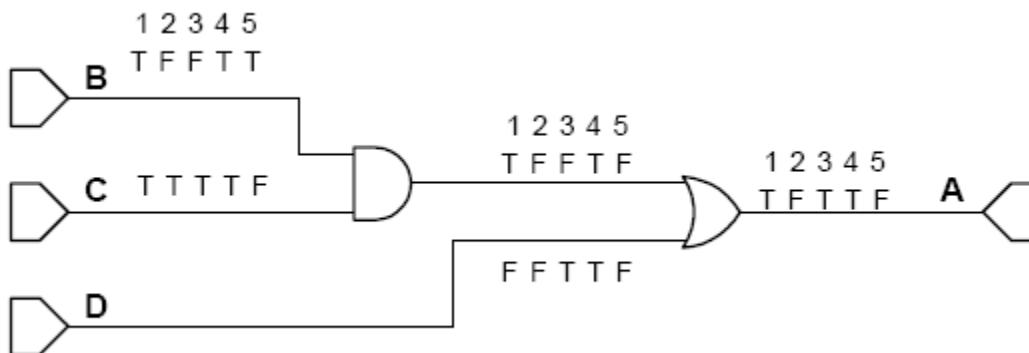
Two test cases may be used to achieve test coverage. One test case confirms that radar\_mode remains off as long as Weight\_On\_Wheels is true. The second test case confirms that radar\_mode could be set to something other than off when Weight\_On\_Wheels is false. In the case where Weight\_On\_Wheels is replaced by a Boolean expression, the Boolean expression would also need to be evaluated, and the setting of radar\_mode to off confirmed.

Applying Boolean Logic to Requirements-Based Testing

This process takes the inputs from the requirements-based test cases and maps them to the schematic representation. This provides a view of the test cases and the source code in a convenient format. Inputs and expected observable outputs for the requirements-based test cases for example 1 are given.

Test Case Number	1	2	3	4	5
Input <b>B</b>	<i>T</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>
Input <b>C</b>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>F</i>
Input <b>D</b>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>
Output <b>A</b>	<i>T</i>	<i>F</i>	<i>T</i>	<i>T</i>	<i>F</i>

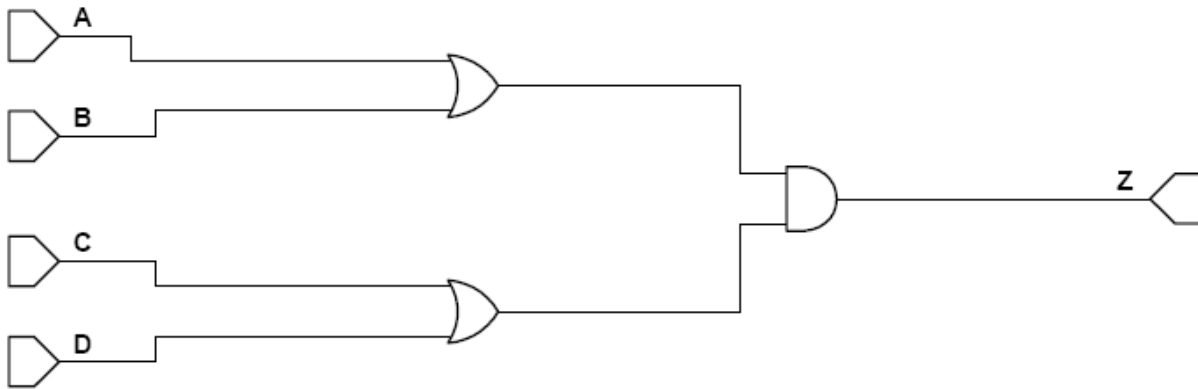
This example shows the test cases annotated on the schematic representation. Note that intermediate results are also determined from the test inputs and shown on the schematic representation.



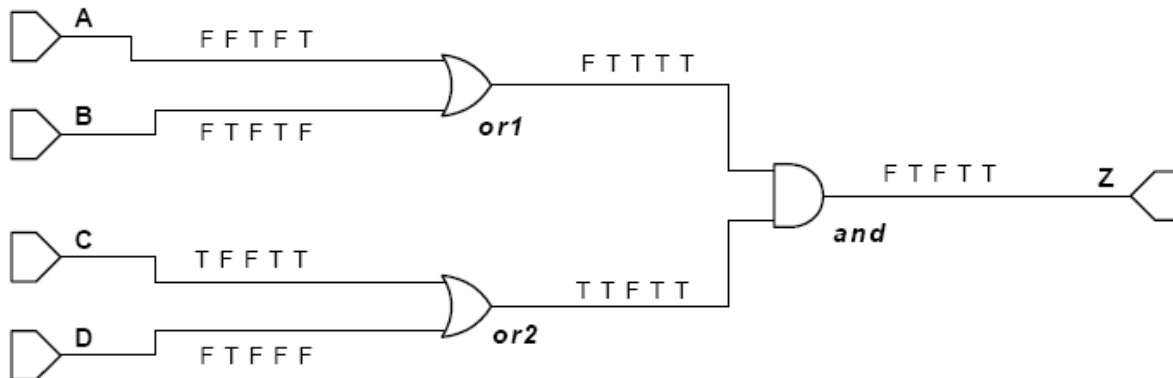
Knowing the intermediate results is important because some inputs may mask the effect of other inputs when two or more logic constructs are evaluated together. Test cases where the output is masked do not contribute to achieving test coverage. Using the annotated figure, the requirements-based tests cases that do not contribute (or count for credit) towards achieving test coverage can be identified. Once those test cases are eliminated from consideration, the remaining test cases can be compared to the building blocks to determine if they are sufficient to meet the test coverage criteria.

Expression:  $Z := (A \text{ or } B) \text{ and } (C \text{ or } D);$

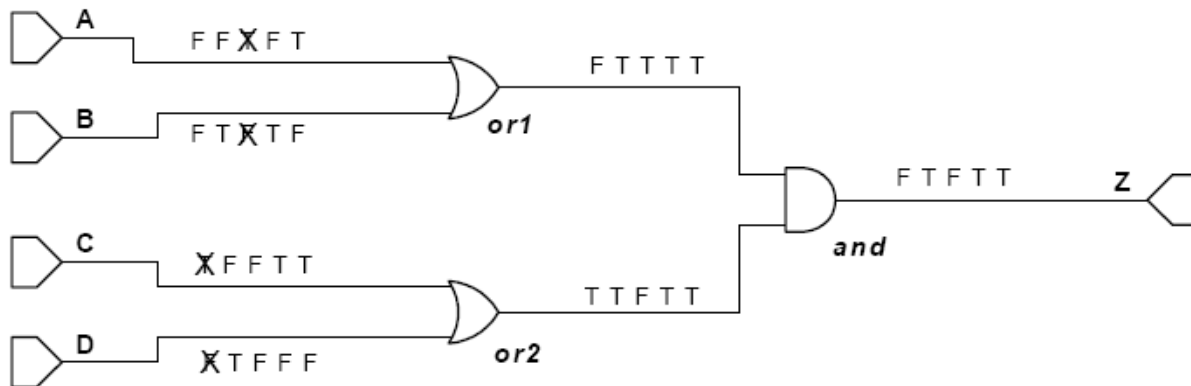
Step 1: Show the source code schematically.



Step 2: Map test cases to the source code picture.



Step 3: Eliminate masked tests. In this case, any *false* input to the "and" gate will mask the other input. In this case, the *false* outcome of "or1" will mask test case 1 for the "or2" gate. Similarly, the *false* outcome of "or2" will mask test case 3 for the "or1" gate.



Step 4: Determine test coverage.

Gate	Valid Test Inputs	Missing Test Cases
<b>or1</b>	FF Case 1 FT Case 2 or 4 TF Case 5	None
<b>or2</b>	FF Case 3 FT Case 2 TF Case 4 or 5	None
<b>and</b>	TT Case 2, 4, or 5 TF Case 3 FT Case 1	None

Step 5: Confirm output. The outputs computed match those provided.

### Symbols for Source Code Representation

Name	Schematic Representation	Code example
Comparator (x with constant)		<b>A</b> := x > constant;
Comparator (x with y)		<b>A</b> := x > y;
Summer (addition or subtraction may be shown)		<b>z</b> := x + y;
Multiplier		<b>z</b> := x * y;
Divider		<b>z</b> := x / y;
If-then-else		If <b>A</b> then <b>z</b> := x; Else <b>z</b> := y; End if;
If-then-else		If <b>A</b> then <b>z</b> := x; <b>w</b> := 3; Else <b>z</b> := y; <b>w</b> := 5; End if;
While Loop		While <b>A</b> Loop Read ( <b>A</b> ); End loop;

#### 4.11.2.5 Coverage Analysis Tools

A structural coverage analysis tool will be used to provide visibility into testing by either instrumenting code or providing other intervention techniques to gain visibility. The tool will be capable of instrumenting the code, provide flags, or other monitoring mechanisms to the original source code or object code. This enables the analysis tool to determine exactly what parts of the code are exercised. Once the code is instrumented, test cases are executed and the coverage analysis tool tracks which parts of the code are exercised by the test cases and, where complex analysis is required, how they are exercised. Pass/fail criteria for structural coverage are specified and tool analyzes the code against these criteria. If the pass/fail criteria are not specified, the tool will report the level of structural coverage the test cases achieve.

The Coverage Analysis Management System will be used to obtain both Statement and Decision Coverage.

#### Coverage Analysis Management System Screen Shot

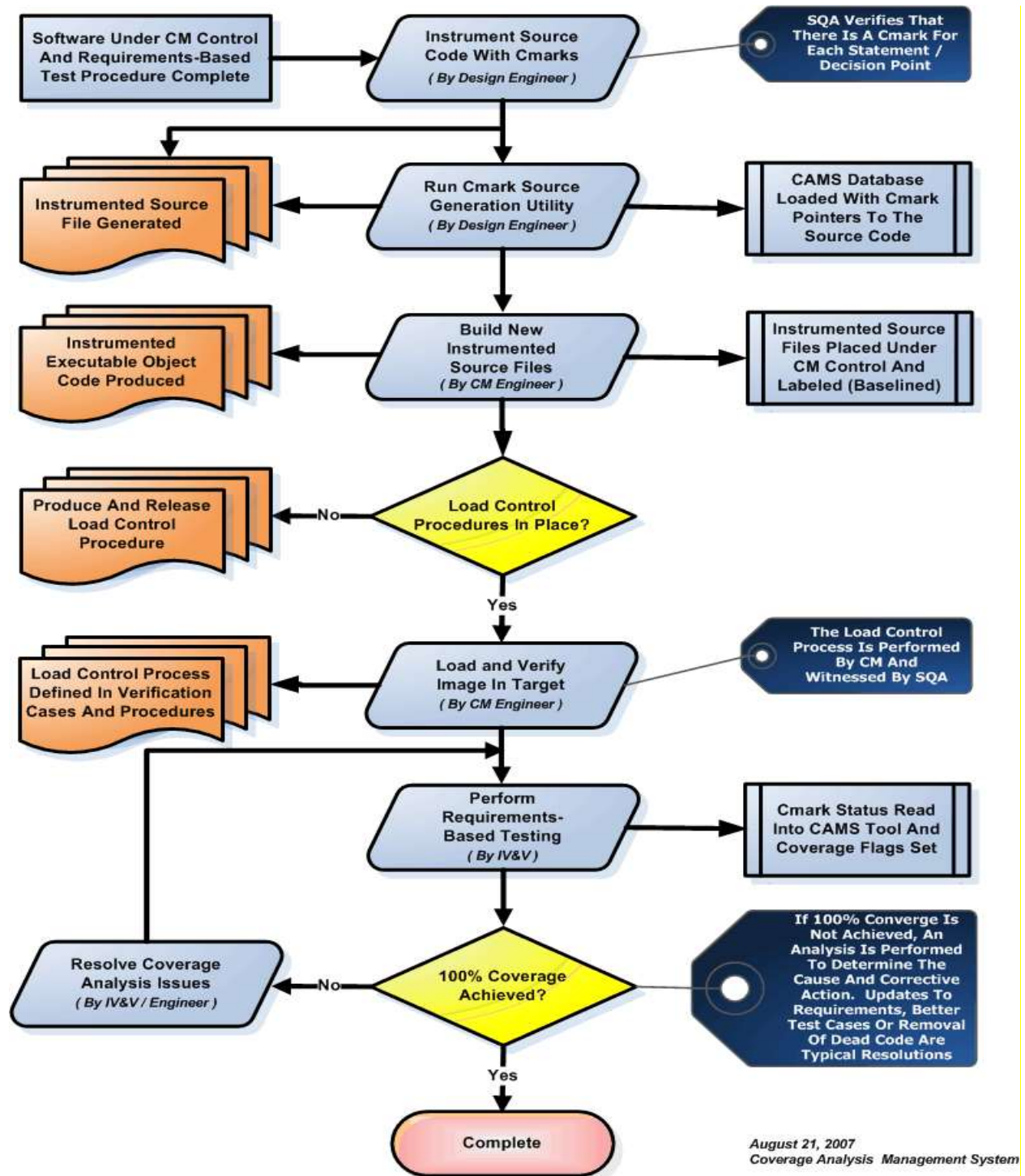
The screenshot displays the Coverage Analysis Management System web application. The browser window shows the URL: <http://www.faeconsultants.com/Projects/ExampleProject/000/CAMS/Just.asp?ProgramID=100&SessionType=New>. The application title is "Coverage Analysis Management System". Below the title, there is a search bar with the text "Select Software Release To Display:" and a dropdown menu labeled "Select A Release". A "Search" button is next to it. The current software release displayed is "Flight Management System".

