SOFTWARE DEVELOPMENT PLAN (SDP) FOR THE COMMERCIAL SYSTEMS (CS) DATA LINK PROJECTS

Document Number 829-6997-600
Revision J
CAGE Code 0EFD0

Rockwell Collins

Contract Number None

	NAME	TITLE	APPROVAL
Prepared By:	Mohammed Musthafa	Preparer	N/A
Approved By:	Hatem I. Abu-Dagga	Engineering	On File
Approved By:	Lori J. Sipper	Engineering	On File
Approved By:	Eileen P. Roberson	DAC Engineer	On File
Approved By:	Mikael M. Sandoval	System Safety	On File

REVISION HISTORY

VER/ REV	DESCRIPTION	DATE	APPROVED
-200/ -	Original Release	2000-08-31	J. W. Grace
-200/ A	Updated to reflect latest practice. Due to the numerous changes, change bars were not used.	2001-7-11	K. M. Epperson
-200/ B	DLNK200001164 – Added merge information form, updated Appendix A checklists, added text to put CRs in a document's version history, added text to notify SQE of every Peer Review, and added clarification on addressing PC-lint output.	2002-01-07	K. M. Epperson
-300/ -	Updated to support the CMU-900 CNS ATN project.	2001-05-16	K. M. Epperson
-400/ -	Updated to reflect a common SDP for all CMU-900 CNS/ATM projects.	2002-05-17	K. M. Epperson
-500/ -	Updated to reflect a common SDP for all Data Link projects, DO-178B Levels E, D and C. Change Request DLNK0000000005	2008-06-27	L. L. Cantaberry
-500/ A	DLNK000000058 – Allow alternate design approach process as entrance criteria for implementation phase	2008-11-04	D. E. Straub
-500/ B	DLNK000000225 – Update to incorporate DER findings from SOI1 review	2009-05-08	L. L. Cantaberry
-500/ C	Updates per SOI1 findings	2011-09-23	L.L. Cantaberry
-600/ -	FUSN00225569 – Document overhaul as part of "Institutionalizing Data Link Processes & Methods" Lean Event	2012-05-14	D.E. Straub
-600/ A	FUSN00271760 – Incorporate comments received from SOI 1 Review ECO-0409756	2012-10-17	D.E. Straub

VER/ REV	DESCRIPTION	DATE	APPROVED
-600/ B	ECO-0476709 FUSN00392752, FUSN00343965, FUSN00307309, FUSN00315334, FUSN00290139	2014-04-07	B.C Willhite
-600/C	ECO-0491980 FUSN00419410 Updates from SOI 2/3	2014-08-26	C. N. Goodsmith
-600/D	CRs implemented for this revision: FUSN00403168 : Update section 7.10 FUSN00428650 : Update for IOCF process FUSN00406393: Update to section 4.3 to add H/W platform definition ECO-0500159	2014-11-7	H. I. Abu-Dagga
-600/E	CRs implemented for this revision: FUSN00465526: Process Updates to SDP for Corresponding Checklist Updates FUSN00465612: Update section 7.3.2.2.2 Deficiency Reason Codes FUSN00464660: Update section 7.3.2.2.1 for Acceptable Reason Codes FUSN00457457: Incorporate comments received from SOI 1 Review	2015-05-05	H. I. Abu-Dagga
-600/F	CRs implemented for this revision: FUSN00522317: SCA Reason Codes - Add back DEF-Implementation FUSN508177: Code Inspection Checklist ECO-0573689	2016-09-07	W. Hu

VER/ REV	DESCRIPTION	DATE	APPROVED
-600/G	CRs implemented in this revision: - FUSN00539188: PROCESS: Update SDP to document use of JIRA along with ClearQuest - FUSN00467407: PROCESS: The SDP 829-6997-600 section 6.3.1 not including the CSUs as part of the input. ECO-0585861	2017-02-07	T. Countryman
-600/H	DLSS-6843 Update Data Link SDP to Address Deviation DLSS-1568 Update SCA Reason code DLSS-4113 Update to remove reference to Industry Documents CO-00094783	2020-07-11	H. Abu-Dagga
-600/J	DLSS-17212: DLCA6500: Missing process to generate the TDF/BDF in the SDP DLSS-17388: Data and Control Coupling is not explained clearly in SDP DLSS-17213: Link2K : Update PSAC & SDP to address SOI-4 findings identified in BL20 (PKG #42129) CO-00147844	2022-05-04	M. Musthafa

Table of Contents

1 SCOPE **10**
1.1 PURPOSE 10
1.2 APPLICABILITY 10
2 REFERENCE DOCUMENTS **11**
2.1 INTERNAL 11
2.1.1 Policies and Procedures 11
2.1.2 Project Specific Documents 11
2.2 EXTERNAL 12
3 LIFE CYCLE OVERVIEW **13**
3.1 LIFE CYCLE MODEL 13
3.2 LIFE CYCLE PROCESS FLOW 15
3.3 LIFE CYCLE PROCESS ACTIVITY DESCRIPTIONS 16
4 DATA LINK DEVELOPMENT ENVIRONMENT **18**
4.1 ORGANIZATION 18
4.1.1 Staffing 18
4.1.2 Roles and Responsibilities 18
4.2 STANDARDS 20
4.2.1 Requirements Standards 20
4.2.1.1 Methods for Developing Software Requirements 20
4.2.1.2 Expressing Software Requirements 20
4.2.1.3 Derived Requirements 21
4.2.1.4 Process and Tools 21
4.2.1.4.1 DOORS Standard Attributes 21
4.2.1.4.2 DOORS Document Specific Attributes 21
4.2.1.5 Safety Related Requirements 22
4.2.2 Design Standards 22
4.2.2.1 Software Architecture 22
4.2.2.2 Software Detailed Design 23
4.2.2.3 Example Design Description Configuration 24
4.2.3 Coding Standards 25
4.3 DEVELOPMENT TOOLS 25
5 SOFTWARE PLANNING AND MANAGEMENT PROCESSES **30**
5.1 SOFTWARE PLANNING PROCESS 30
5.1.1 Overview 30
5.1.2 Entry Conditions 30
5.1.3 Activity Tasks 30
5.1.4 Exit Conditions 32
5.2 SOFTWARE PROJECT MANAGEMENT ACTIVITIES 32
5.2.1 Track Project Action Items 32
5.2.2 Metrics 33
5.2.2.1 Review Metrics 33
5.2.2.2 Code Metrics 33
5.2.2.3 Integration Metrics 33
5.2.3 Track Cost and Schedule Performance 33
5.2.4 Status Meetings 33

5.2.5 Conduct Project Reviews 34
5.2.6 Maintenance of the Software Lifecycle Planning Documents 34
5.2.7 Risk Management 34
5.2.8 Subcontractor Management 34
5.2.8.1 Airtel ATN Subcontractor 35
5.2.8.2 Other Subcontractors 35

6 SOFTWARE DEVELOPMENT PROCESSES 36

6.1 SOFTWARE REQUIREMENTS PROCESS 36
6.1.1 Overview 36
6.1.2 Entry Conditions 36
6.1.3 Activity Tasks 36
6.1.4 Exit Conditions 38
6.2 SOFTWARE DESIGN PROCESS 38
6.2.1 Overview 38
6.2.2 Entry Conditions 38
6.2.3 Activity Tasks 39
6.2.4 Exit Conditions 42
6.3 SOFTWARE CODING PROCESS 42
6.3.1 C/C++/Assembly/APL Coding 42
6.3.1.1 Overview 42
6.3.1.2 Entry Conditions 42
6.3.1.3 Activity Tasks 42
6.3.1.4 Exit Conditions 44
6.3.2 VAPS Coding 44
6.3.2.1 Overview 44
6.3.2.2 Entry Conditions 44
6.3.2.3 Activity Tasks 44
6.3.2.4 Exit Conditions 47
6.4 SOFTWARE INTEGRATION PROCESS 47
6.4.1 Overview 47
6.4.2 Entry Conditions 47
6.4.3 Activity Tasks 47
6.4.4 Exit Conditions 49

7 SOFTWARE VERIFICATION PLAN 50

7.1 ORGANIZATION 50
7.2 INDEPENDENCE 50
7.3 VERIFICATION METHODS 50
7.3.1 Review Method 50
7.3.1.1 PREP Database Locations 50
7.3.1.2 Peer Review Checklists 51
7.3.2 Analysis Methods 51
7.3.2.1 Requirements Based Test Coverage Analysis Method 51
7.3.2.2 Structural Coverage Analysis Method 51
7.3.2.2.1 Acceptable Reason Codes 52
7.3.2.2.2 Deficiency Reason Codes 53
7.3.2.2.3 Deactivated Code 53
7.3.2.3 Link Analysis Method 54
7.3.2.4 Timing Analysis Method 54
7.3.2.5 Stack Analysis Method 55
7.3.3 Test Methods 55
7.4 VERIFICATION PROCESSES 55

7.4.1 Verify Software Requirements 55
7.4.1.1 Overview 55
7.4.1.2 Entry Conditions 56
7.4.1.3 Activity Tasks 56
7.4.1.4 Exit Conditions 56
7.4.2 Verify Software Design 56
7.4.2.1 Overview 56
7.4.2.2 Entry Conditions 57
7.4.2.3 Activity Tasks 57
7.4.2.4 Exit Conditions 58
7.4.3 Verify Software Source Code 58
7.4.3.1 Verify Source Code implemented in C/C++/Assembly/APL 58
7.4.3.1.1 Overview 58
7.4.3.1.2 Entry Conditions 59
7.4.3.1.3 Activity Tasks 59
7.4.3.1.4 Exit Conditions 59
7.4.3.2 Verify Source Code implemented in VAPS 59
7.4.3.2.1 Overview 59
7.4.3.2.2 Entry Conditions 60
7.4.3.2.3 Activity Tasks 60
7.4.3.2.4 Exit Conditions 60
7.4.4 Verify Software Integration 60
7.4.4.1 Overview 60
7.4.4.2 Entry Conditions 61
7.4.4.3 Activity Tasks 61
7.4.4.4 Exit Conditions 61
7.4.5 Develop Software Test Cases 61
7.4.5.1 Overview 61
7.4.5.2 Entry Conditions 62
7.4.5.3 Activity Tasks 62
7.4.5.4 Exit Conditions 65
7.4.6 Verify Software Test Cases 65
7.4.6.1 Overview 65
7.4.6.2 Entry Conditions 65
7.4.6.3 Activity Tasks 65
7.4.6.4 Exit Conditions 66
7.4.7 Develop Software Test Procedures 66
7.4.7.1 Overview 66
7.4.7.2 Entry Conditions 66
7.4.7.3 Activity Tasks 66
7.4.7.4 Exit Conditions 67
7.4.8 Verify Software Test Procedures 67
7.4.8.1 Overview 67
7.4.8.2 Entry Conditions 68
7.4.8.3 Activity Tasks 68
7.4.8.4 Exit Conditions 68
7.4.9 Perform Software Verification Testing 69
7.4.9.1 Verification of Requirements implemented using C/C++/Assembly/APL coding 69
7.4.9.1.1 Overview 69
7.4.9.1.2 Entry Conditions 69
7.4.9.1.3 Activity Tasks 69
7.4.9.1.4 Exit Criteria 71
7.4.9.2 Verification of Requirements implemented using VAPS 71

7.4.9.2.1 Overview ... 71
7.4.9.2.2 Entry Conditions ... 71
7.4.9.2.3 Activity Tasks ... 71
7.4.9.2.4 Exit Conditions ... 71
7.4.10 Verify Test Results ... 71
7.4.10.1 Overview ... 71
7.4.10.2 Entry Conditions ... 72
7.4.10.3 Activity Tasks ... 72
7.4.10.4 Exit Conditions ... 75
7.5 VERIFICATION ENVIRONMENT ... 75
7.5.1 Tool Qualification ... 75
7.6 TRANSITION CRITERIA ... 75
7.7 PARTITIONING ... 75
7.8 COMPILER ASSUMPTION ... 76
7.9 RE-VERIFICATION ... 76
7.10 PREVIOUSLY DEVELOPED SOFTWARE ... 76
7.11 MULTIPLE-VERSION DISSIMILAR SOFTWARE ... 76

8 SOFTWARE CONFIGURATION MANAGEMENT ... 77

8.1 RESPONSIBILITIES ... 77
8.2 ENVIRONMENT ... 77
8.3 ACTIVITIES ... 78
8.3.1 Configuration Identification ... 78
8.3.2 Baselines and Traceabiity ... 78
8.3.3 Change Requests/Problem Reporting ... 79
8.3.4 Change Control ... 79
8.3.5 Change Review ... 80
8.3.6 Configuration Status Accounting ... 80
8.3.7 Archive, Retrieval, and Release ... 80
8.3.8 Software Load Control ... 80
8.3.9 Software Life Cycle Environment Controls ... 80
8.3.10 Software Life Cycle Data Controls ... 80
8.4 TRANSITION CRITERIA ... 81
8.5 SOFTWARE CONFIGURATION MANAGEMENT DATA ... 81
8.6 SUPPLIER CONTROL ... 81

9 SOFTWARE QUALITY ASSURANCE ... 82

10 SOFTWARE DEPLOYMENT ... 83

10.1 SOFTWARE ACCOMPLISHMENT SUMMARY ... 83
10.2 VERSION DESIGN DESCRIPTION ... 83
10.3 FOOTPRINT DOCUMENT ... 83
10.4 EMOD DOCUMENT ... 83

APPENDIX A DATA LINK SOFTWARE PRODUCTS ... 85

APPENDIX B CORRELATION TO DO-178B ... 86

APPENDIX C GLOSSARY ... 87

APPENDIX D ACRONYMS AND ABBREVIATIONS ... 89

List of Figures

FIGURE 3-1 LIFE CYCLE MODEL 13
FIGURE 3-2 SOFTWARE LIFE CYCLE PROCESS FLOW 16
FIGURE 4-1 EXAMPLE DESIGN DESCRIPTION CONFIGURATION 24
FIGURE 6-1 VAPS/TDF/BDF GENERATION PROCESS 45

List of Tables

TABLE 4-1 SOFTWARE DEVELOPMENT TOOLS 25
TABLE 4-2 SOFTWARE VERIFICATION ENVIRONMENT FOR LRU PRODUCTS 27
TABLE 4-3 SOFTWARE VERIFICATION ENVIRONMENT FOR IMA PRODUCTS 28
TABLE 7-1 BOUNDARY VALUE TEST CONDITIONS 63
TABLE 10-1 DATA LINK SOFTWARE PRODUCTS 85

1 Scope

1.1 Purpose

The purpose of this document is to describe the software life-cycle processes, standards, methods, and environment that will be used to develop and verify software for the Commercial Systems Data Link products. This Software Development Plan (SDP) was developed in accordance with the requirements in the *Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B* [17], and using general guidelines from the *Rockwell Collins Technical Consistent Process* (RC-TCP) [1].

1.2 Applicability

This SDP defines the software development and verification processes to be used for select Data Link software products that are managed within the Commercial Systems Data Link organization.

Since many of the Data Link functions are deployed on a wide range of target equipment types supporting Air Transport, Business and Regional, and Government Systems programs, the Data Link software products are developed and managed as individual product-line entities, and then integrated into the various target platforms.

Appendix A provides a listing of Data Link software products, along with the associated target equipment types, for which this SDP is intended. Refer to the PSAC associated with a particular Data Link software product for the exact SDP applicable to that product.

In general, this SDP defines the processes to be followed for developing software that is commensurate with DO-178B Level C. Therefore, when applied to software components that are Level C, all of the processes defined herein are applicable. When applied to software components that are Level D, all of the processes defined herein are applicable except those that are clearly stated as required for Level C software only. Processes defined herein are not required for Level E components.

2 Reference Documents

2.1 Internal

2.1.1 Policies and Procedures

Note: Latest revision assumed unless specifically stated.

[1] Rockwell Collins Technical Consistent Process Version 3.2, RCPN 832-8716-009

[2] DOORS Documentation Method for the Commercial Systems Data Link Organization, RCPN 945-9527-001

[3] Peer Review Method Using PREP for the Commercial Systems Data Link Organization, RCPN 945-9104-001

[4] Change Request and CCB Method Using ClearQuest for the Commercial Systems Data Link Organization, RCPN 945-9035-001

[5] Change Request and CCB Process for the Commercial Systems Data Link Organization Using JIRA , RCPN 946-8189-002

[6] Software Configuration Management Plan, RCPN 832-2963-001

[7] Design Quality Assurance Plan for Hardware, Software and System Development, RCPN 946-5892-100

[8] Software Control Library Release Procedure, RC-CMS-P-003

[9] Integrated Modular Avionics (IMA) Footprint Process Document, RCPN 964-4841-001

[10] Configuration Management Plan for Developmental Avionics Equipment (Emod Procedure), RCPN 829-8407-303

[11] Pro Line Fusion Input Output Common Format Interface Definition Document (IOCF IDD) Process, BRS-ENG-P-006

[12] GUI Conventions Description (GCD) for the ProLine Fusion Product Line Integrated Avionics System, RCPN 964-5034-A01

2.1.2 Project Specific Documents

[13] Coding Standards for the C++ Language, RCPN 832-0536-005

[14] Collins Commercial Avionics Comm and Radio Navigation Dept PL/M and Assembly Software Design and Coding Standards, RCPN 826-9732-001 Rev A

[15] User Manual for the CNS/ATM CMU-900 Application Programming Language (APL), RCPN 829-2747-200

[16] Data Link Products Peer Review Checklists, RCPN 963-9782-100

2.2 External

[17] Software Considerations in Airborne Systems and Equipment Certification, Document No. RTCA DO-178B, December 1992, RTCA, Washington, D.C.

3 Life Cycle Overview

3.1 Life Cycle Model

The Data Link software life cycle processes are based on an iterative spiral model as shown below in Figure 3-1 Life Cycle Model. For a given project, these life cycle processes may be iterated upon as dictated by project complexity, incremental development of system functions, availability of target hardware, or planned software build progressions.

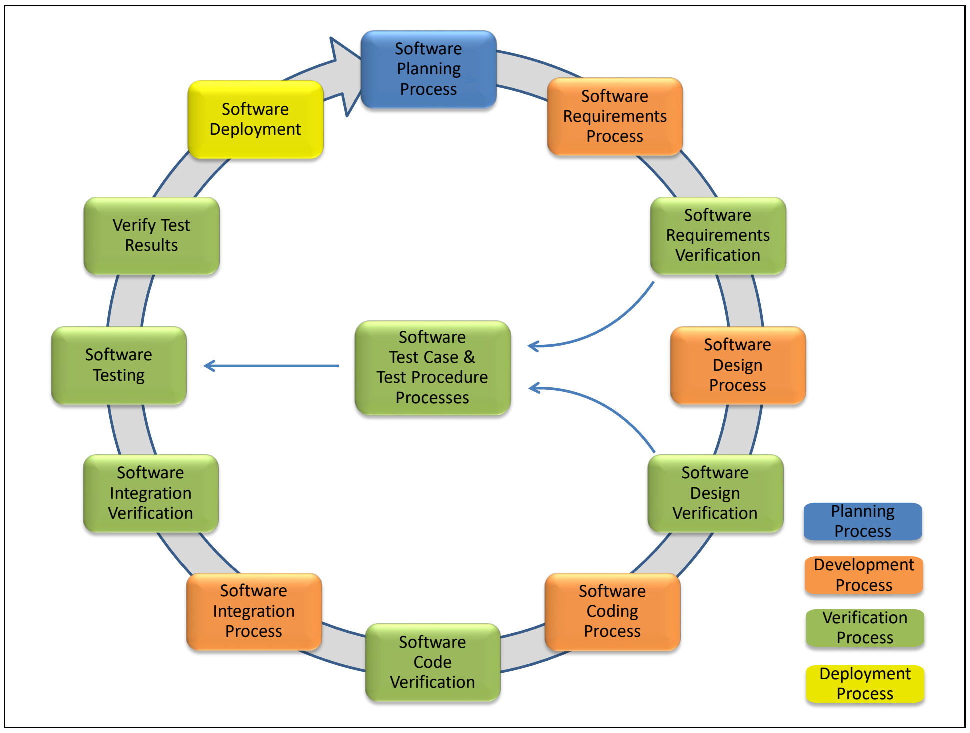

Software Planning Process
Software Requirements Process
Software Requirements Verification
Software Design Process
Software Design Verification
Software Coding Process
Software Code Verification
Software Integration Process
Software Integration Verification
Software Testing
Verify Test Results
Software Deployment
Software Test Case & Test Procedure Processes
Planning Process
Development Process
Verification Process
Deployment Process

Figure 3-1 Life Cycle Model

The following paragraphs provide a brief description of the process activities identified in Figure 3-1 Life Cycle Model above.

Software Planning Processes

Software Planning Process: The software planning process is for the development, agreement, and documentation of the processes, methods, and tools that are planned to be utilized for the successful development of the end software product. The software planning process includes development of the Plan for Software Aspects of Certification (PSAC), Software Development Plan (SDP), Software Verification Plan, Software Configuration Management Plan, and the Software Quality Assurance Plan. Section 5.1 describes the detailed activities associated with this process.

Software Development Processes

Software Requirements Process: The software requirements process generates the software high-level requirements as described in DO-178B section 5.1. The software high-level requirements consist of requirements that have been allocated to software from system requirements, along with derived high-level requirements, in support of a planned software deployment. For Data Link products, the software high-level requirements are defined to be Level 3 (L3) requirements. Reference the Glossary for a definition of all the requirement levels. The software high-level requirements are captured in the project's Level 3 SRS(s). Section 6.1 describes the detailed activities associated with this process.

Software Design Process: The software design process generates the software architecture, detailed design, and low-level requirements as described in DO-178B section 5.2. For Data Link products, the software low-level requirements are defined to be Level 4 (L4) requirements. Reference the Glossary for a definition of all the requirement levels. As the software architecture and design matures, additional derived low-level requirements may be generated to reflect the design choices. The software architecture and detailed design are captured in the project's SDD, the input/output definition and data flows are captured in the IOCF, and the software low-level requirements are captured in the project's L4 SRS(s). Section 6.2 describes the detailed activities associated with this process.

Software Coding Process: The software coding process generates the software source code as described in DO-178B section 5.3. The software source code is generated from the software architecture, detailed design, and low-level requirements in a manner which satisfies the software high-level requirements. Section 6.3 describes the detailed activities associated with this process.

Software Integration Process: The software integration process generates executable object code for the target computer as described in DO-178B section 5.4. This activity includes both software-software integration as well as software-hardware integration. The executable code is produced from compiled and linked source code in the form of a labeled build per the project's build plan. Section 6.4 describes the detailed activities associated with this process.

Software Verification Processes

Software Requirements Verification: This process utilizes reviews to verify the software high-level requirements as described in DO-178B section 6.3.1. Section 7.4.1 describes the detailed activities associated with this process.

Software Design Verification: This process utilizes reviews to verify the software architecture, detailed design, IOCF, and low-level requirements as described in DO-178B sections 6.3.3 and 6.3.2, respectively. Section 7.4.2 describes the detailed activities associated with this process.

Software Code Verification: This process utilizes reviews to verify the software source code as described in DO-178B section 6.3.4. Section 7.4.3 describes the detailed activities associated with this process.

Software Integration Verification: This process utilizes reviews to verify the software integration outputs as described in DO-178B section 6.3.5. Section 7.4.4 describes the detailed activities associated with this process.

Software Test Case & Test Procedure Processes: These processes generate the software test cases and test procedures from the software high-level requirements and software low-level requirements as described in DO-178B section 6.4. Test cases and test procedures will be generated to accommodate hardware/software integration testing and software integration testing in a manner which satisfies high-level requirements-based coverage, low-level requirements-based coverage, and, for Level C software, structural coverage. Note that test cases and test procedures can be developed for test execution on either the host or target environment. Sections 7.4.5 and 7.4.7 describe the detailed activities associated with these processes.

Software Test Cases & Procedures Verification: This process utilizes reviews to verify the software test cases and test procedures that are produced to achieve high-level requirements and low-level requirements based testing of the software product, as described in DO-178B section 6.3.6. Sections 7.4.6 and 7.4.8 describe the detailed activities associated with these processes.

Software Testing: This process describes the hardware/software integration testing and the software integration testing based on the verified test cases and test procedures as described in DO-178B section 6.4. Note that some test execution may occur in a host environment to facilitate software integration testing and, for Level C software, to obtain structural coverage results. Where tests are not constrained by a host-based test harness, they will also be run in the target environment to ensure no errors exist in that environment. Section 7.4.9 describes the detailed activities associated with this process.

Verify Test Results: This process utilizes a combination of reviews and analyses to verify the software test results are correct, and that unexpected results, if any, are explained as described in DO-178B section 6.3.6. Section 7.4.10 describes the detailed activities associated with this process.

Software Deployment Process

Software Deployment: This process refers to the formal delivery of the software product outside the Data Link organization. Depending on the cycle, the delivery may encompass an interim labeled build to a System Integration Lab (e.g. Blue Label builds), or it may encompass a completed build intended for on-aircraft testing (e.g. Red Label builds), or it may encompass the final release of a project's planned development (e.g. Black Label builds). For the latter, this process will also include the development and formal release of the final Software Verification Results document, the Software Accomplishment Summary (SAS), and formal release of all related software artifacts into the Rockwell Collins Production Configuration Management System. Section 10 describes the typical artifacts generated to support deployment of the software product. Additional activities in the software deployment process may include support for transitioning the software products into production and eventually into aircraft service (which may include support for customer acceptance, installation, and operation), as well as required support for certification and regulation authorities.

3.2 Life Cycle Process Flow

Figure 3-2 Software Life Cycle Process Flow below depicts the software development and software verification views of the life cycle process flow in relation to their respective life cycle artifacts that are developed for a planned software deployment. The flows indicate dependencies between the processes and artifacts, such that it can be seen which artifacts are used as inputs to a given process and which artifacts are produced by a given process.

The number at the top left in a process icon is a reference to the section number in this document that further defines that process. The numbers at the bottom in an icon provide a reference to the table and objectives in Annex A of DO-178B to map which specific objectives are being met by that process or artifact icon.