Navigation links include: [Manage Source Files](#), [Instrument Source Code](#), [Perform Coverage Analysis](#), [Clear Database](#), and [Home Page](#).

The main section is titled "Coverage Analysis Detail" and contains a table with the following data:

Release	File	Version	Line No	Function Name	Code Element	Marker	Coverage
1.000	XYZ.c	4.2	038	void DisplayFunction (void)	Simple Statement	CMARK(0)	✓
1.000	XYZ.c	4.2	044	void TestFunction (void)	Simple Statement	CMARK(1)	✓
1.000	XYZ.c	4.2	116	word ADC_GetAvg (byte Chan)	If Statement (1)	CMARK(2)	✓
1.000	XYZ.c	4.2	120	word ADC_GetAvg (byte Chan)	Simple Statement	CMARK(3)	✓
1.000	XYZ.c	4.2	151	static void ADC_RunThresh (void)	Simple Statement	CMARK(4)	✓
1.000	XYZ.c	4.2	159	static void ADC_RunThresh (void)	If Statement (1)	CMARK(5)	✓
1.000	XYZ.c	4.2	175	static void ADC_RunThresh (void)	Case Statement (1)	CMARK(6)	✓
1.000	XYZ.c	4.2	179	static void ADC_RunThresh (void)	If Statement (2)	CMARK(7)	✓
1.000	XYZ.c	4.2	196	static void ADC_RunThresh (void)	Case Statement (2)	CMARK(8)	✓
1.000	XYZ.c	4.2	200	static void ADC_RunThresh (void)	If Statement (3)	CMARK(9)	✓
1.000	XYZ.c	4.2	209	static void ADC_RunThresh (void)	Else If Statement (1)	CMARK(10)	✓
1.000	XYZ.c	4.2	222	static void ADC_RunThresh (void)	Case Statement (3)	CMARK(11)	✓
1.000	XYZ.c	4.2	226	static void ADC_RunThresh (void)	If Statement (4)	CMARK(12)	✓
1.000	XYZ.c	4.2	243	static void ADC_RunThresh (void)	Case Statement (4)	CMARK(13)	✓
1.000	XYZ.c	4.2	247	static void ADC_RunThresh (void)	Complex Path	CNOTE(0)	Analysis

The Coverage Analysis Management System process is as follows. Specifics of this process and this tool are described in the CAMS Tool Qualification Accomplishment Summary.



#### 4.11.3 Source Code to Object Code Traceability

<Level A Only>

*To implement certain features, compilers for some languages may produce object code that is not directly traceable to the Source Code, for example, initialization, built-in error detection, or exception handling. That object code or executable object which cannot be traced to Source Code, must be verified. The software planning process should provide a means to detect this object code and to ensure verification coverage, and should define the means in the appropriate plan.*

##### Methods for Satisfying this Objective

1. *Perform Structural Coverage Analysis at Object Code or EOC Level*
2. *Compiler Qualification*

#### 4.11.4 Data Coupling and Control Coupling Analysis

Analysis of data coupling and control coupling is to ensure the adequacy of integration testing. It follows that this objective cannot be achieved without hardware/software integration testing or software integration testing. DO-178C/ED-12C will hopefully clarify that this is a structural coverage analysis which confirms that the requirements based testing has exercised the data and control coupling between code components. DO-178B/ED-12B also requires a separate review or analysis to verify that the source code matches the data flow and control flow defined in the software architecture. Such a review or analysis would satisfy the objective of DO-178B/ED-12B section 6.3.4b.

##### 4.11.4.1 Structural Coverage Analysis of Data and Control Coupling

The intent of the structural coverage analyses of data coupling and control coupling is to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies. That is, the intent is to show that the software modules/components affect one another in the ways in which the software designer intended and do not affect one another in ways in which they were not intended, thus resulting in unplanned, anomalous, or erroneous behavior. Typically, the measurements and assurance should be conducted on R-BT of the integrated components (that is, on the final software program build) in order to ensure that the interactions and dependencies are correct, the coverage is complete, and the objective is satisfied.

Satisfaction of this objective will be based on the detailed high and low level requirements of the modules/components' interfaces and the thorough requirements-based normal range and robustness tests of the software program. The interfaces and dependencies will be specified in the design requirements, and if those requirements are tested for both normal functioning and robustness. Satisfaction of the data and control coupling objective becomes a by-product of the design and verification processes.

The following sections identify the areas that are applicable and the means with which

verification will occur.

#### 4.11.4.2 Data Coupling Analysis

Data coupling manifests as:

(1) Parameters passed to a function.

In the case of parameters passed to the function (case 1); statement coverage is sufficient to determine whether all control paths through the function that might be influenced by the parameter set have been exercised.

(2) Global data set or used by the function whose value is determined at compile-time or as part of system configuration.

In the case of global configuration data (case 2); analysis should determine the equivalency classes of all potential configurations. Structural coverage analysis should be executed under all equivalency classes.

(3) Global data set or used by the function which represents the current state of execution of the system.

In the case of global state data (case 3); analysis should determine the potential states (or their equivalency classes). Structural coverage analysis through instrumentation should determine if all states have been entered and all legal transitions between states have been exercised.

Note 1: Sub-functions exist where a function parameter determines which of multiple independent execution paths is taken through a function. Usually the parameter is used to determine which case of a large switch statement is executed.

To satisfy the control coupling objective, use the structural coverage results to provide evidence that all functions were executed through high-level test cases. For functions that could not be exercised by high-level tests, develop additional functional analyses and add to the Software Verification and Procedures (SVCP). The intent is to provide confidence that the requirements-based testing has completely exercised the code structure.

To satisfy the data coupling objective, this analysis includes functional parameters, global variables, external data, stored data, and resource contention. Analyze the SVCP and associate test code to confirm the verification coverage of the data coupling in the code. As with the control coupling, structural coverage results can be used to provide evidence that the data coupling through parameters was covered.

CAST-19 Objective	Where and how the objective is met
Identify data dependencies.	This objective is met by defining the data items in the requirements and during the software requirements and code reviews ensuring proper

CAST-19 Objective	Where and how the objective is met
	<p>setting and using of the data.</p> <p>For global and static scoped objects, this objective should be understood to include the explicit verification of initialization dependencies. Because the precedence and declaration order of global and static scoped objects can cause failure to properly initialize, explicit test cases which verify the correct instantiation and initialization of objects in these scopes.</p>
Identify inappropriate data dependencies.	This objective is met by the performance of the software requirements and code reviews.
Define and evaluate the extent of interface depth	This objective is met by the simplicity and small size of the project and verified by the code review.
Determine and minimize coupling interdependencies.	This objective is met by the simplicity and small size of the project. There will be no specific review test or analysis to verify this objective.
Evaluate accurate use of global data	This objective is met by code review and requirements base testing. The requirements based tests will ensure the software performs as required. The combination of these verifications adequately verifies the use of global data.
Evaluate input/output data buffers	This objective will be met by the accumulation of all the requirements based tests being executed, with passed results. The Software Verification Review checklist addresses this objective.

#### 4.11.4.3Control Coupling Analysis

In the C language control coupling manifests in one of three ways:

(1) Static function calls.

In the case of static function calls (case 1); statement coverage is sufficient to determine if all possible calling points for a function have been executed by the test procedures.

(2) Sub-functions (See Note 1)

- In the case of sub-functions (case 2); analysis should reveal if the controlling parameter a constant determined at compile-time or whether the controlling parameter may be dynamically modified during execution.

- If the controlling parameter is a constant determined at compile-time, this case is equivalent to case 1.
- If the controlling parameter may be dynamically modified during execution, this case is equivalent to case 3.

(3) Dynamic function calls (i.e. function called through a pointer.)

- Points where a function is called through a pointer (case 3); it is necessary to determine whether (a) the function pointer has been initialized before use, (b) what the range of possible values for the function pointer is, and (c) that all possible values of the function pointer within that range have been executed.
- In the case of function pointers which belong to a jump table which is initialized at compile-time, this case is reduced to case 1.
- In the case of function pointers that are initialized at powerup, the calling point must be exercised in all potential configurations of the jump table. (Also see Data Coupling case 2.)

CAST-19 Objective	Where and how the objective is met
Identify control dependencies.	This objective is met by defining the data items in the requirements and during the software requirements and code reviews ensuring proper setting and using of the data
Identify inappropriate control dependencies.	Inappropriate control dependencies will be removed. This objective will be verified by the performance of the software requirements and code reviews.
Verify correct execution call sequence, including startup sequences.	This objective is met by reviewing the code against the requirements and by testing execution related requirements, with passed results.
Define and evaluate the extent of interface depth	This objective is met by the simplicity and small size of the project and by the code review.
Verifying scheduling	This objective is met by reviewing the code against the requirements and by testing execution related requirements, with passed results.
Worst-case execution time analysis	This analysis will be part of the Software Integration Analysis.

#### 4.11.4.4Outputs of Data and Control Coupling Activity

Data and control coupling outputs are produced as a result of requirements-based testing. The analysis of these outputs is based on a comparison of the module interface software requirements (data and control coupling requirements) plus data and control flow diagrams against the results of the requirements-based testing plus structural coverage analysis results, showing that the testing was complete and the data and control coupling behavior was as expected. This activity will be performed in one of two ways or in combination:

1. Dynamic Activity (Test) - Using a tool that captures the data and control coupling behavior as part of collecting the structural coverage analysis metrics during requirements-based testing. This method involved incrementing the source code and collecting outputs of the tool.
2. Static Activity (Analysis) - Performing a static analysis including analyzing the link map or call tree.

#### 4.12 Process-Specific Activities

The following sections detail the planned process-specific activities of the Testing Process.

##### 4.12.1 Test Case Development

- Test cases will be developed by a person other than the author of the software.
- Test case development can start after the software requirements have been formally reviewed. An iterative process for updating the test cases works in conjunction with any PRs processed to necessary changes in the software requirements.
- Test cases will be developed using software requirements, any certification document, as required for the function being tested, and information from the software detailed design that indicates additional boundary and robustness test steps are required. Additionally, test steps will be iteratively modified when preliminary coverage data is available to address any coverage deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.
- Test case tools will be chosen based on the verification needs identified. Software Simulation tools, specific lab equipment used in validation, and on-target testing tools (script processing tools, external interface stimulation tools) determine the specific steps developed. Refer to the PSAC for a list of verification tools.
- Test cases will be developed that capture test environment setup and parameters, versions of CM controlled Software, versions of CM controlled test documentation (including test cases) and industry interface ICDs for verification of external interfaces.
- Test cases will be developed based on functional interfaces and components. Where applicable, a test case may be used to verify multiple requirements concerning the same function or functions. The software trace matrix supports tracing from test case to software requirement. A test case may cover more than one software requirement, and the test case and trace matrix will indicate all software requirements covered during the test. Each instance of a core function must employ a separate test case with the appropriate tracing to the requirement. All iterative work in the lifecycle is completed using PRs and CM controls.
- Test cases will be developed to include positive path testing, plus additional testing as warranted for robustness. Robustness testing includes boundary conditions, obscure event mitigation, failure compensation, negative path testing, default case verification and more. Developed test cases indicate when test steps are for robustness testing, and may not trace to a specific software requirement. Additionally, test steps are iteratively modified when preliminary coverage data is available to address robustness deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.

### 4.12.2 Test Case Verification

Test cases will be formally reviewed by an independent party against the software requirements claimed in each test step. The trace matrix will be validated during the review to insure proper credit is taken for the software requirements listed. The software development life cycle steps will be followed to insure any discrepancies found in the review are addressed. All iterative work in the lifecycle will be completed using PRs and CM controls. Refer to Peer Review Process for the test cases.

### 4.12.3 Test Procedure Development

- Test procedures will be developed by a person other than the author of the Software.
- Test procedure development can start after the software requirements have been formally reviewed. Test procedures may be developed in conjunction with the test case. An iterative process for updating the test procedures works in conjunction with any PRs processed to necessary changes in the software requirements or related test cases.
- Test procedures will be developed using software requirements, test cases any certification documents as required for the function being tested, and information from the software detailed design that indicates additional boundary and robustness test steps are required. Additionally, test procedures are iteratively modified when preliminary coverage data is available to address any coverage deficiencies. All iterative work in the life cycle is completed using PRs and CM controls.
- Test tools will be chosen based on the verification needs identified.
- Test procedures will be developed that capture test environment setup and parameters, versions of CM controlled software, versions of CM controlled test documentation (including test cases) and industry interface ICDs for verification of external interfaces.
- When test procedure gaps are discovered during testing, the PR process will be used to address the gaps.
- Test procedures will be developed based on functional interfaces and components. Where applicable, a test procedure may be used to verify multiple requirements concerning the same function or functions. Test procedures will be tied directly to a test case – one for one. The software trace matrix will support tracing from test case to software requirement. The test procedure will be an integral part of the test case trace. As discrepancies in test procedures are identified, iterative changes will be made as necessary to resolve the discrepancy. All iterative work in the lifecycle will be completed using PRs and CM controls.

- Test procedures will be developed to include positive path testing, plus additional testing as warranted for robustness. Robustness testing will include boundary conditions, obscure event mitigation, failure compensation, negative path testing, default case verification and more. Test procedures will be developed to indicate when test steps are for robustness testing, and may not trace to a specific software requirement. Additionally, test procedures will be iteratively modified when preliminary coverage data is available to address robustness deficiencies. All iterative work in the lifecycle will be completed using PRs and CM controls.

### 4.12.4 Test Procedure Verification

Test procedures will be formally reviewed by an independent party against the respective test case and software requirements claimed in each test step. The trace matrix will be validated during the review to insure proper credit is taken for the software requirements listed. The software development life cycle steps will be followed to insure any discrepancies found in the review are addressed. All iterative work in the lifecycle will be completed using PRs and CM controls. Refer to Peer Review Process of test cases.

### 4.12.5 Coverage Analysis Verification

Structural coverage analysis results will be formally reviewed by an independent party. Where code structures are not covered by requirements-based testing, the review will ensure that the uncovered code is removed (dead code) or that additional requirements (and related test procedures) are added to address the undocumented functionality or that the required code structure that can't be reached is specifically identified in the Coverage Results and that the behavior of the code structure is deterministic and would not cause unintended behavior (determined by analysis). Refer to Peer Review Process.

#### 4.12.6 Testing Environment

- Each test case will include the following information:
  - Test Description
  - Tester Name
  - Test Date
  - Software Version tested
  - Test Method used
  - Tool(s) Version(s) used (if applicable)
- If appropriate (i.e., special equipment required) the test procedure will describe the specific bench configuration, test tool configuration, and any special instruction required to insure the tester sets up the correct environment.
- If appropriate (i.e., conformed unit, or special test rig) the test procedure will describe the following to insure the proper equipment and rig configuration is achieved before testing
  - P/N of test unit
  - S/N of test unit
  - Identification of special test rig components and gear
- SQA person will audit the test setup before testing.
- Once a test rig or environment has been conformed, the apparatus will be "Locked Down" for the time required to complete the test procedure. ("Locked Down" means the equipment and test gear involved in the test setup is physically or electronically secured from other personnel changing the environment.)

#### 4.12.7 Test Execution

- On-Target testing consists of normal system level test such as TSO, normal flight test simulation and DO-178B requirements based test. Additionally, special test cases will be created to exercise areas of the software where normal system level tests do not obtain full coverage, or configured options on the standard product may not be enabled. All system level testing will be identified in the software trace matrix for evaluation and review.
- Specific test procedures will be designed to exercise timing interfaces, critical data functions and configured options. Validation of the software at the low level will be achieved by capturing artifacts using lab equipment with electronic output. These resultant artifacts will be formally reviewed by an independent source and put under CM control. Data from these tests will also be used in verification by analysis efforts as required based on total test coverage analysis.

- Each test case will include the following test run information:
  - Test Description
  - Tester Name
  - Test Date
  - software Version tested
  - Test Method used
  - Tool(s) Version(s) used (if applicable)
- Testing will commence once the following are complete:
  - All software requirements are reviewed and under CM control with no outstanding (non-deferrable) PRs
  - All Test Cases/Procedures are reviewed and under CM control with no outstanding (non-deferrable) PRs
  - All software source files are reviewed and under CM control with no outstanding (non-deferrable) PRs
  - All traceability data is reviewed and under CM control with no outstanding (non-deferrable) PRs
  - The final software build has been created and is under CM control
- All discrepancies found as a result from formal testing will be handled in one of the following ways:
  - Analysis determines the test case/procedure can be modified to produce a more complete result. In this case, an PR is written, and the test case/procedure is updated, reviewed and the test re-executed. The new resultant artifacts are then used in the formal data analysis.
  - Analysis determines the deficiency cannot be mitigated by any formal test means as described above. In this case, the deficiency is formally documented in the test results.
- All gaps in test coverage results will be documented in the test results document deficiencies section. Additional analysis of the software will commence on the areas where the deficiencies are identified. The analysis findings will be documented as additional coverage information in the results document.

### 4.12.8 Test Results Verification

When all testing is complete, and the results have been evaluated and documented, the formal findings will be formally reviewed and put under CM control. Any issues found in the formal review will be documented. If the issues found warrant a change in the test case/procedure, a PR will be used to implement this change. If this PR is not deferrable, the software life cycle will be used to correct any artifacts and re-execute the test procedure and re-evaluated the generated results. Refer to Peer Review Process of test cases.

## 5.0 VERIFICATION ENVIRONMENT

The Software verification environment includes a block diagram of the testing environment, description of the equipment for testing, the testing and analysis tools and the guidelines for applying the tools and hardware test equipment. It also identifies the target test environment including any emulation.

### 5.1 Test Environment Description

<Describe the test environment here>

#### 5.1.1 Block Diagram of Test Environment

### 5.2 List of Test Equipment Used To Verify Software

Description	Manufacturer & Model No.

### 5.3 Testing and Analysis Tools

Description	Manufacturer & Model No.

#### 5.3.1 Guidelines for Applying the Tools and Hardware Test Environment

<Guidelines Here>

#### 5.4 Test Procedure Structure

1. Test Case & Procedure Identifier
2. Test Objective
3. Test Coverage

Test Case No.	Test Type	Requirement(s) Tested
Case 001	Normal	000.0000
Case 002	Normal / Robust	000.0000 000.0000 000.0000
Case 003	Normal	000.0000
Case 004	Normal	000.0000
Case 005	Normal	000.0000
Case 006	Normal	000.0000
Case 007	Robust	000.0000

4. Assumptions
5. Constraints
6. Special Requirements
7. Execution Summary

### Setup Identification

Test Date: \_\_\_\_\_

Test Operator: \_\_\_\_\_

Test Support Equipment: \_\_\_\_\_

Test Setup: \_\_\_\_\_

Software Under Test: \_\_\_\_\_

### Comments and Notes

### Procedure Results

Check FAIL if the results of any verify statement in this test procedure produced a fail. Otherwise, check PASS.

PASS \_\_\_\_\_ FAIL \_\_\_\_\_

Optional Comment: \_\_\_\_\_

### Signatures

Executed By: \_\_\_\_\_

### Case and Procedure Descriptions

#### Case 001: Valid Set Command

## **6.0 TRANSITION CRITERIA**

Transition criteria for entering the verification process relative to the planning and development processes are included in Section 5. Specific verification activities are carried out when a software data item completes any phase of development. The criterion for beginning a verification activity is the indication, by the engineer responsible for the production of the data item, that the item is ready for verification. In addition, the item is CM controlled. For-Credit testing cannot be started until the code being tested has been formally reviewed.

Section 4 addresses transition criteria for the following:

- Conditions necessary to consider the verification closed and successful for the Planning Process.
- Conditions necessary to consider the verification closed and successful for the Requirements Process.
- Conditions necessary to consider the verification closed and successful for the Design Process.
- Conditions necessary to consider the verification closed and successful for the Software Coding Process.
- Conditions necessary to consider the verification closed and successful for the Integration Process.
- Conditions necessary to consider the verification closed and successful for the Testing of Outputs of the Integration Process.
- Conditions necessary to consider the verification closed and successful for the Verification of Verification Process Outputs.

## 7.0 PARTITIONING CONSIDERATIONS

Partitioning established that two or more components are protected from the actions of each other. As a definition, partitioning consists of one of the following:

### Strict Protection

Component X can be said to be strictly protected from Y if any behavior of Y has no effect on the operation of X. An example of this type of protection would be two components within a line replaceable unit (LRU) with no communication between them.

### Safety Protection

Component X can be said to be safely protected from Y if any behavior of Y has no effect on the safety properties of X. An example of this would be the use of a Cyclic Redundancy Code around data passed through a non-assured data link. The only safety property of importance would be the corruption of data. Loss of data could not be a safety property of interest in this example. This approach requires the identification of the safety properties that can be derived from the safety analysis/hazard analysis.

### Two-way protection

Component X is protected from Y, and Y is protected from X. An example of this type of protection would be two components within a line replaceable unit (LRU) with no communication between them.

### One-way protection

Component X is protected from Y, but component Y is not protected from X. An example of this would be a computer that can only receive ARINC 429 data from the primary system. In this case, the primary software could affect the maintenance software but the maintenance software would not be able to interfere with the primary software.

## 7.1 Guidelines for Evaluating Protection

A component can effect the operation of other components by affecting the temporal (time) behavior or the data (space) of the other components. The project team first categorizes the type of protection claimed according to the definitions specified above. If the project team's approach to protection is to separate (partition) components in both time and space, then the project team is required to demonstrate the partitioning in time and space between the two components to demonstrate either one-way or two-way strict protection.

However, if the project team proposes to use safety protection, then the team must identify all the safety properties of time and space which could be affected and then demonstrate that the safety properties have not been violated. In evaluating time properties, the following items are considered by SQE as appropriate to the design:

### 7.1.1 Time

The following items can affect the time parameters of a program and need to be investigated to demonstrate that they either have no effect or that their effect is acceptable based on the identified safety parameters. This list is not intended to be all inclusive.

- Interrupts and interrupt inhibits (software and hardware)
- Loops (e.g. infinite loops)
- Real time correspondence:
  - 1) frame overrun
  - 2) interference with real time clock
  - 3) counter/timer corruption
  - 4) pipeline and caching
- Control Flow defects (timing aspects):
  - 1) Incorrect branching into a partition or protected area
  - 2) Corruption of a jump table (double duty?)
  - 3) Corruption of the processor sequence control
  - 4) Corruption of return addresses
  - 5) Unrecoverable hardware state corruption (e.g., mask and halt)
  - 6) Memory, I/O contention
  - 7) Data flags
- Software traps:
  - 1) Divide by zero
  - 2) Un-implemented instruction
  - 3) Specific software interrupt instructions
  - 4) Unrecognized instruction
  - 5) Recursion termination
  - 6) Indirect non terminating call loops
  - 7) Holdup commands (performance hedges)

### 7.1.2 Space

The following items can affect the space parameters of a program and need to be investigated to demonstrate that they either have no effect or that their effect is acceptable based on the identified safety parameters. This list is not intended to be all inclusive.