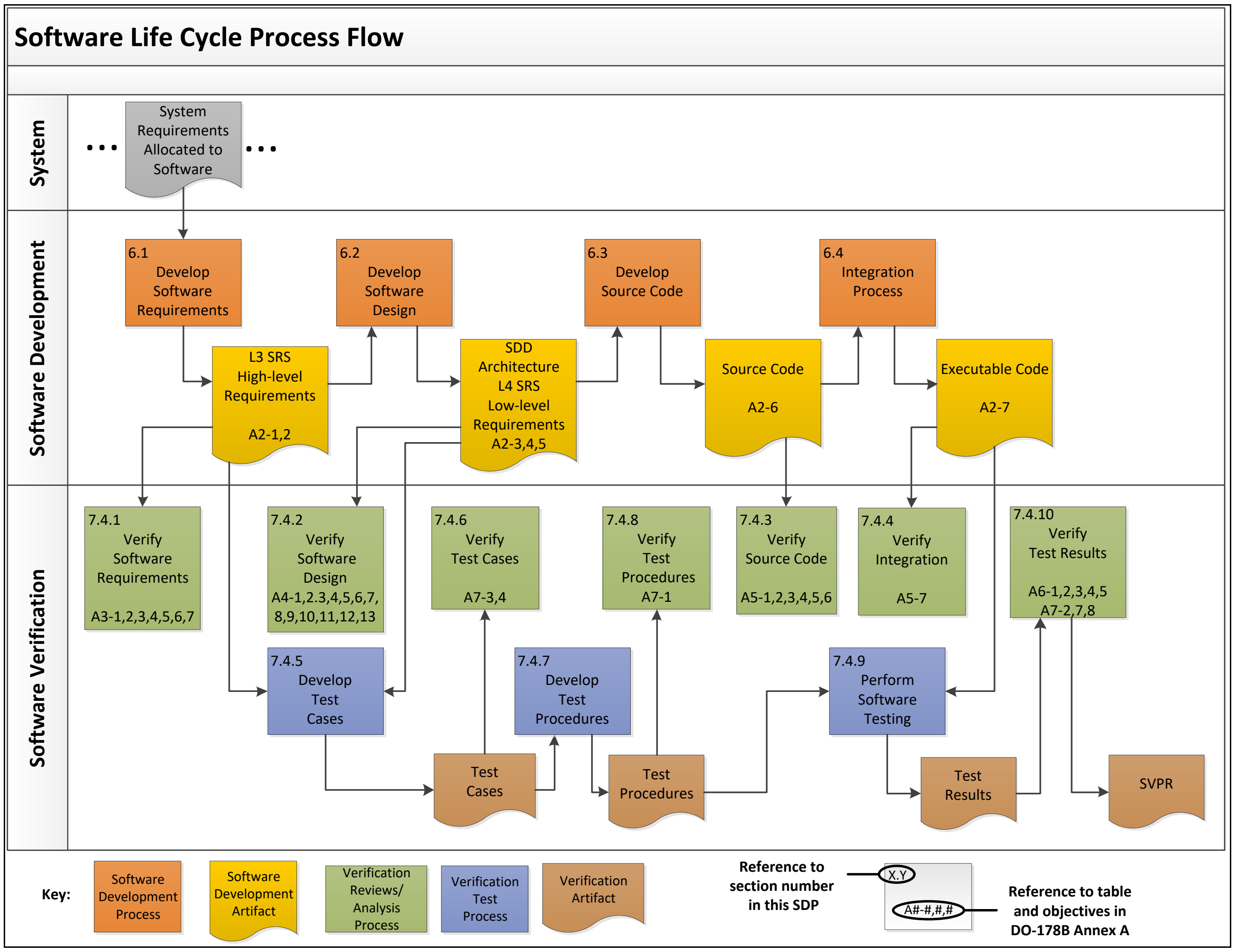

Software Life Cycle Process Flow
System
Software Development
Software Verification
System Requirements Allocated to Software
6.1 Develop Software Requirements
6.2 Develop Software Design
6.3 Develop Source Code
6.4 Integration Process
L3 SRS High-level Requirements A2-1,2
SDD Architecture L4 SRS Low-level Requirements A2-3,4,5
Source Code A2-6
Executable Code A2-7
7.4.1 Verify Software Requirements A3-1,2,3,4,5,6,7
7.4.2 Verify Software Design A4-1,2.3,4,5,6,7, 8,9,10,11,12,13
7.4.6 Verify Test Cases A7-3,4
7.4.8 Verify Test Procedures A7-1
7.4.3 Verify Source Code A5-1,2,3,4,5,6
7.4.4 Verify Integration A5-7
7.4.10 Verify Test Results A6-1,2,3,4,5 A7-2,7,8
7.4.5 Develop Test Cases
7.4.7 Develop Test Procedures
7.4.9 Perform Software Testing
Test Cases
Test Procedures
Test Results
SVPR
Key:
Software Development Process
Software Development Artifact
Verification Reviews/ Analysis Process
Verification Test Process
Verification Artifact
Reference to section number in this SDP
X.Y
A#-#,#,#
Reference to table and objectives in DO-178B Annex A

Figure 3-2 Software Life Cycle Process Flow

3.3 Life Cycle Process Activity Descriptions

The processes of a software life cycle can be characterized by a set of activities where each activity is comprised of five basic components:

- **Entry Conditions** – Defines the transition criteria for entering the activity and must be met prior to starting that activity.
- **Inputs** – An activity accepts one or more inputs that can be received from other activities within the software process or from sources external to the software process. Input(s) will be listed to help identify the items that should be available prior to commencing the activity. Reference the Glossary for the term “reviewed and approved” when applied to input items.
- **Tasks** – An activity utilizes one or more tasks to transform the input(s) into output(s). The task(s) of an activity define the scope of effort for the activity and are auditable requirements for performing the activity. That is, all of the tasks as defined are required to be performed and evidence of their performance must be made available upon request.
- **Outputs** – An activity produces one or more outputs. Output(s) will be listed to help identify the items that are expected to be produced as a result of the activity.

- **Exit Conditions** – Defines the transition criteria for exiting the activity. The condition(s) listed identify the minimum that is necessary to complete the activity.

The life cycle processes and their associated activities will be documented throughout this SDP using the five components described above.

4 Data Link Development Environment

This section provides a description of the Data Link organization, standards, and software development tools that constitute the development environment in which the software life cycle artifacts are generated as described throughout this SDP.

4.1 Organization

4.1.1 Staffing

To optimize efficiency, the Data Link team as a whole is comprised of a multi-disciplinary integrated product development team, composed of systems, software, and verification engineers. Over time, the size and composition of the teams will change to meet the needs of the various Data Link projects. Team members will be expected to be flexible and fill multiple roles as required. A high level of cross training within the team will be used to achieve this flexibility. Note that independence for reviews and testing will be an important factor when team member roles are adjusted within the team.

Each team will accept full responsibility for all aspects of the product being developed by the team, including certification artifacts, and will have broad authority to manage that development. Teams will be encouraged to proactively identify potential issues before they arise and manage them independently. Issues that impact other teams will be raised to either the software leadership team or project leadership team as necessary.

As need dictates, the Data Link team may utilize contract engineers to meet the project staffing needs. In such cases, these contract engineers will be full team members with access to the same tools, files, documentation, and information necessary to accomplish their assigned tasks.

4.1.2 Roles and Responsibilities

The Data Link team has primary responsibility for meeting all the objectives of the software planning, development, verification, and development configuration management processes. The team is also responsible for providing support during the Software Quality Assurance (SQA) process activities.

The Rockwell Collins Design Assurance Center (DAC) for Quality is responsible for the Software Quality Assurance process and is independent from the Data Link team. DAC representatives are also responsible for monitoring the activities of the Data Link team during their life cycle process activities. Refer to the *Commercial Systems Software Quality Engineering Assurance Plan* [7].

The following roles and responsibilities are provided to further define the Data Link team.

Life Cycle Value Stream Manager (LCVSM) – The LCVSM has the overall life cycle responsibility for a given Data Link product with respect to the product's cost and schedule in all aspects including development, manufacture, and deployment into service. The LCVSM is responsible to the customer, senior management, and the program team for overall cost, schedule, and quality performance of the Data Link products.

Technical Product Manager (TPM) – The TPM has the overall responsibility for the technical performance of a given Data Link product. The TPM functions as the technical advisor on Data Link products by providing technical direction as required and overall responsibility for development activities.

Group Manager / Engineering Manager – The Group Manager / Engineering Manager is responsible for providing resources, establishing priorities, resolving conflicts, and providing career development guidance. The Group Manager / Engineering Manager works closely with Project Engineers to ensure that projects are progressing according to plan, within budget, and on schedule.

Systems Engineer – The Systems Engineer has the primary responsibility for defining and allocating the system requirements, creating and maintaining the IOCF document, developing and running System Integration tests on the Systems Integration Rig, and is responsible for tracking cost, schedule, risk and status for the system engineering functions.

Integration and Test Engineer – The Integration and Test Engineer has responsibility for the system integration and performance testing. This is a System Engineering function.

Hardware Engineer – The Hardware Engineer defines the hardware requirements, hardware design, and hardware specification for the various hardware components being designed in the system. Depending on the project size and complexity related to hardware design, there may be a lead hardware engineer for each major component of the system.

Project Engineer (PE) – The PE is responsible for software project detailed planning, including development of the project schedule and work packages that will formulate the project's baseline for measuring earned value. Throughout the development cycle, the PE tracks earned value against the plan and adjusts resources as necessary to maintain scope, schedule, and cost of all work packages. Depending on the project size and complexity, there may be individual domain PE's (e.g. System, Hardware, Software) for a single project.

Software Engineer – The Software Engineer defines the software high-level requirements, software architecture, detailed design, low-level requirements, and source code. The software engineer performs informal unit testing of code units, integrates the software within a software build for the target hardware, and either assists or leads in the investigation and resolution of problem reports. The software engineer may be called upon to assist the verification engineer to help identify test methods, and leads the SCA process by generating instrumented software builds, performs uncovered code analysis, and generates summary coverage reports.

Design Assurance Center (DAC) Representative – The DAC is comprised of DAC Engineers, Subject-Matter-Experts (SMEs), DAC Auditors, and Technical Leads. The primary responsibility of the DAC representatives is to assure that the project is aware of, and compliant with, the quality standards of the operating organizations and the customer. The DAC is independent of the project team and does not participate in the actual software development. The DAC's primary means of gaining assurance is by auditing the process to verify the output artifacts of the various development phases.

Engineering Project Assistant (EPA) – The EPA assists the development team and the PE by facilitating project meetings, peer reviews, and performs general maintenance of engineering data, including administrative control of engineering repositories and process tools, maintaining project management artifacts and generating reports related to scheduling, budgeting, staffing, risk management tracking, etc.

Software Architect – The software architect provides the framework and foundation for software engineering to design and develop the various Data Link software products. The software architect leads or participates in activities such as trade studies, white papers, market trends, competitive landscape, customer proposals, and product roadmaps to steer the development of Data Link software products.

Verification Engineer – The verification engineer performs the verification activities including the development of high-level and low-level requirements-based test cases and test procedures, dry run test execution, formal test execution, and development of the formal Software Verification Procedures and Results (SVPR) document.

NOTE: A member of the project team may fulfill more than one role on the project.

4.2 Standards

This section defines the requirements standards, design standards, and coding standards to be used for the development processes described in this SDP.

4.2.1 Requirements Standards

This section defines the standards to be used when developing or modifying software high-level requirements and software low-level requirements. Although there is a different level of decomposition between software high-level requirements and software low-level requirements, the intent of conforming to the same requirements standard is to facilitate a consistent representation of requirements data to the verification process, in terms of generating test cases and procedures, and demonstrating traceability from test cases and procedures back to the requirements. Throughout this section, where the term “requirement” or “software requirement” is not specifically identified as a high-level or low-level requirement, it will apply to both a high-level requirement and a low-level requirement.

4.2.1.1 Methods for Developing Software Requirements

The Data Link software high-level requirements (L3) are primarily generated as a result of analyses and decomposition of the higher level system requirements allocated to software. During this process, additional derived software high-level requirements may also be generated.

The Data Link software low-level requirements (L4) are primarily generated from the design process as a result of further decomposing the software high-level requirements together with detailed design data resulting from the software architecture. During this process, additional derived software low-level requirements may also be generated.

4.2.1.2 Expressing Software Requirements

Software requirements will be documented in one or more requirements modules.

Where a requirement is expressed in a text object, that text object will contain only one “shall”.

Software requirements will be stated such that the requirements are verifiable. A statement in quantitative terms with acceptable tolerance levels is highly preferred.

A requirement is written with verification in mind so that the text of the requirement is clear and unambiguous in order to define the verification process.

All cross-references within the software requirements documentation should be accurate and consistently formatted.

The specified software requirements should be:

1. Accurate, unambiguous, and sufficiently detailed.
2. Compatible with the target computer (no conflicts should exist between the software requirements and the hardware features of the target computer, such as I/O, storage, and performance capabilities).
3. Complete (traceability to higher level requirements helps to identify gaps in the higher level allocation).

4.2.1.3 Derived Requirements

A derived requirement is a requirement that is introduced at a given level that is not traceable to a higher-level requirement. Derived requirements will be explicitly identified and justified in the software requirements module. Derived software high-level requirements should be identified by analyzing the subsystem design for any design decisions or other design constraints imposed on the software solution that was not otherwise stated as a specific subsystem requirement. Likewise, derived software low-level requirements should be identified by analyzing the software architecture and detailed design for any software design decisions or other design constraints imposed on the software solution that was not otherwise stated as a specific software requirement.

Whenever a derived requirement is introduced, consideration should be given as to whether a higher-level requirement should be introduced and then traced to it. While the goal is to maintain appropriate levels of decomposition, often times the introduction of a derived requirement at one level may signal requirement deficiencies at the higher level.

All software requirements that are derived will be made available to the systems safety team to assess potential impact to safety. This is accomplished via the peer review method where attendance from the safety team is required. If there will be numerous iterative peer reviews on a given requirements document, it may be more practical to hold a separate review for the safety team when the document is considered complete.

4.2.1.4 Process and Tools

Software requirements will be captured, maintained, and managed using the DOORS requirements tool as listed in Table 4-1.

Project specific DOORS modules will be created and maintained in accordance with the *DOORS Documentation Method for the Commercial Systems Data Link Organization [2].*

4.2.1.4.1 DOORS Standard Attributes

In addition to the rules defined for the Standard Attributes as described in [2], the following rules will be applied to all Data Link DOORS requirements modules:

- If _*Derived* = No, there must be an outgoing link to a higher level requirement.
- If _*Derived* = Yes, there must not be an outgoing link to a higher level requirement and the attribute _*Assumptions/Rationale* is required to be completed.
- Any requirement that is allocated from a higher level requirement which has the attribute _*Safety* = Yes, must also have its _*Safety* attribute set to Yes. See section 4.2.1.5 for more information on the _*Safety* attribute.
- For software high-level requirements, the attribute _*Verification* will be set to “Lower”, indicating that the corresponding software low-level requirements will identify the specific verification method(s).
- For software low-level requirements, the attribute _*Verification* will be set to the specific verification method as described in section 7.4.5.3.

4.2.1.4.2 DOORS Document Specific Attributes

As described in [2], each project may elect to define document specific attributes (denoted by a ~). Where document specific attributes are used for a given project, their usage within a module will be described in the module itself.

4.2.1.5 Safety Related Requirements

Safety requirements should never originate at the software level. Safety requirements should always originate from higher level system requirements and be flowed down to the software level. To maintain a high level of awareness on safety related requirements flowed down to the software level, and to facilitate analyses on the impact of changing a safety related requirement, the following process steps will be followed when developing the software requirements:

- All safety related requirements in the software requirements modules will be specifically identified with the _*Safety* attribute set to “Yes”.
- If a system requirement that is identified with the _*Safety* attribute is allocated to a software high-level requirement, then the software high-level requirement will also be identified with the _*Safety* attribute.
- If a software high-level requirement that is identified with the _*Safety* attribute is allocated to a software low-level requirement, then the software low-level requirement will also be identified with the _*Safety* attribute.
- If a software requirement (high-level or low-level) is identified with the _*Safety* attribute, then the higher level requirement that it traces to will also be identified with the _*Safety* attribute.
- When peer reviewing derived requirements in an SRS, the safety team will be invited to the peer review to perform an assessment on potential impact to safety. If the safety team determines a derived requirement has potential impact to safety, then a higher level safety requirement will be created and the derived requirement will trace to the higher level requirement, effectively rendering it no longer derived.