- Loss of input or output data
- Corruption of input or output data
- Corruption of internal data:
  - 1) Direct or indirect memory writes
  - 2) Table overrun
  - 3) Incorrect linking
  - 4) Calculations involving time
  - 5) Delayed data
  - 6) Program overlays
  - 7) Buffer sequence (double jeopardy)
- External device interaction (e.g. displays):
  - 1) Loss of data (e.g. overwritten)
  - 2) Delayed data
  - 3) Incorrect data (unlikely across systems)
  - 4) Protocol halts (e.g. ack nacks)
- Control Flow defects (space aspects):
  - 1) Incorrect branching into a partition or protected area
  - 2) Corruption of a jump table (double duty?)
  - 3) Corruption of the processor sequence control
  - 4) Corruption of return addresses
  - 5) Unrecoverable hardware state corruption (e.g., mask and halt)

### 7.2 Project Specific Partitioning

The project will not use partitioning techniques. All source code will be developed and verified in accordance with applicable DO-178C objectives.

## **8.0 COMPILER ASSUMPTIONS**

The C language was chosen because it is an industry standard for embedded applications. This makes a wide variety of code development tools available for use. The nature of the C language is such that any ANSI-C compatible compiler for the target processor will be acceptable. If compiler optimizations are used, they are specifically accounted for in the software verification overview in the program's Plan for Software Aspects of Certification. In addition, the project team relies on the DO-178C process to ensure the validity of the compiler.

## 9.0 REVERIFICATION GUIDELINES

Once a modification has been implemented into the source code, reverification guidelines are implemented. These guidelines include reviews, inspections, walkthroughs, analysis, and tests of software. They are divided into three specific tasks:

### 9.1 Inspect, Review, or Analyze Changes

This task includes many of the non-testing aspects of the verification process (i.e., reviews, analysis, inspections, and walkthroughs). In this task, the software life cycle data (e.g., requirements, design, architecture, code, test cases, and procedures) are reviewed for accuracy and consistency.

### 9.2 Perform Regression Testing

Regression testing is another aspect of the verification process that is addressed when software changes. Software progresses through several versions before one is ready for release. Regression testing is performed on each version of software. Any specific change can (a) fix only the problem that was reported, (b) fail to fix the problem, (c) fix the problem but adversely affect some other function or aspect that was previously working, or (d) fail to fix the problem and adversely affect something else. Since it is not always possible to re-run every test on every version of software, analysis is used to determine which tests should be run on the interim versions. The following “types” of tests are performed during the regression testing process:

- Bug verification tests – run to verify that the fix for a bug addresses the problem and doesn’t introduce additional problems.
- Build acceptance tests – tests run to make sure that a build is ready to go to the test team.
- Regression test pass with a regression test suite – running regressions tests that have been automated.
- Regression test pass on closed bugs – rerunning the regression tests even after the bugs have been “fixed”. Includes robustness tests.
- Regression test pass without a test suit – manually running regression tests. Includes robustness tests.

It is a standard practice that the most important tests are run first in order to quickly validate operation and assess risks. In some cases, regression testing may be run in parallel with other development activities. Generally, a test that has passed twice should be considered as regressed, unless the code has been frequently changed. A test that has failed once should not be re-executed unless the developer informs the test team that the defect has been fixed.

For tests that have already passed once, the second execution is reserved for the final regression pass, unless frequent changes to the code indicate otherwise. In every case, the change impact analysis must lead to a set of regression tests that are unique to the set of changes being proposed. All of these identified regression tests are run on the final version of the software prior to release.

### 9.3 Perform Other Verification

In addition to the inspections, reviews, analysis, and regression testing, other types of verification activities are performed. Some of these activities are performed at the software level, some at the system level, and some at the integrated system level. For example, requirements-based tests, acceptance tests, bench tests, structural coverage analysis, etc. may need to be performed. These additional verification activities will vary, depending on the extent of the change and the function(s) affected. In all cases, these tests are planned after the change impact analysis is performed and agreed upon.

## **10.0 PREVIOUSLY DEVELOPED SOFTWARE**

Previously developed software includes software that was developed under a different standard, such as MIL-STD-2167A. It also includes software that was developed under a previous revision of DO-178C. When this occurs, certification credit may be requested.

There was no previously developed software used in the program. All software was developed under DO-178C objectives.

### **11.0 MULTIPLE VERSION DISSIMILIAR SOFTWARE**

Multiple version dissimilar software is a set of two or more programs developed separately to satisfy the same functional requirements. This has proven to be an effective method for implementing software redundancy. Errors specific to one of the versions are detected by comparison of the multiple outputs.

This project does not contain multiple version dissimilar software.

**APPENDIX A: SOFTWARE PLANNING REVIEW CHECKLIST**

The complete Software Planning Review objectives and activities checklist is provided below. The Software Planning Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	The activities of the software development processes and integral processes of the software life cycle that will address the system requirements and software level(s) have been defined.	4.1a, 4.3
6	The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria have been determined.	4.1b, 4.3
7	The software life cycle environment, including the methods and tools to be used for the activities of each software life cycle process have been selected.	4.1c
8	Additional considerations, such as those discussed in section 12, have been addressed.	4.1d
9	Software development standards consistent with the system safety objectives for the software to be produced have been defined.	4.1e
10	Software plans that comply with subsection 4.3 and section 11 have been produced.	4.1f, 4.6
11	Development and revision of the software plans have been coordinated.	4.1g, 4.6
12	Archive, retrieval, and release have been established.	7.2.7
13	The means of compliance has been proposed and FAA ACO is in agreement with the Plan for Software Aspects of Certification is obtained.	9.1
14	The software plans were developed at a point in time in the software life cycle that provided timely direction to the personnel performing the software development processes and integral processes.	4.2a
15	The software development standards to be used for the project have been defined or selected.	4.2b

ID	Checklist Item	Reference
16	Methods and tools have been chosen that provide error prevention in the software development processes.	4.2c
17	The software planning process provides coordination between the software development and integral processes to provide consistency among strategies in the software plans.	4.2d
18	The software planning process includes a means to revise the software plans as a project progresses.	4.2e
19	If multiple-version dissimilar software is used in a system, the software planning process includes the methods and tools to achieve the error avoidance or detection necessary to satisfy the system safety objectives.	4.2f
20	The software plans and software development standards are under change control and reviews of them completed.	4.2g
21	If deactivated code is planned, the software planning process describes how the deactivated code (selected options, flight test) will be defined, verified and handled to achieve system safety objectives.	4.2h
22	If user-modifiable code is planned, the process, tools, environment, and data items substantiating the guidelines of paragraph 5.2.3 are specified in the software plans and standards.	4.2i
23	A means for detecting object code that is not directly traceable to the source code and a means to ensure its verification coverage are defined.	4.4.2b
24	The SCM process provides a defined and controlled configuration of the software throughout the software life cycle.	7.1a
25	The SCM process provides the ability to consistently replicate the Executable Object Code for software manufacturing or to generate it in case of a need for investigation or modification.	7.1b
26	The SCM process provides control of process inputs and outputs during the software life cycle that ensures consistency and repeatability of process activities.	7.1c
27	The SCM process provides a known point for review, assessing status, and change control by control of configuration items and the establishment of baselines.	7.1d
28	The SCM process provides controls that ensure problems receive attention and changes are recorded, approved, and implemented.	7.1e
29	The SCM process provides evidence of approval of the software by control of the outputs of the software life cycle processes.	7.1f
30	The SCM process provides an assessment of the software product compliance with requirements.	7.1g
31	The SCM process ensures that secure physical archiving, recovery and control are maintained for the configuration items.	7.1h

ID	Checklist Item	Reference
32	A process exists that ensures that configuration identification will be established for the software life cycle data.	7.2.1a
33	A process exists that ensures that configuration identification will be established for each configuration item, for each separately controlled component of a configuration item, and for combinations of configuration items that comprise a software product.	7.2.1b
34	A process exists that ensures that configuration items will be configuration-identified prior to the implementation of change control and traceability data recording.	7.2.1c
35	A process exists that ensures that a configuration item will be configuration-identified before that item is used by other software life cycle processes, referenced by other software life cycle data, or used for software manufacture or software loading.	7.2.1d
36	If the software product identification cannot be determined by physical examination (for example, part number plate examination), then a process exists that ensures that Executable Object Code will contain configuration identification which can be accessed by other parts of the airborne system or equipment.	7.2.1e
37	A process exists that ensures that baselines will be established for configuration items used for certification credit. (Intermediate baselines may be established to aid in controlling software life cycle process activities.)	7.2.2a
38	A process exists that ensures that a software product baseline will be established for the software product and defined in the Software Configuration Index.	7.2.2b
39	A process exists that ensures that baselines will be established in controlled software libraries (physical, electronic, or other) to ensure their integrity. Once a baseline is established, it will be protected from change.	7.2.2c
40	A process exists that ensures that change control activities are followed when developing a derivative baseline from an established baseline.	7.2.2d
41	A process exists that ensures that baselines will be traceable to the baseline from which it was derived.	7.2.2e
42	A process exists that ensures that a configuration item will be traceable to the configuration item from which it was derived.	7.2.2f
43	A process exists that ensures that each baseline or configuration item will be traceable either to the output it identifies or to the process with which it is associated.	7.2.2g
44	A process exists that ensures that problem reports will be prepared that describes the process non-compliance with plans, output deficiency, or software anomalous behavior, and the corrective action taken.	7.2.3a

ID	Checklist Item	Reference
45	A process exists that ensures that problem reports will include configuration identification of affected configuration items(s) or definition of affected process activities, status reporting or problem reports, and approval and closure of problem reports.	7.2.3b
46	A process exists that ensures that problem reports requiring corrective action of the software product or outputs of software life cycle processes invoke the change control activity.	7.2.3c
47	The documented change control process will preserve the integrity of the configuration items and baselines by providing protection against their change.	7.2.4a
48	The change control process ensures that any change to a configuration item requires a change to its configuration identification.	7.2.4b
49	A process exists that ensures that changes to baselines and to configuration items under change control will be recorded, approved, and tracked. Problem reporting is related to change control, since resolution of a reported problem may result in changes to configuration items or baselines.	7.2.4c
50	A process exists that ensures that software changes will be traced to their origin and the software life cycle processes repeated from the point at which the change affects their outputs. (For example, an error discovered at hardware/software integration, that is shown to result from an incorrect design, should result in design correction, code correction and repetition of the associated integral process activities.)	7.2.4d
51	A process exists that ensures that throughout the change activity, software life cycle data affected by the change should be updated and records should be maintained for the change control activity.	7.2.4e
52	The change review activity includes a confirmation that affected configuration items are configuration identified.	7.2.5a
53	The change review activity includes an assessment of the impact on safety-related requirements with feedback to the system safety assessment process.	7.2.5b
54	The change review activity includes an assessment of the problem or change, with decisions for action to be taken.	7.2.5c
55	The change review activity includes feedback of problem report or change impact and decisions to affected processes.	7.2.5d
56	The status accounting activity includes reporting on configuration item identification, baseline identification, problem reporting status, change history, and release status.	7.2.6a
57	The status accounting activity includes a definition of the data to be maintained and the means of recording and reporting status of this data.	7.2.6b