4.2.2 Design Standards

This section defines the standards to be used when developing or modifying the software architecture and detailed design. While the software design process generates the design description, which is the software architecture and low-level requirements, this standard applies only to the software architecture and detailed design data. The software low-level requirements will be documented as described by the Requirements Standards in section 4.2.1.

There are no specific naming conventions required for the design data. The complexity restrictions and design constraints associated with Object Oriented Design (OOD) and Object Oriented Programming (OOP) are defined in the *Software Coding Standards for the C++ Language*[13].

4.2.2.1 Software Architecture

The software architecture, also referred to as software high-level design, begins with a context diagram of the software solution domain and, through functional decomposition, identifies the major software components and interfaces within that domain. In this context, the software components are high-level functional entities that will be implemented by one or more software units to perform a specified function in the overall software solution.

There is no restriction on the tools and/or specific output artifact that captures the software architecture. Typical Data Link projects use Microsoft Word to generate the software architecture description in an SDD, supplemented with Microsoft Powerpoint and/or Microsoft Visio to generate high-level figures, diagrams, and illustrations within the SDD. Alternatively, other tools, such as UML modeling tools, can be used. Regardless of tools or output format, the following content should be captured as part of the software architecture description:

- A description of the software architecture and high level components
- A description of the input/output at the high-level interfaces
- Data flow and control flow
- Resource limitations, including timing and memory
- Scheduling procedures and inter-process communication schemes
- A description of the design methods and details for their implementation
- A description of partitioning and the means of preventing breaches
- A description of deactivated code (if applicable) and the means to ensure that the code cannot be enabled in the target computer. In this context, “deactivated code” means code associated with high-level functions that are designed to be included/excluded based on a configuration item (e.g. strap, program pin, license key, etc.).
- A description of the rationale for decisions on software designs, particularly decisions that are related to parent safety requirements. Design decisions are typically captured in a separate section of the SDD and provide a useful means to document critical design decisions. The intent is to document answers to questions such as “Why did they do it that way?”

4.2.2.2 Software Detailed Design

Each of the high-level software components defined in the software architecture is further decomposed to identify the specific software units (coding modules) that will be developed for a particular functional entity. The act of identifying software units is at the very core of performing software detailed design; as class structures, data structures, class hierarchies, class relationships, and function declarations, are developed to provide the software coding framework from which source code can be developed to satisfy the software high-level and low-level requirements.

There is no restriction on the tools and/or specific output artifact that captures the software detailed design. This can be captured as text based design description in the same Microsoft Word document as the software architecture, or it can be captured in a separate UML model if modeling is chosen for the project, or it can be captured in the project source code header files, or it can be captured in any other design tool. When the detailed design description is captured in an external form, it is not necessary to duplicate this same information in the Microsoft Word SDD document; a reference to the external form will be sufficient.

Regardless of tools or output format, the following content should be captured as part of the software detailed design:

- A description of each software unit that is created to satisfy the software high-level and low-level requirements, including algorithms, data structures, class structures, and class relationships.
- Further decomposition of the architecture description within the software unit description as required to facilitate robustness testing

If the primary means to capture the detailed design is not in DOORS (e.g. Microsoft Word and/or modeling tool and/or header files), then it is recommended to create a separate module in DOORS that contains a proxy representation of the design units using individual DOORS objects. In this manner, the design units can be traced to software low-level requirements.

If the primary means to capture the detailed design is accomplished via modeling, the following UML diagrams may be used to capture the detailed design:

- Use case diagrams
- Class diagrams
- Component diagrams
- Sequence diagrams
- Collaboration diagrams
- Modules and their process structure
- State Transition diagrams

4.2.2.3 Example Design Description Configuration

Figure 4-1 below illustrates a sample configuration for capturing the software design description. In this example, the Microsoft Word Document SDD is the primary SDD document, where the software design decisions and software architecture are captured using a combination of text and figures. If modeling is chosen for the project, a reference to the model should be inserted in the Detailed Design section of the Word document (as shown via the optional dashed line). Alternatively, if the detailed design is captured in the project source code header files, a reference to the CPCI (which contains a baselined version of the header files) should be inserted in the Detailed Design section. The software low-level requirements (L4 SRS) are captured in one or more DOORS modules. To facilitate traceability from detailed design to the low-level requirements, a DOORS SDD proxy module is created which contains a representation of the individual design units (e.g. class names) and their corresponding links to applicable low-level requirements. A reference to the DOORS proxy module should be inserted in the Microsoft Word SDD document.

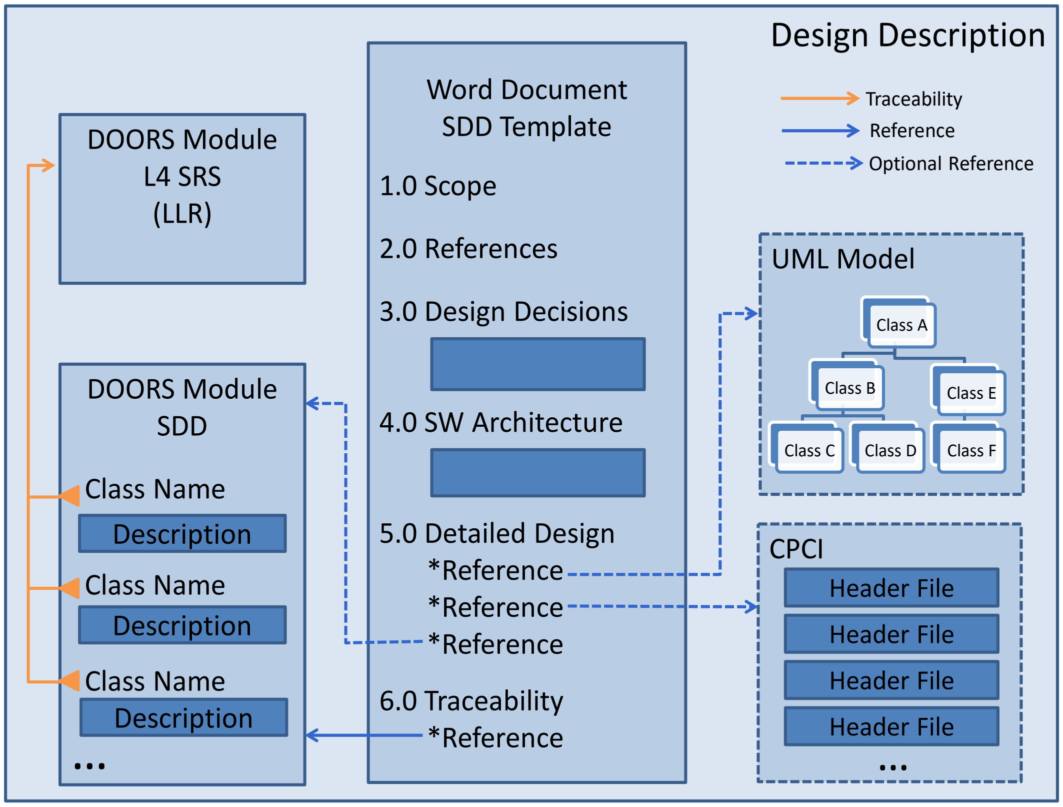


Figure 4-1 Example Design Description Configuration

This example configuration offers the flexibility of using different tools to capture the entire design description in a manner that best suits each tool's intended function:

- Microsoft Word: Best suited for capturing textual descriptions along with associated figures and tables
- UML Model: Best suited for capturing electronic versions of UML models
- Header Files: Best suited for capturing class, function, and data structure definitions
- DOORS: Best suited for capturing software low-level requirements, linking objects, and demonstrating traceability

4.2.3 Coding Standards

The software coding standards for developing C/C++ code are described in the *Software Coding Standards for the C++ Language [13].*

The software coding standards for developing assembly code are described in the *ASM Coding Standards [14].*

The software coding standards for producing application data sets are described in the *User Manual for the CNS/ATM CMU-900 Application Programming Language (APL) [15].*

The standards for VAPS developments are defined in *GUI Conventions Descriptions (GCD)[12].*

4.3 Development Tools

The tables below identify the individual tools that are used for software development and software verification across the Data Link products. Note that some projects may not need every tool listed. In the tables below, project specific Developer Guides and Verification User Guides are used to aid the engineers with installation, configuration, and general user guidance associated with the tools and environment. The development hardware platform used for all the projects is a standard 4th generation Intel based (i.e. Intel Core i3, Intel Core i5, or Intel Core i7) workstation (PC) running Microsoft Windows XP, Windows 7, or Windows 10.

Table 4-1 Software Development Tools

Life Cycle	Tool Capability	Tools Used
Support	Documentation	DOORS, Microsoft Word, Microsoft Visio, Microsoft Powerpoint
	Schedule Management	Microsoft Project, SAP
	Problem Report Tracking	Rational ClearQuest JIRA
	Developmental SW Configuration Management	Rational ClearCase, Subversion
	Peer Review Management	PREP (Peer Review Eclipse Plug-in)

Life Cycle	Tool Capability	Tools Used
High-level Requirements	Traceability	DOORS
	Specification	DOORS
Low-level Requirements	Traceability	DOORS
	Specification	DOORS
Architecture and Detailed Design	Software Design Methodology and Specification	Microsoft Word, Microsoft Visio, DOORS, Doxygen, Enterprise Architect, Understand
Code & Unit Test (LRU-based targets)	IDE	Microsoft Dev Studio
	C/C++ Compiler	Watcom C/C++
	Assembler/Linker	ASM86, PLM86, LINK86, LOC86, OH86
	Linker/Locater	Pharlap
	Application Debugger	Microsoft Dev Studio
	Test Simulator	ABLE, AAR, ATC Ground Station, Data Link Tester, Data Manager Tester
	Source Code Analyzer	Gimpel PC Lint for C/C++
Code & Unit Test (IMA-based targets)	IDE	Eclipse CDT and MinGW
	C/C++ Compiler	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows
	Assembler	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows
	Linker	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows
	Application Debugger	GNU/GDB
	Test Simulator	VISTA
		ATC Ground Station
		AAR, Data Link Tester, Data Manager Tester, AGPS, DLCAgen, Trace Tool
	Source Code Analyzer	Gimpel PC Lint for C/C++

Life Cycle	Tool Capability	Tools Used
Code & Unit Test (VAPS)	VAPS Layers	VAPSXT
	IDE	Eclipse CDT and MinGW
	C/C++ Compiler	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows (for autogenerated code from definition files)
	Linker	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows
	Application Debugger	GNU/GDB
	Test Simulator	VISTA
		ATC Ground Station
		AAR, Data Link Tester, Data Manager Tester, AGPS, DLCAgen, Trace Tool
	Source Code Analyzer	Gimpel PC Lint for C++ ((for autogenerated code from definition files)) Format Viewer for VAPS

Table 4-2 Software Verification Environment for LRU products

Life Cycle	Tool Capability	Tool Name	DO-178B Qualification Required?
Software Verification (Hardware Target Platform)	Operating System	VRTX 32	No
	Test Stand	ABLE Test Stand	No
Software Verification (Software)	Target Compiler	Watcom C/C++	No
	Target Linker	Pharlap	No
	Coverage Analysis	LDRA, VectorCAST	Yes*

Life Cycle	Tool Capability	Tool Name	DO-178B Qualification Required?
	Simulation Environment	ABLE	No
		DataLink Tester	No
		ATC Ground Station	No
		Airtel ATN Router	No
	Software Loading Capability	Arinc 615-3 compatible dataloader	No
Software Verification Procedures & Results	Documentation	DOORS, MS Word	No

* The project PSAC will define if tool(s) requiring qualification have already been qualified for a previous certification.

Table 4-3 Software Verification Environment for IMA products

Life Cycle	Tool Capability	Tool Name	DO-178B Qualification Required?
Software Verification (Hardware Target Platform)	Operating System	LynxOS-178	No
	Test Stand	CTA Test Stand	No
Software Verification (Software)	Target/Host Compiler	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows	No
	Target/Host Linker	LynxOS-178 CDK GNU gcc PPC cross compiler for Windows	No
	Coverage Analysis	VectorCAST	Yes*
	Simulation Environment	VISTA	Yes*
		ABLE	No
		Data Manager Tester	No
		DataLink Tester	No
		ATC Ground Station	No
		AGPS	No

Life Cycle	Tool Capability	Tool Name	DO-178B Qualification Required?
		Airtel ATN Router	No
		DLCAgen	No
		Vision Framework Tool	Yes*
		Message Library Tester	No
		DLCA Trace Tool	No
Software Verification Procedures & Results	Documentation	DOORS, MS Word	No

* The project PSAC will define if tool(s) requiring qualification have already been qualified for a previous certification.

5 Software Planning and Management Processes

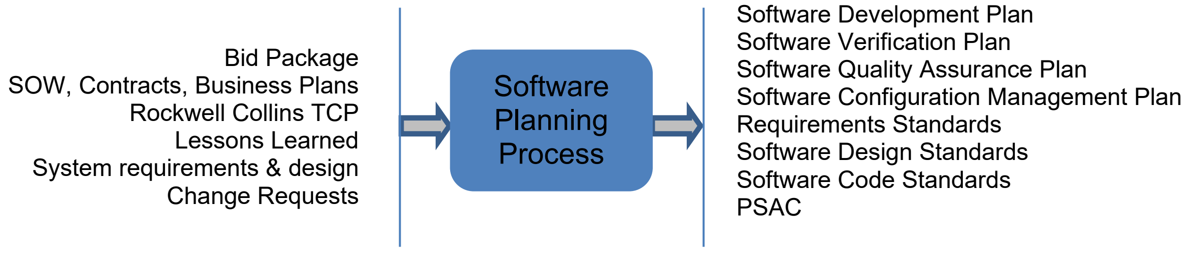
5.1 Software Planning Process

5.1.1 Overview

The objective of this activity is to develop and document the software development processes and corresponding integral processes, in accordance with DO-178B software planning objectives, to support the successful development of Data Link software products. The software planning process includes development of the following:

- Plan for Software Aspects of Certification (PSAC)
- Software Development Plan (SDP)
- Software Verification Plan
- Software Configuration Management Plan
- Software Quality Assurance Plan
- Software Requirements Standards
- Software Design Standards
- Software Code Standards

Other than the PSAC, the Software Code Standards, the Software Quality Assurance Plan, and the Software Configuration Management Plan, the plans listed above are documented in their entirety as part of the SDP (this document).