ID	Checklist Item	Reference
58	A process exists that ensures that software life cycle data associated with the software product will be retrievable from the approved source.	7.2.7a
59	Procedures have been established to ensure the integrity of the stored data (regardless of medium of storage) by: <ol style="list-style-type: none"> <li>1. Ensuring that no unauthorized changes can be made.</li> <li>2. Selecting storage media that minimize regeneration errors or deterioration.</li> <li>3. Exercising and/or refreshing archived data at a frequency compatible with the storage life of the medium.</li> <li>4. Storing duplicated copies in physically separate archives that minimize the risk of loss in the event of a disaster.</li> </ol>	7.2.7b
60	The duplication process will be verified to produce accurate copies and procedures exist that ensure error-free copying of the Executable Object Code.	7.2.7c
61	A process exists that ensures that configuration items will be identified and released prior to use of software manufacture and the authority for their release should be established. As a minimum, the components of the software product loaded into the airborne system or equipment (which includes the Executable Object code and may also include associated media for software loading) will be released.	7.2.7d
62	Data retention procedures have been established to satisfy airworthiness requirements and enable software modification.	7.2.7e
63	All Review checklist items have been addressed and marked?	NA
64	All action items have been entered into QCMS?	NA
65	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX B: SOFTWARE REQUIREMENTS REVIEW CHECKLIST**

The complete Software Requirements Review objectives and activities checklist is provided below. The Software Requirements Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	High-level requirements have been developed.	5.1.1a
6	Derived high-level requirements have been defined and have been indicated to the System Safety Assessment Process.	5.1.1b
7	System functions to be performed by the software have been defined and the functional, performance, and safety-related requirements of the system are satisfied by the software high-level requirements, and that derived requirements and the reason for their existence have been correctly defined.	6.3.1a
8	Each high-level requirement is accurate, unambiguous and sufficiently detailed and the requirements do not conflict with each other.	6.3.1b
9	No conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially, system response times and input/output hardware.	6.3.1c
10	Each high-level requirement can be verified.	6.3.1d
11	The Software Requirements Standard were followed during the software requirements process and deviations from the standards are justified.	6.3.1e
12	The functional, performance, and safety-related requirements of the system that are allocated to software were developed into the software high-level requirements.	6.3.1f
13	Configuration items have been identified.	7.2.1
14	Baselines and traceability have been established.	7.2.2
15	Problem reporting, change control, change review, and configuration status accounting have been established.	7.2.3 - 7.2.6

ID	Checklist Item	Reference
16	The system functional and interface requirements that are allocated to software have been analyzed for ambiguities, inconsistencies and undefined conditions.	5.1.2a
17	Input to the software requirements process detected as inadequate or incorrect have been reported as feedback to the input source processes for clarification correction.	5.1.2b
18	Each system requirement that is allocated to software has been specified in the high-level requirements.	5.1.2c
19	High-level requirements that address system requirements allocated to software to preclude system hazards have been defined.	5.1.2d
20	The high-level requirements conform to the Software Requirements Standards, and are verifiable and consistent.	5.1.2e
21	The high-level requirements are stated in quantitative terms with tolerances where applicable.	5.1.2f
22	The high-level requirements do not describe design or verification detail except for specified and justified design constraints.	5.1.2g
23	Each system requirement allocated to software is traceable to one or more software high-level requirements.	5.1.2h
24	Each high-level requirement is traceable to one or more system requirements, except for derived requirements.	5.1.2i
25	Derived high-level requirements have been provided to the system safety assessment process.	5.1.2j
26	High-level requirements and traceability to those high-level requirements have been verified.	6.2a
27	All Review checklist items have been addressed and marked?	NA
28	All action items have been entered into QCMS?	NA
29	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX C: SOFTWARE PRELIMINARY DESIGN REVIEW CHECKLIST**

The complete Software Preliminary Design Review objectives and activities checklist is provided below. The Software Preliminary Design Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	The software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.	6.3.3a
6	The accuracy and behavior of the proposed algorithms have been verified, especially in the area of discontinuities.	6.3.1g
7	The software architecture was developed from the high-level requirements.	5.2.1a
8	A correct relationship exists between the components of the software and the architecture. This relationship exists via data flow and control flow.	6.3.3b
9	No conflicts exist in the architecture, especially initialization, asynchronous operation, synchronization and interrupts, between the software architecture and the hardware/software features of the target computer.	6.3.3c
10	The software architecture can be verified (e.g., there are no unbounded recursive algorithms).	6.3.3d
11	The Software Design Standards were followed during the software design process and deviations to the standards were justified, especially complexity restrictions and design constructs that would not comply with the system safety objectives.	6.3.3e
12	All Review checklist items have been addressed and marked?	NA
13	All action items have been entered into QCMS?	NA

ID	Checklist Item	Reference
14	<p>The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?</p> <p>Naming Example: SS1060.pdf</p>	NA

**APPENDIX D: SOFTWARE CRITICAL DESIGN REVIEW CHECKLIST**

The complete Software Critical Design Review objectives and activities checklist is provided below. The Software Critical Design Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	The software low-level requirements satisfy the software high-level requirements and derived requirements and the design basis for their existence were correctly defined.	6.3.2a
6	Each low-level requirement is accurate and unambiguous and the low-level requirements do not conflict with each other.	6.3.2b
7	No conflicts exist between the software requirements and hardware/software features of the target computer, especially, the use of resources (such as bus loading), system response times, and input/output hardware.	6.3.2c
8	Each low-level requirements can be verified.	6.3.2d
9	The Software Design Standards were followed during the software design process, and deviations from the standards were justified.	6.3.2e
10	The high-level requirements and derived requirements were developed into the low-level requirements.	6.3.2f
11	The accuracy and behavior of the proposed algorithms, especially in the area of discontinuities have been verified.	6.3.2g
12	Low-level requirements were developed from high-level requirements.	5.2.1a
13	Derived low-level requirements have been defined and provided to the System Safety Assessment Process.	5.2.1b
14	Partitioning beaches have been prevented or isolated.	6.3.3f
15	Low-level requirements and software architecture developed during the software design process conform to the Software Design Standards and are traceable, verifiable and consistent.	5.2.2a

ID	Checklist Item	Reference
16	Derived requirements have been defined and analyzed to ensure that the high level requirements are not compromised.	5.2.2b
17	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 has been defined as derived requirements and proved the system safely assessment process.	5.2.2c
18	Control flow and data flow have been monitored when safety-related requirements dictate (e.g., watchdog timers, reasonableness-checks and cross-channel comparisons).	5.2.2d
19	Responses to failure conditions are consistent with the safety-related requirements.	5.2.2e
20	Inadequate or incorrect inputs detected during the software design process have been provided to either the system life cycle process, the software requirements process, or the software planning process as feedback for clarification or correction.	5.2.2f
21	All Review checklist items have been addressed and marked?	NA
22	All action items have been entered into QCMS?	NA
23	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX E: SOFTWARE CODE REVIEW CHECKLIST**

The complete Software Code Review objectives and activities checklist is provided below. The Software Code Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Source Code has been developed that is traceable, verifiable, consistent and correctly implements low-level requirements.	5.3.1a
6	The Source Code is accurate and complete with respect to the software low-level requirements, and no Source Code implements and undocumented function.	6.3.4a
7	The Source Code matches the data flow and control flow defined in the software architecture.	6.3.4b
8	The Source Code does not contain statements and structures that cannot be verified and the code does not have to be altered to test it.	6.3.4c
9	The Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives.	6.3.4d
10	The software low-level requirements were developed into Source Code.	6.3.4e
11	Verification evidence exists that ensures the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of uninitialized variables of constants, unused variables or constraints, and data corruption due to task or interrupt conflicts.	6.3.4f
12	Software load control has been established.	7.2.8
13	Software life cycle environment control has been established.	7.2.9
14	The Source Code implements the low-level requirements and conforms to the software architecture.	5.3.2a

ID	Checklist Item	Reference
15	The Source Code conforms to the Software Code Standards.	5.3.2b
16	The Source Code is traceable to the Design Description.	5.3.2c
17	Inadequate or incorrect inputs detected during the software coding process have been provided to the software requirements process, software design process or software planning process as feedback for clarification or correction.	5.3.2d
18	The results of the traceability analyses and requirements-based and structural coverage analyses show that each software requirement is traceable to the code that implements it and to the review, analysis, or test case that verifies it.	6.2b
19	Software load control, which includes procedures for part numbering and media identification that identify software configurations that are intended to be approved for loading into the airborne system or equipment, has been established.	7.2.8a
20	Software load control, which includes whether the software is delivered as an end item or is delivered installed in the airborne system or equipment, records should be kept that confirm software compatibility with the airborne system or equipment hardware, has been established.	7.2.8b
21	Configuration identification has been established for the Executable Object Code (or equivalent) of the tools used to develop, control, build, verify, and load the software.	7.2.9a
22	The SCM process for controlling qualified tools, complies with the objectives associated with Control Category 1 or 2 data.	7.2.9b
23	The SCM process for controlling the Executable Object Code (or equivalent) of tools used to build and load the software (for example, compilers, assemblers, linkage editors) complies with the objectives associated with Control Category 2 data, as a minimum.	7.2.9c
24	All Review checklist items have been addressed and marked?	NA
25	All action items have been entered into QCMS?	NA
26	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX F: INTEGRATION REVIEW CHECKLIST**

The complete Integration Review objectives and activities checklist is provided below. The Integration Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Verification evidence that the Executable Object Code has been successfully loaded into the target hardware for hardware / software integration has been produced.	5.4.1a
6	The test cases have been accurately developed into test procedures and expected results.	6.3.6b
7	Output of software integration process is complete and correct.	6.3.5
8	Executable Object Code is compatible with target computer.	6.4.3a
9	Verification evidence exists that the Executable Object Code can be generated from the Source Code and linking and loading data.	5.4.2a
10	Verification evidence exists that the software has been successfully loaded into the target computer for hardware/software integration.	5.4.2b
11	Inadequate or incorrect inputs detected during the integration process have been provided to the software requirements process, the software design process, the software coding process or the software planning process as feedback for clarification or correction.	5.4.2c
12	Software integration testing has been performed to verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.	6.4
13	Hardware/Software integration testing has been performed to verify correct operation of the software in the target computer environment.	6.4
14	Low-level testing has been performed to verify the implementation of software low-level requirements.	6.4

ID	Checklist Item	Reference
15	All Review checklist items have been addressed and marked?	NA
16	All action items have been entered into QCMS?	NA
17	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX G: SOFTWARE VERIFICATION REVIEW CHECKLIST**

The complete Software Verification Review objectives and activities checklist is provided below. The Software Verification Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	The test results have been verified to be correct and discrepancies between actual and expected results are explained.	6.3.6c
6	Executable Object Code complies with low-level requirements.	6.4.2.1, 6.4.3
7	Executable Object Code complies with high-level requirements.	6.4.2.1, 6.4.3
8	Executable Object Code is robust with low-level requirements.	6.4.2.2, 6.4.3
9	Executable Object Code is robust with high-level requirements.	6.4.2.2, 6.4.3
10	Test coverage of high-level requirements has been achieved.	6.4.4.1
11	Test coverage of low-level requirements has been achieved.	6.4.4.1
12	Test coverage of software structure (modified condition/decision coverage) has been achieved.	6.4.4.2a, 6.4.4.2b
13	Test coverage of software structure (decision coverage) has been achieved.	6.4.4.2a, 6.4.4.2b
14	Test coverage of software structure (statement coverage) has been achieved.	6.4.4.2a, 6.4.4.2b
15	Test coverage of software structure (data coupling and control coupling) has been achieved.	6.4.4.2c
16	Software development processes and integral processes comply with approved software plans and standards.	8.1a
17	The transition criteria for the software life cycle processes have been satisfied.	8.1b
18	Communication and understanding between the applicant and the certification authority has been established and maintained.	9.0
19	Compliance substantiation has been provided.	9.2