Bid Package
SOW, Contracts, Business Plans
Rockwell Collins TCP
Lessons Learned
System requirements & design
Change Requests
Software Planning Process
Software Development Plan
Software Verification Plan
Software Quality Assurance Plan
Software Configuration Management Plan
Requirements Standards
Software Design Standards
Software Code Standards
PSAC

5.1.2 Entry Conditions

This activity may begin once a program or project has management approval.

5.1.3 Activity Tasks

Task 1: Review Bid/Proposal

The TPM and/or PE will retrieve, organize and review available bid data. Missing information will be generated as necessary to complete the documents in this activity.

Task 2: Identify/Collect Metrics

See section 5.2.2.

Task 3: Identify Lessons Learned Strategy

After every large effort, a lesson learned meeting will be held after project completion to identify and document successes and problems encountered. In addition, possible solutions and recommendations will be captured. Metrics collection will be associated with this activity.

Task 4: Plan, Monitor and Control Software Subcontracting

See Section 5.2.8.

Task 5: Update Software Planning Documents

Software Planning Documents should reflect the project's intent during substantial project modifications. Changes to plans while actively performing development activities will be documented by creation of change requests and in the project's SAS. Types of plans used by Data Link are as follows:

- Design Plan/Project Commitment
- Project schedule and associated key milestones
- Subcontract Management Plan
- Software Development Plan (SDP) (this document)
- Software Verification Plan (SVP) (included in this document)
- Software Configuration Management Plan (SCMP) (partially in this document, see Section 8)
- Software Quality Plans
- Plan for Software Aspects of Certification (PSAC)

Task 6: Perform Configuration Control

The software planning documents are placed under developmental configuration control prior to being reviewed. For more details regarding developmental configuration control see section 8.3.1.

Task 7: Conduct Review and Capture Minutes on Software Planning

Prior to the Peer Review, the software planning document (or portions thereof) will be complete and under configuration control. This task may be done incrementally as portions of the project are completed.

A peer review will be conducted as described in section 7.3.1, using the following checklists depending on the artifact type being reviewed:

- Checklist for Producing PSAC
- Checklist for Reviewing PSAC
- Checklist for Producing SDP
- Checklist for Reviewing SDP
- Checklist for Producing SVP
- Checklist for Reviewing SVP
- Checklist for Producing SCMP
    - This checklist is only used to peer review the SCMP for *developmental* configuration management. The checklist used to peer review the SCMP for *production release* configuration management is not governed by this SDP. See Section 8.
- Checklist for Reviewing SCMP
    - This checklist is only used to peer review the SCMP for *developmental* configuration management. The checklist used to peer review the SCMP for

production release configuration management is not governed by this SDP. See Section 8.

- Checklist for Producing SW Requirements Standards
- Checklist for Reviewing SW Requirements Standards
- Checklist for Producing SW Design Standards
- Checklist for Reviewing SW Design Standards
- Checklist for Producing SW Code Standards
- Checklist for Reviewing SW Code Standards

The checklist used to peer review the Software Quality Engineering Plan [7] is not governed by this SDP. See Section 9.

Task 8: Formal Release to SCL

Upon completion of the peer review and all findings resolved, the software planning documents will be formally released to the SCL as described in section 8.3.1.

Task 9: Perform Change Control

The software plans and associated data are subject to change control mechanisms as defined in section 8.3.4.

5.1.4 Exit Conditions

The project planning process is considered complete when all of the planning documents have been formally released to the SCL.

5.2 Software Project Management Activities

This section describes typical activities performed by project management and/or the PE during the development of a project's life cycle artifacts. These activities may supplement the tasks described in the software planning processes, and are used to help manage the tasks described in the software development processes and software verification processes from two perspectives:

1. Manage the processes according to the defined schedule and budget
2. Manage the processes according to the defined processes

5.2.1 Track Project Action Items

The program Technical Performance Review will be used for recording, monitoring, tracking, and closing action items relating to each of the Data Link software development projects. The Project Engineer (PE) will track the list to ensure that all commitments are met and that all items are cleared from the list when required. Items in this review might include, for example, information required by another LRU to complete its assignments, incoming and outgoing dependencies.

Action items associated with peer reviews will be tracked in the peer review tool along with the associated peer review. The peer review tool will enforce valid state transitions, and forces required data fields to be completed for each peer review based on the current state of the change request.

5.2.2 Metrics

Review metrics, Code metrics, and Integration metrics will be collected against the project as defined below. Other metrics can be gathered on demand, as requested by the project leadership team, based on data that is typically kept during the normal software development life cycle. Examples might include: Number of CRs per delivery, Number of outstanding CRs at any given time, Number of design reviews against plan, Number of code reviews against plan, Number of test reviews against plan, etc. These are mainly project tracking type of metrics that can either be easily collected from other tools, such as ClearQuest, or CM tool, or can be tracked on an ongoing basis as part of the project management process.

Metrics are a measure or combination of measures tracked over time to identify trends. Metrics can be used to improve project estimates, identify risks, and improve the project’s process.

5.2.2.1 Review Metrics

Peer Review metric data is captured in the peer review tool itself. These metrics will be collected and analyzed as required by the PE and/or project management throughout the project schedule to benchmark the project’s peer review actuals against the plan.

5.2.2.2 Code Metrics

Code metrics data will be collected as required.

- Number of Source Lines Of Code (SLOC)
- Number of Code Modules/Files

5.2.2.3 Integration Metrics

Integration metrics will be collect as required:

- Memory utilization of each Computer Software Configuration Item (CSCI)
- Allocated execution/processing time
- Tests Run for Score Statistics (# of tests run and # of errors)

5.2.3 Track Cost and Schedule Performance

The PE is responsible for tracking and reporting the cost and schedule performance of the project. Earned value projections are initially defined in the detailed planning phase and maintained using SAP or Percentage of Completion / S-Curves. As the project progresses, the PE measures and records earned value against the baseline plan as required by the program management and/or engineering management. Engineering Administration will distribute staffing/cost actuals reports for the SAP activities according to the TPM’s TPR schedule. The PE reviews these reports, prepares the TPR presentation, and makes appropriate adjustments in plans or forecasts in SAP and the master project schedule. Actual costs (hours and funds) per SAP activity are reviewed with regard to plan during the TPR.

5.2.4 Status Meetings

The PE will attend, as required, regular TPM meetings and monthly Technical Performance Reviews (TPRs) to communicate the project status to program management and engineering management. The PE will hold project team meetings on an as-needed basis to share and receive project related information and to work out design details for the project’s features.

5.2.5 Conduct Project Reviews

The PE, or designated Subject Matter Expert (SME), should participate in project peer reviews throughout the product life cycle. Peer reviews will be conducted per section 7.3.1.

5.2.6 Maintenance of the Software Lifecycle Planning Documents

The software life cycle planning documents are subject to periodic review and updates. The PE may have higher visibility into potential changes, but any team member can propose an update or change to the planning documents by submitting a Change Request as described in section 8.3.4 (Change Control). If the Change Request is accepted, the document will be modified, peer reviewed, re-released, and distributed to all team members.

5.2.7 Risk Management

Software risks are managed in accordance with the overall program's risk management plan. A risk register may be created and maintained for each product line. The purpose of the risk register is to track program/project risks and mitigate them whenever possible, as well as to track opportunities and harvest them whenever possible. The PE is responsible for the definition, publication, and maintenance of software risks. Meetings will be held on a regular basis to review the risks and provide updates. The risk register will be maintained throughout the life of the program.

5.2.8 Subcontractor Management

The Subcontracts organization is responsible for the contractual and programmatic relationship with the subcontractor. A lead technical contact (LTC) (usually the PE) is responsible for the technical compliance of the software product. Supply Chain Quality Assurance will evaluate the subcontractor's quality system for compliance to Rockwell Collins standards. In order to use a subcontractor they must be approved in SAP.

The subcontractors are identified in the applicable PSAC or SAS. The LTC will be responsible for receiving a quote from the selected subcontractor prior to an agreement being put in place.

Rockwell Collins Subcontract organization will be responsible for the Purchase Service Agreement. A Purchase Service Agreement may be either a Time and Materials Contract or a Firm Fixed Contract. Details of the Purchase Service Agreement will be defined in the Statement of Work (SOW). The SOW will be used to identify the project, identify the work, define the artifacts to be delivered to Rockwell Collins, define a schedule of delivery of the artifacts, completion of effort, and payment due to the delivery of the artifacts. The PE, SQE and the Subcontracts organization will monitor the progress of the subcontractor throughout the subcontract life cycle.

The SOW will define how status will be received from the subcontractors. The PE and the SQE will determine as required when trips to the subcontractors' location are necessary in order to monitor the progress of the contract.

Subcontract work may be completed on the premises of the subcontractor, via remote access, or at the RCI facilities. The PE will define where this activity will be performed. Location details will be mutually agreed to between the subcontractor and the PE. Written details of the location will be in the SOW or via e-mails between the PE and the subcontractor.

5.2.8.1 Airtel ATN Subcontractor

Airtel ATN is a subcontractor that provides the Mobile Air/Ground ATN Router and Upper Layer services defined in the ATN SARPS. The Airtel ATN Router is used in the DLNKX generations of the CMU-900, CMU-4000, and RIU-40X0 Data Link products. The Airtel ATN Router is provided as a console application consisting of a main program and functional libraries. For the Data Link target products, the Airtel main program is replaced by target based system services while maintaining the exact same interfaces to the functional libraries. This allows the functional libraries to be easily tested in a target based test environment. Airtel uses the Solaris UNIX platform for their software development environment and has developed test scripts to test the router software requirements. After all tests are successfully executed on the Solaris UNIX platform, Airtel delivers the verification test artifacts to Rockwell Collins for review and analysis to assure compliance to requirements. Additionally, when fully integrated into the Data Link target products, the ATN router is functionally tested during normal operation. Airtel produces and maintains the RTCA DO-178B Software Life Cycle Data items for the ATN Router, for which Rockwell Collins SQE performs process and product audits at Airtel facilities. The deliverable artifacts supplied to Rockwell Collins are formally released to the SCL as described in section 8.3.1.

5.2.8.2 Other Subcontractors

Artifacts created or modified by subcontractors (other than Airtel ATN) will be delivered to the Rockwell Collins LTC or the LTC's delegate (usually a software lead). The LTC will capture the delivered artifacts and place them into configuration control prior to peer review. This effort may be delegated to the subcontractor by the LTC. Peer reviews will be a joint effort between Rockwell Collins and the subcontractor. Rockwell Collins SQE will be included in all peer reviews and may optionally participate. The LTC will be responsible for overseeing the configuration management of artifacts being used by the subcontractor. Rockwell Collins engineering will be responsible for overseeing the configuration management of all artifacts created or modified by the subcontractors. Subcontractors will be responsible to resolve outstanding change requests written against artifacts they have generated.

Change requests initiated by the subcontractor will be handled in the project change request system. The Rockwell Collins LTC is responsible for entering the change request into the system and informing the SQE.

Rockwell Collins engineering may integrate software that has been delivered by the subcontractor or may require the subcontractor to complete the integration of the software based on complexity and schedule. Final testing of software delivered by subcontractors will be conducted with a build generated by Rockwell Collins engineering. Final tests will be run on a hardware configuration defined by Rockwell Collins engineering. Final configuration and verification of delivered software is the sole responsibility of Rockwell Collins engineering.

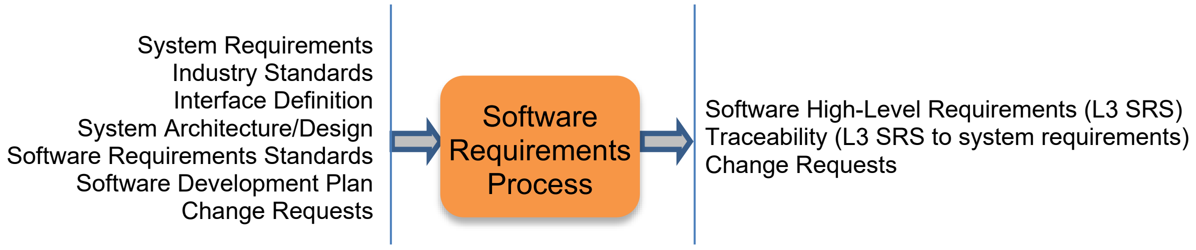
6 Software Development Processes

The following sections describe the specific activities associated with the software development processes as depicted in Figure 3-2 Software Life Cycle Process Flow. These processes include the development of software high-level requirements, software architecture and detailed design, low-level requirements, source code, and the integration process to produce the software executable.

6.1 Software Requirements Process

6.1.1 Overview

The purpose of this activity is to create and document the software high-level requirements. Industry standards (see Note) and the system requirements allocated to software are analyzed and decomposed into software high-level requirements. This activity is conducted with the interaction of the customer, systems/subsystems engineering, software engineering, and hardware engineering to create an understanding of the software portion of the problem domain and its high-level requirements.


System Requirements
Industry Standards
Interface Definition
System Architecture/Design
Software Requirements Standards
Software Development Plan
Change Requests
Software Requirements Process
Software High-Level Requirements (L3 SRS)
Traceability (L3 SRS to system requirements)
Change Requests

Note: Even though the industry standards are used to derive HLR and LLR requirements, the tracing method has been used differently in different datalink products. DLCA derives the HLR and LLR from the Industry standards, but the trace will be provided only from the LLR to Industry standards. But Link2k derives the HLR and LLR from industry standards, and the tracing will be established from both HLR and LLR.