ID	Checklist Item	Reference
20	If the code tested is not identical to the airborne software, those differences have been specified and justified.	6.2c
21	When it was not possible to verify specific software requirements by exercising the software in a realistic test environment, other means were provided and their justification for satisfying the software verification process objectives are recorded in the Software Verification Results.	6.2d
22	Deficiencies and errors discovered during the software verification process have been reported to the software development processes for clarification and correction.	6.2e
23	Software integration testing has been performed to verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.	6.4
24	Hardware/Software integration testing has been performed to verify correct operation of the software in the target computer environment.	6.4
25	Low-level testing has been performed to verify the implementation of software low-level requirements.	6.4
26	Objective evidence exists that normal range test cases were performed that demonstrate the ability of the software to respond to normal inputs and conditions which include real and integer input variables were exercised using valid equivalence classes and boundary values.	6.4.2.1a
27	Objective evidence exists that normal range test cases were performed that demonstrate the ability of the software to respond to normal inputs and conditions which include for time-related functions, such as filters, integrators and delays, multiple iterations of the code were performed to check the characteristics of the function in context.	6.4.2.1b
28	Objective evidence exists that normal range test cases were performed that demonstrate the ability of the software to respond to normal inputs and conditions which include for state transitions, test cases were developed to exercise the transitions possible during normal operation.	6.4.2.1c
29	Objective evidence exists that normal range test cases were performed that demonstrate the ability of the software to respond to normal inputs and conditions which include for software requirements expressed by logic equations, the normal range test cases verified the variable usage and the Boolean operators.	6.4.2.1d

ID	Checklist Item	Reference
30	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions which include c. The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.	6.4.2.2a
31	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions which included for loops where the loop count is a computed value, test cases were developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.	6.4.2.2a
32	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions: Real and integer variables were exercised using equivalence class selection of invalid values.	6.4.2.2a
33	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions: System initialization was exercised during abnormal conditions.	6.4.2.2a
34	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions: A check was made to ensure that protection mechanisms for exceeded frame times respond correctly.	6.4.2.2a
35	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions: For time or time-related functions, such as filters, integrators and delays, test cases were developed for arithmetic overflow protection mechanisms.	6.4.2.2a
36	Objective evidence exists that robustness test cases were performed that demonstrate the ability of the software to respond to abnormal inputs and conditions: For state transitions, test cases were developed to provoke transitions that are not allowed by the software requirements.	6.4.2.2a
37	Objective evidence exists that for shortcomings in requirements-based test cases or procedures, the test cases were supplemented or tested procedures changed to provide the missing coverage. The method(s) used to perform the requirements-based coverage analysis may need to be reviewed.)	6.4.4.3a

ID	Checklist Item	Reference
38	Objective evidence exists that for shortcomings in software requirements, the software requirements were modified and additional test cases developed and test procedures executed.	6.4.4.3b
39	Objective evidence exists that for dead code, the code was removed and an analysis performed to assess the effect and the need for re-verification.	6.4.4.3c
40	Objective evidence exists that for deactivated code, there are a couple verifications: For deactivated code which is not intended to be executed in any configuration used within an aircraft or engine, a combination of analysis and testing shows that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated. For deactivated code which is only executed in certain configurations of the target computer environment, the operational configuration needed for normal execution of this code was established and additional test cases and test procedures developed to satisfy the required coverage objectives.	6.4.4.3d
41	Objective evidence exists that the SQA process has take an active role in the activities of the software life cycle processes, and have those performing the SWA process enabled with the authority, responsibility and independence to ensure that the SQA process objectives are satisfied.	8.2a
42	Objective evidence exists that the SQA process provides assurance that software plans and standards are developed and reviewed for consistency.	8.2b
43	Objective evidence exists that the SQA process provided assurance that the software life cycle processes comply with the approved software plans and standards.	8.2c

**APPENDIX H: SOFTWARE CONFORMITY REVIEW CHECKLIST**

The complete Software Conformity Review objectives and activities checklist is provided below. The Software Conformity Review Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable objectives, activities and lifecycle data of a particular design assurance level. The checklists also include the applicable control categories.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?	NA
3	A person has been assigned to document action items in QCMS? It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	A conformity review of the software product will be conducted following this review.	8.1c, 8.3
6	If certification credit was sought for the use of previously developed software, objective evidence exists that the current software product baseline is traceable to the previous baseline and the approved changes to that baseline.	8.3i
7	Planned life cycle process activities for certification credit, including the generation of software life cycle data, have been completed and records of their completion are retained.	8.3a
8	Software life cycle data developed from specific system requirements, safety-related requirements, or software requirements are traceable to those requirements.	8.3b
9	Software life cycle data complies with software plans and standards, and is controlled in accordance with the SCM Plan.	8.3c
10	Problem reports comply with the SCM Plan, have been evaluated and have their status recorded.	8.3d
11	Software requirement deviations are recorded and approved.	8.3e
12	The Executable Object Code can be generated from the archived source code.	8.3f
13	The approved software can be loaded successfully through the use of released instructions.	8.3g
14	Problem reports deferred from a previous software conformity review are re-evaluated to determine their status.	8.3h

ID	Checklist Item	Reference
15	Planned software life cycle process activities for certification credit, including the generation of software life cycle data, have been completed and records of their completion are retained.	8.3a
16	Software life cycle data developed from specific system requirements, safety-related requirements, or software requirements are traceable to those requirements.	8.3b
17	Software life cycle data complies with software plans and standards, and is controlled in accordance with the SCM plan.	8.3c
18	Problem reports comply with the SCM Plan, have been evaluated and have their status recorded.	8.3d
19	Software requirement deviations are recorded and approved.	8.3e
20	The Executable Object Code can be regenerated from the archived Source Code.	8.3f
21	The approved software can be loaded successfully through the use of released instructions.	8.3g
22	Problem reports deferred from a previous software conformity review are re-evaluated to determine their status.	8.3h
23	If certification credit is sought for the use of previously developed software, the current software product baseline is traceable to the previous baseline and the approved changes to that baseline.	8.3i
24	All Review checklist items have been addressed and marked?	NA
25	All action items have been entered into QCMS?	NA
26	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX I: PEER REVIEW CHECKLIST - PLANNING**

The complete Planning Peer Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Has the Plan for Software Aspects of Certification (PSAC) been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
6	Has the Software Development Plan (SDP) been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
7	Has the Software Verification Plan (SVP) been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
8	Has the Software Configuration Management Plan (SCMP) been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
9	Has the Software Quality Assurance Plan (SQAP) been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA

ID	Checklist Item	Reference
10	Has the Software Requirements Standards document been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
11	Has the Software Design Standards document been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
12	Has the Software Coding Standards document been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies?	NA
13	If there are any other plan documents for the project, have they been reviewed, with the review records recorded in DRMS, with all comments closed with approved modifications to the reviewed material to correct review deficiencies? (Identify the specific plan in the comment block)	NA
14	Has the Plan for Software Aspects of Certification (PSAC) been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
15	Has the Software Development Plan (SDP) been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
16	Has the Software Verification Plan (SVP) been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
17	Has the Software Configuration Management Plan (SCMP) been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
18	Has the Software Quality Assurance Plan (SQAP) been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
19	Has the Software Requirements Standards document been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
20	Has the Software Design Standards document been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA

ID	Checklist Item	Reference
21	Has the Software Coding Standards document been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)?	NA
22	If there are any other plan documents for the project, have they been signed and released into the project's Configuration Management system (CC1 or CC2 as appropriate for the software level)? (Identify the specific plan in the comment block)	NA
23	Does the Plan for Software Aspects of Certification (PSAC) content comply with DO-178C Section 11.1?	NA
24	Does the Software Development Plan (SDP) content comply with DO-178C Section 11.2?	NA
25	Does the Software Verification Plan (SVP) content comply with DO-178C Section 11.3?	NA
26	Does the Software Configuration Management Plan (SCMP) content comply with DO-178C Section 11.4?	NA
27	Does the Software Quality Assurance Plan (SQAP) content comply with DO-178C Section 11.5?	NA
28	Does the Software Requirements Standards document content comply with DO-178C Section 11.6?	NA
29	Does the Software Design Standards document content comply with DO-178C Section 11.7?	NA
30	Does the Software Coding Standards document content comply with DO-178C Section 11.8?	NA
31	Is each plan/standard internally consistent?	NA
32	Is the system/software description between the various plans documents consistent? That is, does the text in each plan document appear to be describing the same system?	NA
33	Are the software development and verification life cycle activities defined consistently and in sufficient detail in the planning documents?	NA
34	Are the inputs, activities, transition criteria (entrance and exit), and outputs specified for each process (as appropriate to the software level)? (This includes evaluating the consistency of the specifications between various plans, such as between the PSAC and SDP.)	NA
35	Are all certification basis inputs cited in the plans? (For example, any project that has an FAA Issue Paper that affects software invoked on it should include that Issue Paper, along with DO-178C)	NA

ID	Checklist Item	Reference
36	If the plans and standards are followed as written, would this ensure that all applicable objectives are met (including any additional objectives imposed by the certification authority)?	NA
37	Are the plans and standards written with sufficient clarity to allow project personnel to follow them without further definition?	NA
38	Are the interfaces and communications channels between the software and system development processes addressed in the plans, and are they clearly defined?	NA
39	Are the interfaces and communications channels between the software development and system safety assessment process addressed in the plans, and are they clearly defined?	NA
40	Is all COTS software identified and addressed in the plans?	NA
41	Is user-modifiable software identified and addressed in the plans?	NA
42	Is field-loadable software identified and addressed in the plans?	NA
43	Is option-selectable software identified and addressed in the plans?	NA
44	Is multiple-version dissimilar software identified and addressed in the plans?	NA
45	Are any product service history cIRAMS for certification credit identified and addressed in the plans?	NA
46	Are any proposed alternative methods of compliance identified and addressed in the plans?	NA
47	Are any other applicable additional considerations identified and addressed in the plans?	NA
48	Is the appropriate level of structural coverage identified in the plans, and is the coverage analysis and resolution process clearly identified?	NA
49	Do the plans clearly identify the development and/or verification tools to be used on the project?	NA
50	<p>Do the plans provide rationale for why the identified tools do or do not require qualification?</p> <p>(If any of these 3 questions are answered "no", the given tool does not require qualification:</p> <ol style="list-style-type: none"> <li>1. Can the tool insert and error into the airborne software or fail to detect an existing error within the scope of its intended usage?</li> <li>2. will the tool's output not be verified or confirmed by other verification activities, as specified in Section 6 of RTCA/DO-178C?</li> <li>3. Are processes of DO-178C eliminated, reduced, or automated by the use of the tool?)</li> </ol>	NA

ID	Checklist Item	Reference
51	Is service history claimed for the use of any tool? If yes, has the tool changed, or is the use of the tool different from the cited historical usage? Does the documented tool service history support the intended use of the tool?	NA
52	Are any tools to be qualified supported with a tool qualification plan that conforms to the requirements in DO-178C Section 12.2 (either in the PSAC, or as a separate document)?	NA
53	Are there any unique additional considerations associated with the project (such as unique alternative means or methods of compliance, unique approaches to development/verification/SCM/SQA, etc.) that do not comply with certification authority published policy or issues?	NA
54	Is the use of a Real-Time Operating System (RTOS) planned? If yes, do the plans describe where the RTOS requirements are to be defined, and how they will be traced?	NA
55	Is the use of a Board Support Package (BSP) planned? If yes, do the plans describe where the BSP requirements are to be defined, and how they will be traced?	NA
56	Is an Application Programming Interface (API) planned to be used? If yes, do the plans describe where the API requirements are to be defined, and how they will be traced?	NA
57	Is a device driver planned to be used? If yes, do the plans describe where the device driver requirements are to be defined, and how they will be traced?	NA
58	Is Object-Oriented (OO) design/programming planned to be used? If yes, then are the additional considerations attendant with OO and the specific OO language to be used addressed in the plans?	NA
59	Do the plans describe how and where software performance requirements are defined and how they will be traced? (This includes software timing, size restrictions, throughput, etc.)	NA
60	Do the plans describe how and where fail-safe and fail-operational requirements are defined how they will be traced?	NA
61	Do the plans describe the partitioning scheme, with emphasis on its ability to support the high-level requirements and the software level(s) established by the system safety assessment process?	NA
62	Do the plans describe where system response times are addressed?	NA
63	Do the plans identify the mechanism to be used to determine if the Input / Output (I/O) of the system is adequate?	NA