6.1.2 Entry Conditions

Software high-level requirements definition may begin once the relevant system requirements and industry standards are sufficiently understood. Normally, this means the relevant system requirements have been documented, placed under developmental configuration control, and are ready for peer review. Activity tasks started prior to this are at risk of rework and should be coordinated with the PE.

6.1.3 Activity Tasks

Task 1: Review and Approve System Requirements Allocated to Software

This task is completed as part of the System Development Process and is intended to include members of the software development team. The intent here is that the software development team accepts the system requirements allocated to software before generating the associated software high-level requirement(s) and trace data. If a system requirement allocated to software is no longer applicable, or if the software team recognizes deficiencies in the system requirements, the software development team will issue a Change Request against the system requirements.

Task 2: Define Software Requirements from Applicable Inputs

The software development team analyzes the system requirements allocated to software, together with other applicable system data (ICD's, system architecture, and/or system design data), and industry standards to generate the software high-level requirements. The software high-level requirements will be developed in a manner consistent with the iterative development process and in accordance with the software requirement standards (see section 4.2.1). The software high-level requirements will be documented in the project L3 SRS(s) using the project's software requirement development tool.

As the system requirements allocated to software are decomposed into software high-level requirements, additional derived software high-level requirements may be generated. Derived high-level requirements will be documented in the project L3 SRS(s) and will not trace to system requirements. The justification or reason for the derived high-level requirement will be identified in the project requirement tool, and will be made available for review by the system safety assessment process.

The following considerations should be evaluated for applicability to the scope of software high-level requirements being developed for a given iteration. The developer should then ensure a complete definition of the software high-level requirements is provided for each applicable item.

- Emphasis on safety related requirements (addressed by requirement standards in section 4.2.1)
- Functional and operational requirements under each mode of operation
- Performance criteria
- Timing requirements and constraints
- Memory size constraints
- Hardware and software interfaces
- Failure detection and safety monitoring requirements
- Partitioning requirements

The developer should also consider the following DO-178B objectives when the software high-level requirements are being developed for a given iteration, as these will be evaluated during the high-level requirements verification process (section 7.4.1).

A3-1	Software high-level requirements comply with system requirements
A3-2	High-level requirements are accurate and consistent
A3-3	High-level requirements are compatible with target computer
A3-4	High-level requirements are verifiable
A3-5	High-level requirements conform to standards
A3-6	High-level requirements are traceable to system requirements
A3-7	Algorithms are accurate

Portions of the software high-level requirements document may be completed while others are either in process of development or not yet started.

Task 3: Trace Software Requirements to Higher Level Requirements

Traceability of the software high-level requirements to the system requirements allocated to software will be captured in the project requirements tool. Derived high-level requirements will have no system requirement to trace to. Tracing will be established from high-level requirement module (L3 SRS) to Industry standards if the high-level requirement is derived from the industry standards.

Task 4: Perform Configuration Control

The software high-level requirements documents are placed under developmental configuration control prior to being reviewed as described in section 8.3.1.

Task 5: Perform Change Control

The system requirements, the software high-level requirements and the associated trace data are subject to change control mechanisms as defined in section 8.3.4.

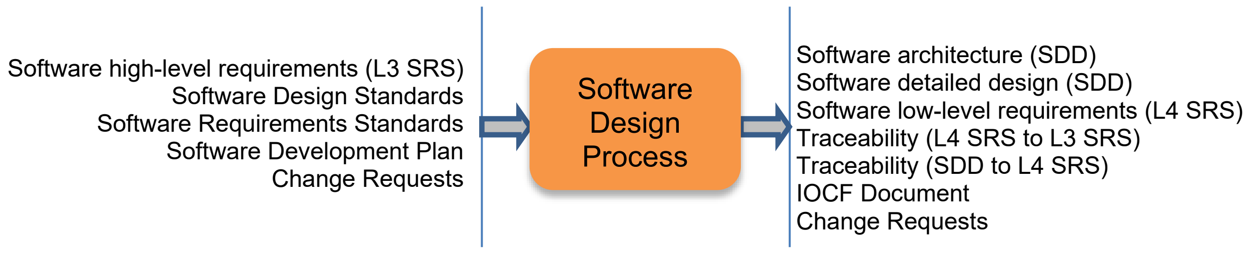
6.1.4 Exit Conditions

This activity is considered complete when the software high-level requirements and corresponding traceability to system requirements are captured and placed under developmental configuration management control.

6.2 Software Design Process

6.2.1 Overview

The purpose of this activity is to create and document the software design description, which includes the software architecture, detailed design, and low-level requirements. The architectural framework is developed to ensure conceptual integrity throughout the development of the software product. Architectural design is an iterative process that strives to find the simplest model and mechanisms that meet all high-level requirements, guide software development, and minimize anticipated future modifications. The detailed design is a refinement of the software architecture and includes the identification and description of the software elements that will satisfy the software high-level requirements. The software low-level requirements are a further decomposition of the software high-level requirements as defined by the software architecture and detailed design. The Input/Output definition is captured using the process defined in [11] to create IOCF document. For DLCA and Link2k, software low-level requirements are also derived from industry standards.


Software high-level requirements (L3 SRS)
Software Design Standards
Software Requirements Standards
Software Development Plan
Change Requests
Software Design Process
Software architecture (SDD)
Software detailed design (SDD)
Software low-level requirements (L4 SRS)
Traceability (L4 SRS to L3 SRS)
Traceability (SDD to L4 SRS)
IOCF Document
Change Requests

6.2.2 Entry Conditions

Software design activities may begin once the relevant software high-level requirements are sufficiently understood. Normally, this means the relevant software high-level requirements have been documented, placed under developmental configuration control, and are ready for peer review. Activity tasks started prior to this are at risk of rework and should be coordinated with the PE.

6.2.3 Activity Tasks

Task 1: Present a Design Approach Review (DAR) (optional)

This activity task is optional for the specific project. Depending on the resolution of the life cycle iteration, maturity of the software product, and the skill set of the development team, the PE, TPM, or Engineering Manager may request this activity as a prerequisite to formally capturing the software architecture design.

The purpose of the DAR is to ensure that the developer has an adequate understanding of the high-level requirements and that the proposed design approach is commensurate with the existing architecture framework and is aligned with the anticipated product evolution.

The DAR is an informal review where preliminary architecture design data is presented and used to facilitate the discussion. This design data is not subject to development configuration control or formal peer review, although it is suggested to archive the data in the project's repository. Typically this preliminary design data is copied into and forms the basis for the formal Software Architecture Design as described in the next task.

Task 2: Define Software Architectural Design

The software engineering team is responsible for the definition of software architecture. The software architecture will be developed in an iterative manner consistent with the software design standards (see section 4.2.2).

The software architecture will identify all major functional entities within the software system and document the interfaces, data flows, and control flows among them. Areas of emphasis include partitioning, tasking, interrupts, operating system services, hardware interfaces, major software interfaces, etc. High level descriptions of control flow, data flow, data definitions, and algorithms should be documented as part of the software architectural design activities.

High level architectural diagrams, state diagrams, sequence diagrams, etc., will be documented using either Microsoft Visio, Microsoft Powerpoint, or other suitable diagramming tool such that the architecture design description can be captured in a single Microsoft Word document (SDD).

See section 4.2.2.3 for an example SDD configuration.

The following considerations should be evaluated for applicability to the scope of software architecture being developed for a given iteration.

- Description of the software architecture defining the software structure to implement the high-level requirements
- Data dictionary and/or input/output descriptions used throughout the architecture
- The data flow and control flow of the design
- Resource limitations
- Scheduling procedures and inter-processor/inter-task communication mechanisms
- Design methods and details of their implementation
- Partitioning methods and means of preventing breaches
- Description of deactivated code and the means to enable/disable
- Rationale for design decisions

The developer should then ensure a complete description of the software architecture is provided for each applicable item.

The developer should also consider the following DO-178B objectives when the software architecture is being developed for a given iteration, as these will be evaluated during the design verification process (section 7.4.2).

A4-8	Software architecture is compatible with high-level requirements
A4-9	Software architecture is consistent
A4-10	Software architecture is compatible with target computer
A4-11	Software architecture is verifiable
A4-12	Software architecture conforms to standards
A4-13	Software partitioning integrity is confirmed

Portions of the software architecture document may be completed while others are either in process of development or not yet started.

Task 3: Capture the input/output dataflow and produce the IOCF

In this task the system level input/output interface is analyzed using the process defined in [11] to create the Input Output Common Format (IOCF) document.

Task 4: Develop Software Detailed Design

The detailed design description will identify class definitions and relationships, including class method and data descriptions, along with helper functions (e.g. container classes) that collectively constitute all of the software components created for the software product.

The detailed design description can be documented in the same SDD as the software architecture, or in a separate documentation tool (e.g. UML database) that is referenced from the SDD. In either case, a proxy module will be created in the requirements tool that contains a parallel representation of the detailed design description (e.g. a listing of the header files and/or class names) to facilitate traceability from the software design to software low-level requirements. See section 4.2.2.3 for an example SDD configuration.

Portions of the software detailed design document may be completed while others are either in process of development or not yet started.

Task 5: Develop Software Low-level Requirements

Software low-level requirements are created from a decomposition of the software high-level requirements as defined by the software architecture and detailed design. The software low-level requirements will be developed in an iterative manner consistent with the software requirements standards (see section 4.2.1). The software low-level requirements will be documented in the project L4 SRS(s) using the project's software requirement development tool.

As the software high-level requirements are decomposed into software low-level requirements, additional derived software low-level requirements may be generated. These derived low-level requirements will be documented in the project L4 SRS(s) and will not trace to software high-level requirements. The justification or reason for the derived low-level requirement will be identified in the project requirement tool, and will be made available for review by the system safety assessment process.

The following considerations should be evaluated for applicability to the scope of software low-level requirements being developed for a given iteration.

- Emphasis on safety related requirements (addressed by requirement standards in section 4.2.1)

- Detailed description of how the software satisfies the high-level requirements, including algorithms and data structures

The developer should then ensure a complete definition of the software low-level requirements is provided for each applicable item.

The developer should also consider the following DO-178B objectives when the software low-level requirements are being developed for a given iteration, as these will be evaluated during the design verification process (section 7.4.2).

A4-1	Low-level requirements comply with high-level requirements
A4-2	Low-level requirements are accurate and consistent
A4-3	Low-level requirements are compatible with target computer
A4-4	Low-level requirements are verifiable
A4-5	Low-level requirements conform to standards
A4-6	Low-level requirements are traceable to high-level requirements
A4-7	Algorithms are accurate

Portions of the software low-level requirements document may be completed while others are either in process of development or not yet started.

Task 6: Develop Traceability from Low-Level Requirements to High-Level Requirements

High-level requirements to design/low-level requirements allocation is performed to ensure that all software high-level requirements are addressed. The allocation of high-level requirements to low-level requirements is accomplished by generating traceability from the low-level requirements to the high-level requirements. This traceability is facilitated within the requirements tool by generating links from the low-level requirements module (L4 SRS) to the corresponding high-level requirements (L3 SRS). For DLCA and Link2k, the tracing will be established from low level requirement module (L4 SRS) to Industry standards if the low-level requirement is derived from the industry standards.

Task 7: Develop Traceability from Detailed Design to Low-Level Requirements

Low-level requirements to detailed design allocation are performed to ensure that all software low-level requirements are addressed within the defined software components. The allocation of low-level requirements to detailed design is accomplished by generating traceability from the detailed design proxy module (discussed in Task 2) to the low-level requirements. This traceability is facilitated within the requirements tool by generating links from the proxy module that contains the detailed software component description (SDD) to the corresponding low-level requirements in the L4 SRS.

Task 8: Perform Configuration Control

The software architecture (SDD), detailed design (SDD), IOCF, and low-level requirements documents (L4 SRS) are placed under developmental configuration control prior to being reviewed as described in section 8.3.1.

Task 9: Perform Change Control

The software architecture (SDD), detailed design (SDD), IOCF, low-level requirements (L4 SRS), and associated trace data are subject to change control mechanisms as defined in section 8.3.4.

6.2.4 Exit Conditions

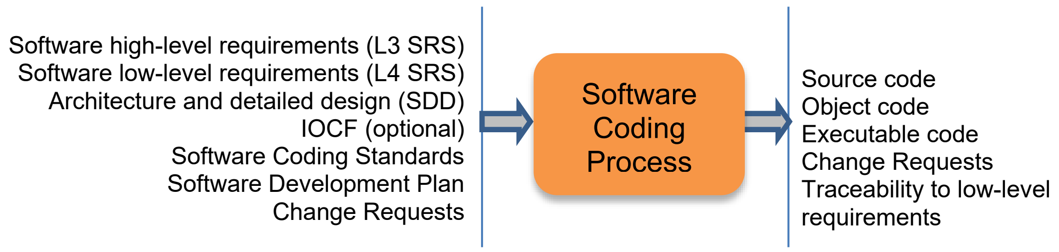
This activity is considered complete when the software architecture, detailed design, IOCF document, low-level requirements, traceability from detailed design to software low-level requirements, and traceability from software low-level requirements to software high-level requirements are captured and placed under developmental configuration management control.

6.3 Software Coding Process

6.3.1 C/C++/Assembly/APL Coding

6.3.1.1 Overview

The purpose of this activity is to develop the software source code that formulates the software product which meets the software high-level and low-level requirements. The software engineering team is responsible for generating the source code by applying sound software engineering principles, as outlined by the activity tasks described herein, to produce the final software product.


Software high-level requirements (L3 SRS)
Software low-level requirements (L4 SRS)
Architecture and detailed design (SDD)
IOCF (optional)
Software Coding Standards
Software Development Plan
Change Requests
Software Coding Process
Source code
Object code
Executable code
Change Requests
Traceability to low-level requirements

6.3.1.2 Entry Conditions

Software coding activities may begin once the relevant software low-level requirements, architecture, and detailed design data are sufficiently understood. Normally, this means the relevant software low-level requirements, architecture, detailed design, and IOCF data have been documented, placed under developmental configuration control, and are ready for peer review. The IOCF data is an optional input to the software coding process when the design change does not involve the I/O interface.