ID	Checklist Item	Reference
64	Do the plans identify how and where requirements for time-critical tasks are specified? (Time-critical task requirements need to be specified in quantifiable terms.)	NA
65	Do the plans describe where software requirements will address timing constraints, strategy for dealing with timing limits, required timing margins, and methods to be used in measuring timing margins?	NA
66	Do the plans describe where error prevention, fault tolerance, and error detection are to be specified in the requirements, design, and code?	NA
67	Is use of one or more compiler-provided libraries planned? If no, the remaining library questions do not need to be answered.	NA
68	Do the plans identify where the requirements are to be specified, and how they will be traced?	NA
69	Do the plans identify if the source code is available for run-time library functions to be used on the project?	NA
70	Do the plans identify if structural coverage appropriate to the software level will be applied to the libraries?	NA
71	Do the plans describe how library code not used by the application will be dealt with?	NA
72	Do the plans identify if there is dead or deactivated code known to be in the libraries?	NA
73	Do the plans describe how problems found in the library routines will be dealt with by the library developer and the project team?	NA
74	All Review checklist items have been addressed and marked?	NA
75	All action items have been entered into QCMS?	NA
76	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX J: PEER REVIEW CHECKLIST - REQUIREMENTS**

The complete Requirements Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	<p>The following definitions have been discussed and are understood?</p> <p>System Requirements: System requirements are inspection requirements (i.e., weight, measurement, power, etc.) and categories or place holders for groups of high-level requirements (i.e., the system shall process ARINC 429 messages). System requirements are not verifiable through Test or Analysis.</p> <p>Note: System requirement validation is not part of the software life cycle process. The validity of the system requirements should be assured by the system life cycle process. System requirements are included in the software lifecycle to the extent that it identifies those system requirements that will be implemented in software.</p> <p>High-Level Requirements: High-level software requirements are those requirements that are developed from analysis of the system requirements, safety-related requirements, and system architecture. High-level requirements are written as individually verifiable requirements. High-level requirements are verifiable through test or analysis. They are used by Test Engineering to develop requirements-based and robustness test cases. High-level requirements are tested at the “black-box” level.</p> <p>Low-Level Requirements: Low-level software requirements are detailed implementation instructions directed at the Engineer responsible for writing the source code. They are verified through low-level test and analysis. Low-level requirements are typically documented in terms of function descriptions and pseudo code.</p> <p>Derived Requirements: Derived software requirements are additional requirements resulting from the software development processes, which may not be directly traceable to higher level requirements.</p>	NA
2	An attendee list has been generated and circulated for signature?	NA

ID	Checklist Item	Reference
3	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
4	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
5	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
6	Is each high-level requirement uniquely identified (does each entry in the tool contain only one requirement)?	NA
7	Are the high-level requirements unambiguous? (Does each requirement mean the same thing to all stake holders, including the specifier, the systems engineer, the software engineer, the verification engineer, etc?)	NA
8	Is the terminology used in the high-level requirements fully defined? (For example, a requirement that states "The controller shall assert control of the engine within 150 msec of a warm-start" should define what "assert control" means)	NA
9	Are the high-level requirements consistently written (e.g.: terminology attributes, data definitions)?	NA
10	Are the high-level requirements complete, that is, are all system requirements allocated to software reflected in the high-level requirements?	NA
11	Is each high-level requirement verifiable through testing? (This includes clear definition of test parameters, such as defining the time measurement start and stop criteria for the requirements "The controller shall assert control of the engine within 150 msec of a warm-start.")	NA
12	Does each requirement conform to the Software Requirements Standards?	NA
13	Have requirements been reviewed to determine that algorithms are accurate?	NA
14	Are performance requirements (such as response time requirements) stated?	NA

ID	Checklist Item	Reference
15	If the requirements involve complex decision chains, are they expressed in a form that facilitates comprehension?	NA
16	Has the precision and accuracy of calculations been specified?	NA
17	Have Assumptions and Dependencies been clearly stated?	NA
18	Are the high-level requirements consistent with each other? That is, do the requirements NOT conflict or contradict each other?	NA
19	Are the high-level requirements accurate?	NA
20	Is each high-level requirement either traced to one or more specific system requirement(s) or identified as a derived requirement?	NA
21	For high-level requirements traced to one or more specific system requirement(s), does the high-level requirement logically relate to the system requirement(s)?	NA
22	For each system-level requirement allocated to software, are there high-level software requirements that (collectively) cover all aspects of the system requirement?	NA
23	If a Real-Time Operating System (RTOS) is used, are the requirements and interfaces identified and traced?	NA
24	If a Board Support Package (BSP) is used, are the requirements and interfaces identified and traced?	NA
25	If an Application Programmable Interface (API) is used, are the requirements and interfaces identified and traced?	NA
26	If a device driver is used, are the requirements and interfaces identified and traced?	NA
27	Are derived requirements clearly identified?	NA
28	Should any identified derived requirements logically be traced to a higher-level requirement?	NA
29	All Review checklist items have been addressed and marked?	NA
30	All action items have been entered into QCMS?	NA
31	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA



**APPENDIX J: PEER REVIEW CHECKLIST - DESIGN**

The complete Design Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	<p>The following definitions have been discussed and are understood?</p> <p>System Requirements: System requirements are inspection requirements (i.e., weight, measurement, power, etc.) and categories or place holders for groups of high-level requirements (i.e., the system shall process ARINC 429 messages). System requirements are not verifiable through Test or Analysis.</p> <p>Note: System requirement validation is not part of the software life cycle process. The validity of the system requirements should be assured by the system life cycle process. System requirements are included in the software lifecycle to the extent that it identifies those system requirements that will be implemented in software.</p> <p>High-Level Requirements: High-level software requirements are those requirements that are developed from analysis of the system requirements, safety-related requirements, and system architecture. High-level requirements are written as individually verifiable requirements. High-level requirements are verifiable through test or analysis. They are used by Test Engineering to develop requirements-based and robustness test cases. High-level requirements are tested at the “black-box” level.</p> <p>Low-Level Requirements: Low-level software requirements are detailed implementation instructions directed at the Engineer responsible for writing the source code. They are verified through low-level test and analysis. Low-level requirements are typically documented in terms of function descriptions and pseudo code.</p> <p>Derived Requirements: Derived software requirements are additional requirements resulting from the software development processes, which may not be directly traceable to higher level requirements.</p>	NA
2	Is each low-level requirement uniquely identified (does each entry in the tool contain only one requirement)?	NA
3	Are the low-level requirements unambiguous? (Does each requirement mean the same thing to all stake holders, including the specifier, the systems engineer, the software engineer, the verification engineer, etc.)	NA
4	Is the terminology used in the low-level requirements fully defined?	NA

ID	Checklist Item	Reference
5	Are the low-level requirements consistently written (e.g.: terminology attributes, data definitions)?	NA
6	Are the low-level requirements complete, that is, are all high-level requirements reflected in the low-level requirements?	NA
7	Is each low-level requirement verifiable through inspection, analysis, or testing?	NA
8	Does each low-level requirement conform to the Software Design Standards?	NA
9	Have the low-level requirements been reviewed to determine that algorithms are accurate?	NA
10	Are performance requirements (such as timing, size, and throughput) stated?	NA
11	If the requirements involve complex decision chains, are they expressed in a form that facilitates comprehension?	NA
12	Have any real-time constraints been specified in sufficient detail?	NA
13	Has the precision and accuracy of calculations been specified?	NA
14	Are units specified consistently?	NA
15	Are the low-level requirements consistent with each other? That is, do the requirements NOT conflict with or contradict each other?	NA
16	Are the low-level requirements accurate?	NA
17	Has review of the design identified problems with the requirements, such as: * missing requirements? * ambiguous requirements? * extraneous requirements? * untestable requirements? * implied requirements?	NA
18	Is the design consistent with the high-level requirements?	NA
19	Are deviations from the requirements documented and approved?	NA
20	Are all assumptions documented?	NA
21	Have major design decisions been documented?	NA
22	Is the design consistent with the documented major design decisions?	NA
23	Are run-time libraries used in the design? If so, address the following questions.	NA
24	Are the libraries specified?	NA
25	Do the requirements, design, and code exist for the used library functions?	NA
26	Will structural coverage be applied on the libraries or just on features used by the application program?	NA
27	How is code not used by the application dealt with?	NA
28	Is there dead code in the libraries?	NA
29	Have the libraries been verified?	NA
30	Are requirements and design for time-critical tasks specified in quantifiable terms?	NA

ID	Checklist Item	Reference
31	Do software requirements and design address timing constraints, strategy for dealing with timing limits, required timing margins, method of measuring timing margins?	NA
32	If used, is error prevention, fault tolerance, or error detection specified in the requirements, design, and code?	NA
33	If interrupt service routines (ISRs) are used, are they documented in the requirements/design? Do they work properly?	NA
34	Does the ISR - Block any continuing execution?	NA
35	Does the ISR - Call reentrant functions?	NA
36	Does the ISR - Pass stress testing?	NA
37	Does the ISR - Allow calls to functions before completing?	NA
38	Have the common concurrency problems such as Deadlock been addressed?	NA
39	Have the common concurrency problems such as Livelock been addressed?	NA
40	Have the common concurrency problems such as Race conditions been addressed?	NA
41	Have the common concurrency problems such as Re-entrancy been addressed?	NA
42	Have the common concurrency problems such as Priority inversion been addressed?	NA
43	Have the common concurrency problems such as Mutual exclusion violation been addressed?	NA
44	Have the common concurrency problems such as Non-deterministic execution order been addressed?	NA
45	Is partitioning/protection used? If so, is it documented in requirements and design?	NA
46	How is synchronization and communication addressed in the system (e.g., synchronous or asynchronous)? Are the synchronization and communication mechanisms documented in the requirements and design data?	NA
47	If buffers are shared, has the reader-writer (producer-consumer) problem been addressed?	NA
48	Have the common communication problems such as Lost data been addressed?	NA
49	Have the common communication problems such as Stale data been addressed?	NA
50	Have the common communication problems such as System hanging been addressed?	NA
51	Have the common communication problems such as Bounded buffer been addressed?	NA
52	Have the common communication problems such as Corrupted data been addressed?	NA
53	Are critical sections protected? How are they protected? Is the protection adequate and accurately implemented?	NA
54	What kind of scheduling algorithm has been selected for the real-time system? Is the algorithm documented in the requirements and design? Is the scheduling algorithm deterministic and verifiable?	NA
55	If the scheduler uses priorities, does the design detail how priorities are determined?	NA
56	If the scheduler uses priorities, does the design detail what happens when two tasks have the same priority?	NA

ID	Checklist Item	Reference
57	If the scheduler uses priorities, does the design detail how has priority inversion been addressed?	NA
58	If the scheduler uses priorities, does the design detail how are interrupts handled?	NA
59	If concurrent tasks are run, are the handled correctly? (I.e., Is multitasking used?) What algorithms are used to implement concurrency? Are threads used? If threads are used, how do they affect timing?	NA
60	Is there a mechanism to detect when real-time tasks that do not meet their deadlines? If detected, what is the response and is it consistent with the safety requirements?	NA
61	Are fail-safe, fail-operational requirements specified?	NA
62	Is each low-level requirement either traced to one or more specific high-level requirement(s) or identified as a derived requirement?	NA
63	For low-level requirements traced to one or more specific high-level requirement(s), does the low-level requirement logically relate to the high-level requirement(s)?	NA
64	For each high-level requirement, are there low-level software requirements that (collectively) cover all aspects of the high-level requirement? (That is, would implementation of the low-level requirements mean that the high-level requirement is properly	NA
65	Is the low-level to high-level traceability able to be followed but forward (high-to-low) and backward (low-to-high)?	NA
66	Are there any inconsistencies between the data reviewed and the software development plans?	NA
67	Do any conversations with developers indicate that the plans were not followed? (Determine through interview/discussion with developers.)	NA
68	Do verification records exist to demonstrate verification of all applicable design objectives? Were the verification activities thorough and well documented?	NA
69	Is the architecture sufficient to provide service to time-critical tasks?	NA
70	Does the architecture conform to design standards?	NA
71	Is the software architecture compatible with target computer?	NA
72	Does the design adequately address real-time requirements?	NA
73	Does the design adequately address performance issues (memory and timing)?	NA
74	Does the design adequately address spare capacity (CPU and memory)?	NA
75	Does the design adequately address maintainability?	NA
76	Does the design adequately address understandability?	NA
77	Does the design adequately address data requirements?	NA
78	Does the design adequately address loading and initialization?	NA
79	Does the design adequately error handling and recovery?	NA
80	Are memory and timing budgets reasonable and achievable?	NA
81	Is the partitioning schema sufficient to support the high-level requirements and the software level established by the system safety assessment?	NA

ID	Checklist Item	Reference
82	Has partition integrity (i.e., protection) been achieved (in terms of time, space, and throughput)?	NA
83	Does the interrupt/control structure support the known system priorities and high-level requirements?	NA
84	Does the architecture support the timing and sizing requirements?	NA
85	Are the synchronous vs. asynchronous aspects of the design supported by the architecture?	NA
86	Is exception handling properly addressed?	NA
87	Are data flows consistent?	NA
88	Are interfaces consistent?	NA
89	Does the communication mechanism specified in the low-level requirements for each interface support the high-level requirements?	NA
90	Are derived requirements clearly identified?	NA
91	Should any identified derived requirements logically be traced to a higher-level requirement?	NA
92	All Review checklist items have been addressed and marked?	NA
93	All action items have been entered into QCMS?	NA
94	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX K: PEER REVIEW CHECKLIST - CODE**

The complete Code Peer Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Are naming conventions following the standards?	NA
6	Have all code modules been technically reviewed and are under appropriate configuration management control per the software level?	NA
7	Do all code modules compile without error, and without unacceptable warnings?	NA
8	Does the calling sequence correspond with the software architecture?	NA
9	Does the source code have to be altered to test it?	NA
10	Are the data definitions correct? Consider the following criteria:	NA
11	Data typing is correct and consistent.	NA
12	Units are consistent between modules (e.g., radians, degrees).	NA
13	All variables used are defined prior to use.	NA
14	Data are properly initialized.	NA
15	Global data integrity is assured.	NA
16	Variables are not used for more than one purpose.	NA
17	Does the source code conform to standards? Consider the following areas typically found in standards:	NA
18	Is indentation schema being followed?	NA
19	Are prologue headers per the standards?	NA
20	Is the size of the modules per the standards?	NA
21	Does the code do what the comments say it does?	NA
22	Is there only one entry and exit point?	NA

ID	Checklist Item	Reference
23	Are only the standard coding constructs as defined in the coding standards used?	NA
24	Are nesting considerations being addressed?	NA
25	Has computational correctness been achieved? Consider the following criteria:	NA
26	Sign conventions are consistent and correct.	NA
27	Precision is maintained in mixed mode arithmetic.	NA
28	Desired accuracy is maintained during rounding or truncation.	NA
29	Divide by zero is prohibited/ trapped.	NA
30	Are the logic constructs and data handling correct? Consider the following criteria:	NA
31	Loops are correctly implemented.	NA
32	Subscripts are used properly.	NA
33	Is each loop executed the correct number of times?	NA
34	Will each loop terminate?	NA
35	Will the program terminate?	NA
36	Are all possible loop fall-throughs correct?	NA
37	Are all CASE statements evaluated as expected?	NA
38	Is there any unreachable code?	NA
39	Are there any off-by-one iteration errors?	NA
40	Are there any dangling ELSE clauses?	NA
41	Is pointer addressing used correctly?	NA
42	Are priority rules and brackets in arithmetic expression evaluation used as required to achieve desired results?	NA
43	Are boundary conditions considered? (e.g., null or negative values, adding to an empty list, etc.)	NA
44	Are pointer parameters used as values and vice-versa?	NA
45	Is the number of input parameters equal to the number of arguments?	NA
46	Do parameter and argument attribute match?	NA
47	Do the units of parameters and arguments match?	NA
48	Are any input-only arguments altered?	NA
49	Are global variable definitions consistent across modules?	NA
50	Are any constants passed as arguments?	NA
51	Are any functions called and never returned from?	NA
52	Are all interfaces correctly used as defined in the Software Design	NA
53	Are returned VOID values used?	NA
54	Are data mode definitions correctly used?	NA

ID	Checklist Item	Reference
55	Are data and storage areas initialized before use, correct fields accessed and/or updated?	NA
56	Is data scope correctly established and used?	NA
57	If identifiers with identical names exist at different procedure call levels, are they used correctly according to their local and global scope?	NA
58	Is there unnecessary packing or mapping of data?	NA
59	Are all pointers based on correct storage attributes?	NA
60	Is the correct level of indirection used?	NA
61	Are any string limits exceeded?	NA
62	Are all variables EXPLICITLY declared?	NA
63	Are all arrays, strings, and pointers initialized correctly?	NA
64	Are all subscripts within bounds?	NA
65	Are there any non-integer subscripts?	NA
66	Is the code understandable (i.e., choice of variable names, use of comments, etc.)	NA
67	Is there sufficient and accurate commentary to allow the reader to understand the code?	NA
68	If the program uses deactivated code, answer the following questions:	NA
69	Does the code use a common code deactivation mechanism throughout?	NA
70	Does the deactivation mechanism agree with the software plans?	NA
71	Is the deactivation mechanism clear and understandable to a code reviewer?	NA
72	Does traceability exist between the code and the software low-level requirement?	NA
73	Is the design implemented completely and correctly?	NA
74	Are there missing or extraneous functions?	NA
75	All Review checklist items have been addressed and marked?	NA
76	All action items have been entered into QCMS?	NA
77	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX L: PEER REVIEW CHECKLIST - INTEGRATION**

The complete Integration Peer Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Are the compiler options set according to the project standard for each code file?	NA
6	Does each code file compile without error, and without disallowed warnings?	NA
7	Does the program link without error?	NA
8	Does the linker screen out any compiled code that is not used in the Executable Object Code?	NA
9	Does the link map have any overlapping sections?	NA
10	Are differently scoped memory blocks properly contained in the link map?	NA
11	Are dynamic memory blocks (e.g.: STACK or HEAP) separate from static memory blocks (such as code or variable memory)?	NA
12	Are the hardware addresses in the link map correct?	NA
13	Are there any missing components from the software, according to the link map?	NA
14	All Review checklist items have been addressed and marked?	NA
15	All action items have been entered into QCMS?	NA
16	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory? Naming Example: SS1060.pdf	NA

**APPENDIX M: PEER REVIEW CHECKLIST – TEST PROCEDURES**

The complete Test Procedure Peer Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Is the test environment defined?	NA
6	Does each test have a header that identifies the author, revision date, test objectives, required configuration, and initial setup?	NA
7	Is each test traceable to a specific requirement or requirements?	NA
8	Does the test procedure define the exact sequence of steps required to execute the test?	NA
9	For each test procedure, are the expected results clearly defined?	NA
10	Are the expected results consistent with the requirements?	NA
11	Do the collective test procedures achieve the objectives for the case?	NA
12	Have normal range test cases been developed for all requirements? Example: Verify if A then B.	NA
13	Have negative test cases been developed for all requirements? Example: Verify is NOT A, then NOT B.	NA
14	Do the test cases show positive proof for the occurrence of events whenever possible (e.g.: a variable changes value to show a specific action is taken)?	NA
15	Do test cases against range-based requirements include test to verify the bottom, midpoint and top of the range?	NA
16	Do test cases against range-based requirements where zero is included in the allowed range include test cases near and at the zero value, as appropriate?	NA

ID	Checklist Item	Reference
17	Have a complete set of robustness test cases have been developed?	NA
18	If test cases are run on a simulator or emulator, have any of the test steps been eliminated by the simulator or emulator?	NA
19	Have test cases and procedures been reviewed for correctness?	NA
20	Do the test cases and procedures adhere to the relevant plans and standards? For example, have coding standards, especially those relevant to limitations of structural coverage tools, been followed?	NA
21	Are the test cases and procedures appropriately commented to allow future updates?	NA
22	Have the test cases and procedures been subjected to appropriate change and configuration control?	NA
23	Is the rationale for each test case clearly explained?	NA
24	Do the test cases and procedures specify required input data, expected output data, and input/output data (e.g., temporary stores)?	NA
25	Were the inputs for each test case derived from the requirements (as opposed to being derived from the source code)?	NA
26	Have the appropriate memory locations and variables been preset?	NA
27	Are the test cases and procedures sufficient to meet coverage requirements?	NA
28	Are sufficient tests to provide coverage identified for each logic construct?	NA
29	Are requirements where analysis is required in addition to (or in lieu of) requirements-based testing clearly documented (e.g., requirements for hardware polling)?	NA
30	Will the test results reveal whether the results of the test cases that are counted for credit are observable?	NA
31	Will the test results reveal test cases that violate project standards?	NA
32	Will the test results reveal test cases that are not expected to achieve 100% structural coverage (e.g., hardware polling)?	NA
33	Will the test results specify where further evaluation of specified tolerances is required?	NA
34	Is the separation between test cases clear? For example, are test start and stop identified? This assists tracing the source of unexpected drops in coverage.	NA
35	Does each test case contain inputs, conditions, and expected results?	NA
36	Does each test case have procedures for test set-up (to include environment), test execution, and pass-fail criteria?	NA

ID	Checklist Item	Reference
37	Are test cases that depend on results from previous test cases clearly identified (e.g.: A test case that assumes that variable X is set to a specific value as a result of the previous test case)?	NA
38	If the program uses deactivated code, answer the following question:	NA
39	Do test cases exist to verify the deactivation mechanism?	NA
40	Are the test cases and procedures sufficient to cover all the relevant requirements? That is, do the traceability matrices provide clear association between test cases and requirements?	NA
41	Is the design implemented completely and correctly?	NA
42	Are there missing or extraneous functions?	NA
43	All Review checklist items have been addressed and marked?	NA
44	All action items have been entered into QCMS?	NA
45	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA

**APPENDIX N: PEER REVIEW CHECKLIST – TEST RESULTS**

The complete Test Results Peer Review checklist is provided below. This Checklist is automatically leveled by the Qualtech Compliance Management System. Each checklist includes the applicable review and analysis criteria.

ID	Checklist Item	Reference
1	An attendee list has been generated and circulated for signature?	NA
2	The Review Evaluator has been identified and added to this checklist?  The Review Evaluator is someone other than the person presenting the requirements, design or test data. This documents the independence evidence.	NA
3	A person has been assigned to document action items in QCMS?  It is best practice to add action items directly into QCMS as they occur.	NA
4	All of the data to be reviewed (i.e., Presentations, Excel matrix containing all of the requirements to be reviewed, architectural diagrams, etc.) has been documented and uploaded to the /Review Results Folder?	NA
5	Are the test result files clearly linked to the test procedures and code? (i.e., does configuration control and traceability exist?)	NA
6	Is each test result clearly linked to a test case?	NA
7	Are failed test cases obvious from the test results?	NA
8	Do the test results indicate whether each procedure passed or failed and the final pass/fail results?	NA
9	Do the test results adhere to the relevant plans, standards, and procedures?	NA
10	Have the test results been subjected to appropriate configuration control, per the software level?	NA
11	Is there an acceptable rationale for deviations from expected results, standards, or plans?	NA
12	Are explanations for the failed test cases intelligible?	NA
13	Do explanations for failed test cases contain accurate references to relevant problem reports?	NA
14	Are explanations for code or test rework suitable to address the failure?	NA
15	Have test cases been re-executed in compliance with plans for regression testing?	NA
16	Have the test results from regression testing been documented?	NA

ID	Checklist Item	Reference
17	If the rationale for a failed test case or other deviation from expected results includes a "test stand tolerance" issue, is the test stand generally adequate for running that particular test case?	NA
18	Did any safety-related test case fail?	NA
19	Is 100% structural coverage (as appropriate to the software level) achieved through requirements-based testing?	NA
20	If 100% structural coverage (as appropriate to the software level) is not achieved through requirements-based testing, is there an explanation detailing which parts of the code were not executed and why?	NA
21	Are explanations for drops in coverage sufficiently detailed and acceptable?	NA
22	Are there problem reports associated with dead code?	NA
23	Has dead code been removed?	NA
24	Is deactivated code indicated as NOT exercised?	NA
25	All Review checklist items have been addressed and marked?	NA
26	All action items have been entered into QCMS?	NA
27	The attendee list (with signatures) have been scanned to PDF, properly named and uploaded to the /Signatures directory?  Naming Example: SS1060.pdf	NA