Activity tasks for Software Coding Process may start concurrently with the activity tasks for Software Design Process (section 6.2). Any Software Coding Process activity task started prior to Software Design Process is at risk of rework and should be coordinated with the PE.

6.3.1.3 Activity Tasks

Task 1: Implement Software Source Code

The software engineer will develop the software source code in accordance with the software architecture and detailed design to satisfy the software high-level and low-level requirements.

The software source code will be developed in accordance with the applicable coding standards (see section 4.2.3). Any constraints imposed on the source code language will be documented in the coding standards.

The developer should consider the following DO-178B objectives when the software source code is being developed for a given iteration, as these will be evaluated during the source code verification process (section 7.4.3).

A5-1	Source code complies with low-level requirements
A5-2	Source code complies with software architecture
A5-3	Source code is verifiable
A5-4	Source code conforms to standards
A5-5	Source code is traceable to low-level requirements (see Task 3)
A5-6	Source code is accurate and consistent

Portions of the software source code may be completed while others are either in process of development or not yet started.

Task 2: Perform static code analysis

All C/C++ code developed for Data Link products will be statically analyzed for correct syntax, potential design errors, and typical programming language pitfalls. PC-Lint is the designated tool (see Table 4-1) for this activity.

PC-Lint is a highly configurable tool that can be “tuned” to include/exclude various levels of informational, warning, and error output messages. The PE is responsible for defining the profile/configuration to be used by PC-Lint.

The PC-Lint configuration will be captured in the project specific Software Environment Configuration Index (SECI) document and maintained under development configuration control.

Task 3: Develop Software Element to Software Design Traceability

Traceability from the software source code to the detailed design will be accomplished through the use of consistent class name identifiers and/or header file names used in both the detailed design and the software code elements. Traceability from the source code to the low-level requirements will also be created and maintained. Section 6.2.3 provides a description of the detailed design proxy module that contains the class name identifiers and/or header file names. These identifiers provide a direct correlation to the source code elements, and it is through this direct correlation that traceability from software source code to detailed design is achieved.

Task 4: Perform Unit Test

Unit testing is an informal task performed by the developer on the host platform to determine if the code performs as expected. The intent of performing unit testing is to reduce the amount of software defects prior to software integration and formal test. The developer should perform unit testing as required to achieve a high level of confidence that the software performs its intended function prior to integration on the target.

At a minimum, the developer should step through the execution of the software using a host-based debugger to ensure the code performs its intended function. If necessary, the developer may also need to create test code to stimulate the inputs needed to achieve thorough unit testing in cases where software-software integration testing cannot achieve the same result.

While test code created for the purpose of unit testing does not need to be formally peer reviewed, it should be maintained in the project source code repository under developmental configuration control.

Task 5: Perform SW/HW Integration

This task is only required for coding changes to software products that have achieved an initial baseline integration with target hardware. Otherwise, the coding changes are considered initial development and thus excluded from this task.

Upon completing host based unit testing and prior to conducting a peer review of the new/modified source code, the developer should incorporate the coding changes into an informal target build and perform SW/HW integration testing in the target environment. The intent of this informal integration testing is to ensure that the new/modified code didn't "break the target build" or introduce obvious errors into the loading and runtime execution of the software product.

Task 6: Perform Configuration Control

The software source code is placed under developmental configuration control prior to being peer reviewed as described in section 8.3.1.

Task 7: Perform Change Control

The software source code is subject to change control mechanisms as defined in section 8.3.4.

6.3.1.4 Exit Conditions

This activity is considered complete when the source code has been generated, unit tested, target tested (if applicable), and placed under developmental configuration management control.

6.3.2 VAPS Coding

6.3.2.1 Overview

Datalink products which use the A661 interface to exchange messages between datalink application and the A661 Graphical Server (AGS) for the A661 HMI pages needs to create the Virtual Application Prototyping System (VAPS) and A661 definition files (BDF/TDF).

A661 definition file will be used to inform the ARINC-661 Graphic Server (AGS) of the widget data necessary to allocate the memory resources for graphics, as well as to establish a means for the datalink application to describe and update the user interface details. The definition file will be a binary file generated based on the Virtual Application Prototyping System (VAPS) widget layout. The A661 Definition file also contains a Text Data File (TDF). This file is a human readable file of the Binary Definition File (BDF).

6.3.2.2 Entry Conditions

This activity may begin once the relevant software low-level requirements, and standards (*GCD[12]*) are sufficiently understood. Normally, this means the relevant software low-level requirements have been documented, placed under developmental configuration control, and are ready for peer review.

6.3.2.3 Activity Tasks

Figure 6-1 below shows the document tree and the steps involved in the generation of VAPS and definition files (BDF/TDF).

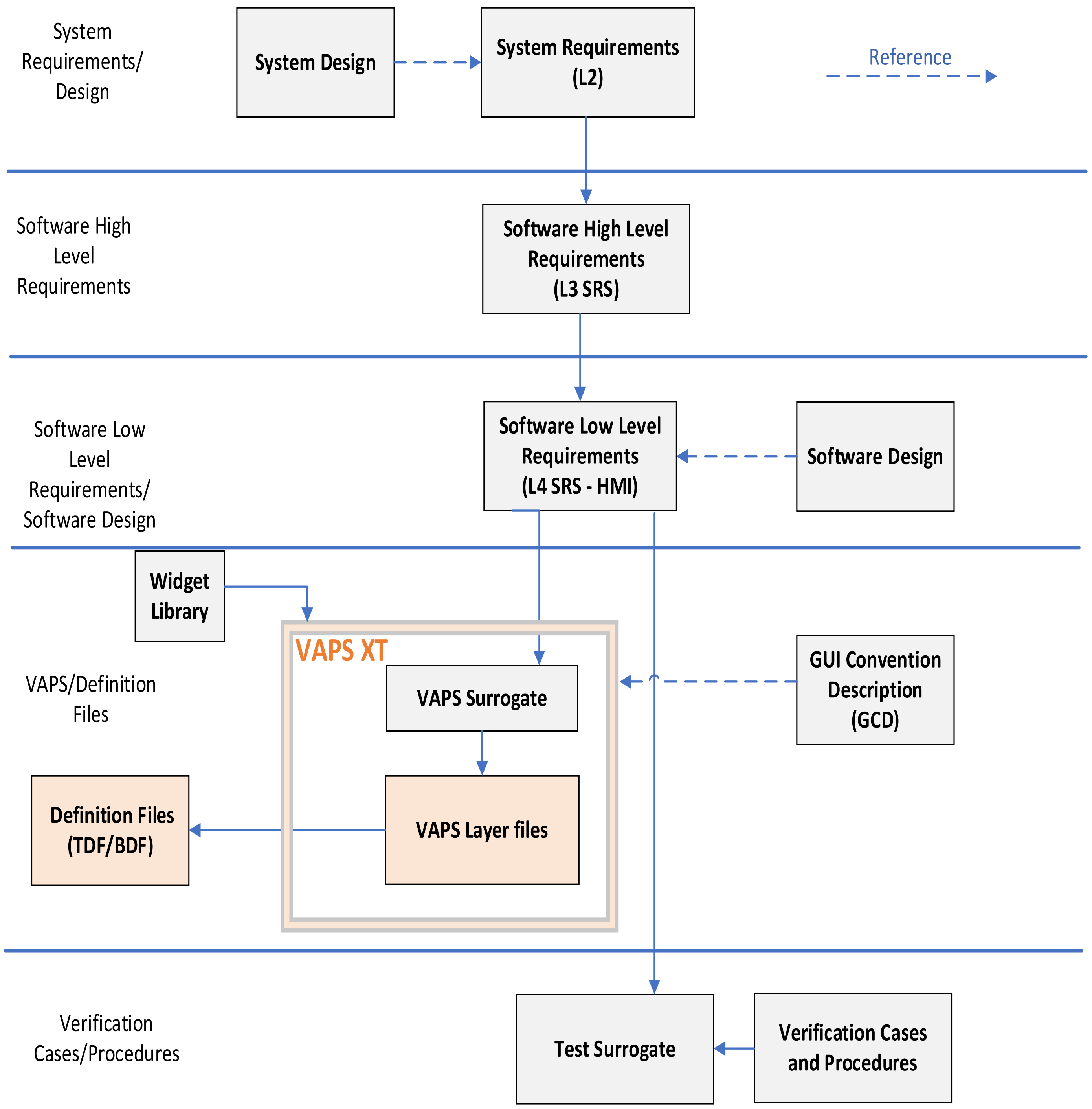

System Requirements/ Design
System Design
System Requirements (L2)
Reference
Software High Level Requirements
Software High Level Requirements (L3 SRS)
Software Low Level Requirements/ Software Design
Software Low Level Requirements (L4 SRS - HMI)
Software Design
Widget Library
VAPS XT
VAPS/Definition Files
VAPS Surrogate
GUI Convention Description (GCD)
Definition Files (TDF/BDF)
VAPS Layer files
Verification Cases/Procedures
Test Surrogate
Verification Cases and Procedures

Figure 6-1 VAPS/TDF/BDF Generation Process

Task 1: Generate VAPS Layer Files

The system design and applicable system requirements (L2) are analyzed and software requirements for the VAPS Models are captured in the software requirement documents.

VAPS layer files for BDF/TDF are created based on the software low level requirements and the GCD[12]. The GCD will be used as a system level guideline for laying out graphics in a format and for identifying graphics conventions that should be followed. The VAPS layer files will be created by defining values of data elements of individual widgets and local symbols using the VAPS XT tool. VAPSXT, a software tool from Presagis that allows the developer to place and move widgets on a 'page', as well as set the parameters for the widgets and containers for use in the code. VAPSXT creates a <proj>.vlyr file, which is a formatted XML file describing the structure of the VAPS project, and all the attributes of every widget used. DLCA VAPS needs to contain a superset of all widgets and the names should be uniform.

The developer should consider the following DO-178B objectives during the VAPS development process, as these will be evaluated during the source code verification process (section 7.4.3).

A5-1	Source code complies with low-level requirements
A5-3	Source code is verifiable
A5-4	Source code conforms to standards
A5-5	Source code is traceable to low-level requirements (see Task 3)
A5-6	Source code is accurate and consistent

Task 2: Generate TDF/BDF Files

VAPS XT tool output (VAPS layers) and widget libraries are used to generate the Binary Definition File (BDF). The Text Definition File (TDF), a text equivalent of the BDF is also generated along with the Graphical BDF.

Task 3: Develop Traceability

The development engineer will create/modify traceability between the VAPS surrogate and HMI software low level requirements. Figure 6-1 illustrates the document tree structure and traceability which the development engineer will use in establishing traces.

Task 4: Generate the code from definition file

A python script, also referred as Genesis tool, will be used to autogenerate the .cpp/.h files from the BDF/TDF, which are used directly in the C++ code. These autogenerated .cpp/.h files will be used to integrate with the datalink software with the TDF/BDF. These autogenerated files will be reviewed using the peer review process.

Task 5: Perform Informal Test

Informal Tests will be conducted prior to integration by developers/reviewers. To accomplish this, informal tests are performed by loading Binary Definition File (BDF) in the display simulation and verify the look and feel based on the VAPS.

Task 6: Perform Configuration Control

All VAPS layer files, Binary Definition File (BDF), Text Definition File (TDF), and autogenerated cpp/.h files from definition files will be placed under developmental configuration control.

Task 7: Perform Change Control

VAPS layer files, BDF, TDF, and autogenerated cpp/.h files from definition files are subject to change control mechanisms as defined in sections 8.3.3 and 8.3.4.

6.3.2.4 Exit Conditions

This activity is considered complete when the VAPS layer files, BDF/TDF, .cpp/.h files from BDF/TDF using the python script (also referred as Genesis tool) have been generated, informally tested, and placed under developmental configuration management control.

6.4 Software Integration Process

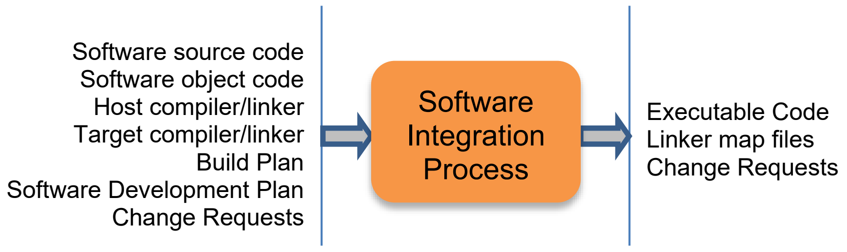
6.4.1 Overview

This activity includes the tasks that build and integrate the software elements into executable software (deliverable software build) that is loadable into target hardware and ready for hardware/software integration testing.

This activity is performed in an iterative manner based on a series of planned builds. For each build produced, there will be an informal verification activity followed by a formal verification activity.

The informal verification activity is performed by the development team and is further described as an activity task within this process (see Task 4 below).

The formal verification activity is performed by the verification team and is further described in section 7.4.4.



6.4.2 Entry Conditions

This activity may begin once the applicable inputs, or portions thereof, have been placed under developmental configuration control.

6.4.3 Activity Tasks

Task 1: Update Software Build Plan

The software build plan initially contains a defined series of planned builds that are identified during the project planning phase. Typically, these planned builds are based on change requests that represent an ordered completion of incremental functionality. As the project matures, it may become necessary to adjust the planned frequency/number of builds and/or to produce intermediate builds. The build plan should be kept updated to reflect the project's most recent build plan. The build plan is an informal document maintained under developmental configuration control.

Updates to the software build plan should be coordinated with the CCB such that assigned CRs can be accurately assigned to a planned build.

Task 2: Define a Build Procedure

The build procedure includes the instructions and files needed to build the software executable. The build process described in the build procedure must be compatible with the information necessary for releasing software to the Software Control Library. During development, build procedures are informally maintained as part of the project notes. These notes are modified as necessary during development and finalized into a formal released document (CPCI) when the software is eventually released to the Software Control Library.

The intent of the build procedure is to document what is in the build:

- Identify the baseline for this build (the release number if for the original build or the build identifier of the previous build for continuous development)
- Identify the sequence of steps to create the software build so they may be repeated
- Identify the environment in which the software will be tested (the type of hardware the software will run on: engineering units, factory boxes or simulators)
- Identify the tools needed, their exact version, and which options to use
- Identify the software elements to be used in the build
    - The version number of each modified element
    - A label or tag used to retrieve elements from source code control
- The location of the elements to add or replace existing elements
- The location of the elements to be retained in the build from previous builds
- The location for the new executable

Task 3: Integrate the Software Elements

Integrate Software with Software – As software components become available they will be compiled and linked together as specified by the build procedure to create executable object code.

Integrate Software with Hardware – As each executable software build is produced, it will be loaded into the target computer and invoked to execute. This initial level of software/hardware integration provides a level of confidence that the build procedure successfully produces a software executable.

Task 4: Verify Basic Functionality

This task is performed by the software development team as an informal verification activity to achieve a level of confidence that the software build performs as intended. This is a prerequisite before the software build is delivered to the verification team for formal verification of incorporated changes. The intent is to demonstrate that the new software build did not unintentionally introduce errors or other adverse effects and that the incorporated changes appear to meet the stated high-level requirements.

Task 5: Perform Configuration Control

Integration builds will be placed under developmental configuration control as described in section 8.3.1.

Task 6: Perform Change Control

All of the input elements used to produce a software build are subject to change control mechanisms as defined in section 8.3.4.

6.4.4 Exit Conditions

This activity is considered complete when all of the activity tasks have been successfully completed and the corresponding outputs and/or portions thereof have been placed under developmental configuration control.

7 Software Verification Plan

7.1 Organization

See section 4.1 for a description of the Data Link organization which includes software verification engineering.

The software verification team consists of engineers and technicians located in the Rockwell Collins facilities in Cedar Rapids, Iowa, and may be augmented with additional off-shore personnel located in Bangalore, India, as well as the Rockwell India Design Center (IDC) located in Hyderabad, India, or other locations as arranged by the specific project.

The entire software verification team is managed by one or more Software Verification Lead(s) located in Cedar Rapids, Iowa.

7.2 Independence

Independence is maintained throughout the software verification process commensurate with DO-178B Level C. All artifacts of the software development life cycle are reviewed and approved by at least one individual other than the author of the artifact. While not required, the following additional levels of independence are generally employed:

- High-level and low-level requirements-based functional testing and structural coverage testing (as required for Level C software) of each software subsystem is generally performed by an individual who did not develop the high-level requirements, low-level requirements, or the source code for the subsystem.
- Structural Coverage Analysis (as required for Level C software) is generally performed by an individual who did not develop the source code or the corresponding functional test cases and procedures.

The independence of the software verification testing activities is assured by having a separate team in place to perform software verification testing. Software verification artifacts (test cases, procedures, results, analysis, etc.) are formally peer reviewed and invitees include the software verification team, members of the software development team, systems, and software quality organizations.

7.3 Verification Methods

7.3.1 Review Method

Reviews will be performed in accordance with the *Peer Review Process Using PREP for the Commercial System Data Link Organization [3]*, tailored for Data Link projects as specified in the following subsections.

7.3.1.1 PREP Database Locations

The primary PREP database for Data Link projects is PREP_CS_Datalink. This database can be selected from the PREP Project Explorer by initiating the “Connect to Project” wizard and selecting the applicable subfolder(s) associated with the specific Data Link project.

The PE is responsible for ensuring all team members have access, as needed, to the project specific PREP databases.

7.3.1.2 Peer Review Checklists

All peer review checklists are documented in *Data Link Products Peer Review Checklists* [16]. There are three classifications of checklists:

- ***Producer Checklist:*** A checklist that is intended to be completed by the producer of a peer review prior to the review being held. The intent of a *Producer Checklist* is to aid the producer in ensuring all of the required process steps have been completed before holding the peer review.
- ***Reviewer Checklist:*** A checklist that is intended for each reviewer to reference during the actual review of the artifact(s). The intent of a *Reviewer Checklist* is to aid the reviewer in ensuring that the artifact(s) under review indeed meet the life cycle process objectives of DO-178B. Only one reviewer, referred to as the *Main Reviewer*, actually completes the *Reviewer Checklist* after confirming there are no *Reviewer Checklist* findings from the other reviewers.
- ***Leader Checklist:*** A checklist that is intended to be completed by the project Leader at the conclusion of the peer review. The intent of the *Leader Checklist* is to ensure that the findings have been adequately addressed and that the result of the review is satisfactory.

At least one checklist from each classification is required for a given peer review. The specific *Producer Checklist* and *Reviewer Checklist* to be used for a given artifact type is documented within this SDP in the corresponding section that describes the verification of that artifact. There is only one *Leader Checklist* and it is the same for all peer reviews.

The peer review checklists listed in *Data Link Products Peer Review Checklists* [16] cover various DO-178B Design Assurance Levels (DAL). Only the checklist questions relevant to the project's assigned DAL(s) are required to be loaded into PREP and used by the team. Refer to the project PSAC for the assigned DAL(s).

7.3.2 Analysis Methods

7.3.2.1 Requirements Based Test Coverage Analysis Method

The method used for performing requirements based test coverage analysis is to use the capabilities available in DOORS (e.g. views, filters, etc.) to identify requirement objects that are missing outgoing links as described below.

When all software high-level requirements are allocated to software low-level requirements (i.e. have outgoing links to low-level requirements), and all software low-level requirements are allocated to test cases (i.e. have outgoing links to test cases), then requirements based test coverage is considered complete.

7.3.2.2 Structural Coverage Analysis Method

This section is only applicable for Level C software.

The method used for performing structural coverage analysis is to perform a manual inspection of the coverage data reports that are generated by the structural coverage tool, and to augment these reports with annotations that document the rationale for uncovered statements. Included in the annotation will be a specific reason code that uniquely classifies the uncovered statement as acceptable or a deficiency (see below). Using a fixed set of reason codes facilitates a common approach to track and disposition action items resulting from the analysis process, and also normalizes the coverage results for metric collection.

Note that the reason codes defined below represent the recommended set to be used when performing formal SCA. There may be additional reason codes applied during dry runs (informal SCA) to facilitate maturing of the instrumentation, testing, and collection processes. If a project augments or tailors this list for formal SCA, such modifications should be documented in project-specific SCA guidelines and also described in the project SAS.

7.3.2.2.1 Acceptable Reason Codes

Acceptable reason codes are used to annotate non-covered statements as being acceptable in the software product based on a specific rationale as outlined in this section and because the analysis is able to confirm that no unexpected behavior would result if the statements were to execute. No Change Request is needed because there is no expectation to gain coverage on these statements in future structural coverage testing.

ACC-DefensiveCode

Code structures that are intended to trap against invalid/unexpected values during the integration phase and invoke a fatal error. The defensive code could be implemented as assert statements or to provide an alternate recovery path to protect the normal expected path from producing erroneous results. The black box test environment is usually not capable of forcing these invalid values.

ACC-DeactivatedCode

This reason code is only applicable to code structures that are part of Type 1 deactivated code (see section 7.3.2.2.3).

ACC-CoveredByAnalysis

This reason code is to be used for manual coverage credit on statements for which it is impractical or impossible to obtain coverage data using the coverage tool due to tool limitations, sizing or timing constraints, coding standard, language constrains, etc. The analysis needs to include the reason of why the code can’t be tested by requirement base verification, the inputs, the outputs, and the paths taken through the code.

Examples to when to use CoveredByAnalysis

- Uncovered statements (by the tool) can be analyzed as being covered because in-line code before and/or after those lines have been shown to be covered, and a logical argument can be presented that leads to the conclusion that the code was indeed executed.
- To analyze code structures that are required to support the tenets of C++ and OOD.
- To analyze code structures that are required per the project’s Coding Standards. Code structures that exist in the source file but do NOT exist in the executable (binary) software that runs on the target. Such code structures may have been instrumented together with target based code, but do not get compiled and linked into the target executable. Examples may include entire files that are host-based only and sections of files that are excluded from the build based on compiler conditionals. VectorCast does not preprocess compiler conditionals while instrumenting the code, so there may be code structures that appear in the results as being non-covered, when in fact they do not exist in the target.
- To analyze code structures that are not target code.

Note: ACC reason codes: ACC-CodingStandard, ACC-LanguageRequirement, ACC-NotTargetCode, ACC-UnusedUtilityCode, and ACC-Coveredbyinspection were used in the past to indicate a subcategory of ACC-CoveredByAnalysis. These ACC reason code may still exist within the analysis off unchanged code.

7.3.2.2.2 Deficiency Reason Codes

Deficiency reason codes are used to annotate non-covered statements for which there are no Acceptable reason codes applicable. A CR will be identified/generated for the deficiency and there is an expectation that coverage results associated with the identified statements will improve as a result of implementing the CR.

DEF- Implementation

The existing implementation has a deficiency within the code whereby the code partially implements or incorrectly implements the requirement(s) such that coverage cannot be obtained. Items meeting the criteria will have a CR written to modify the implementation.

DEF-Test

Deficiency within an existing test case where there is no explicit test procedure to exercise the code. Additional tests need to be created to gain coverage. A CR has been written to modify the test case.

DEF-TestProcedure

Deficiency whereby an explicit test exists to exercise the code, but coverage was not obtained. This may be a procedural deficiency either in the manner in which the test procedure was executed or in the manner in which coverage results were collected. A CR has been written to potentially enhance the test case to ensure coverage results are obtained in future runs.

DEF-Requirements

High-level and/or low-level requirements are not clear or are missing, resulting in a lack of tests to exercise the code. A CR has been written to clarify/add high-level and/or low-level requirements. Accordingly, new test procedures will be created to exercise the code.

DEF-DeadCode

Dead code is executable object code which, as a result of a design error cannot be executed in an operational configuration of the target computer environment and is not traceable to a system requirement, software high-level requirement, or software low-level requirement. Per DO-178B, “The code should be removed and an analysis performed to assess the effect and the need for re-verification.” If any code is identified as dead code, a CR will be generated identifying the problem. The goal is to remove all dead code in the product; however, caution should be exercised against autocratic removal of dead code late in a project’s delivery schedule, whereby a careful analysis of the existing software may be preferential to the risks associated with code removal and the generation of a new software deliverable. The CCB will have ultimate responsibility for weighing these risks at critical phases in a project’s schedule when dispositioning the CR.

7.3.2.2.3 Deactivated Code

There are two types of deactivated code:

1. (Type 1) Deactivated code which is not intended to be executed in any configuration used within a given aircraft installation. This type of code is usually associated with product-line solutions that contain functional code only applicable to certain aircraft installations. An example would be a specific display device interface (e.g. CTP) that is only applicable to aircraft installations with that type of display.
2. (Type 2) Deactivated code which is only executed in certain configurations of the target computer environment. An example of this type of code would be optional functions that can be enabled or disabled on a given aircraft installation via a program pin, license key, or other configuration input.

Per DO-178B, for Type 1 deactivated code, "A combination of analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated". Hence, any code responsible for preventing, isolating, or eliminating the deactivated code (not the deactivated code itself) will be fully tested as part of the software verification process, both functionally and structurally.

Per DO-178B, for Type 2 deactivated code, "The operational configuration needed for normal execution of this code should be established and additional test cases and test procedures developed to satisfy the required coverage objectives". Hence, this type of deactivated code will be fully tested as part of the software verification process, both functionally and structurally.

7.3.2.3 Link Analysis Method

The Link Analysis method is used to ensure that the output of the software integration process is complete and correct. This analysis will focus on three primary objectives:

1. Ensure there are no incorrect hardware addresses. The linker map file will be inspected for linker segment addresses, code and data space allocations, and I/O assignments to hardware addresses to ensure that all address ranges are within the defined boundary allocated to the software product.
2. Ensure there are no memory overlaps. The linker map file will be inspected to ensure that the linker segment addresses, code and data space allocations, and I/O assignments have all been assigned to unique ranges within the memory map.
3. Ensure there are no missing components. The linker map file will be inspected to ensure that expected object files and libraries were included as inputs to the link process.

The results of the analysis will be documented in the SVPR and peer reviewed as described in section 7.3.1 using the following checklists:

- Checklist for Producing Software Integration Review
- Checklist for Reviewing Software Integration

7.3.2.4 Timing Analysis Method

There are two timing analysis methods described, depending on the time partitioning requirements levied on the software product:

1. Run to Completion timing. In this environment, individual partitions are not constrained to run within a predetermined time budget. The analysis method in this environment will be to measure the average and peak cycle times for each partition, and then perform an aggregate analysis across all partitions to measure the overall round-robin task timing. These times will then be compared to performance requirements against the aggregate system to obtain a measure of timing margin in the system.

2. Fixed Cycle Time: In this environment, individual partitions are allocated a fixed cycle time budget. The analysis method in this environment will be to measure average and peak cycle times for each partition and compare those against the allocated budget to obtain a measure of timing margin in the system.

The results of the timing analysis will be documented in the SVPR and a Change Request will be generated if it is determined there is insufficient margin.

7.3.2.5 Stack Analysis Method

The objective of the Stack Analysis is to assure that the size of each functional task stack in the processor is sufficient to prevent a stack overflow from occurring under any potential operating condition. The method used to perform this analysis will utilize empirical measurements captured during high-level and low-level requirements based testing. Each task stack will be initialized to constant values and specialized test code will walk the stack looking for high water marks (where the initialized constant value has changed). These measurements will be compared to the allocated stack sizes to obtain a measure of each stack margin.

The results of the stack analysis will be documented in the SVPR and a Change Request will be generated if it is determined there is insufficient stack capacity.

7.3.3 Test Methods

The software verification team performs *hardware/software integration testing* and *software integration testing* based on the verified test cases and test procedures as described in DO-178B section 6.4. Note that some test execution may occur in a host environment to facilitate *software integration testing* and, for Level C software, to obtain structural coverage results. Where tests are not constrained by a host-based test harness, they will also be run in the target environment to ensure no errors exist in that environment.

The specific methods used for *hardware/software integration testing* and *software integration testing* are captured in project specific Verification User Guides.

7.4 Verification Processes

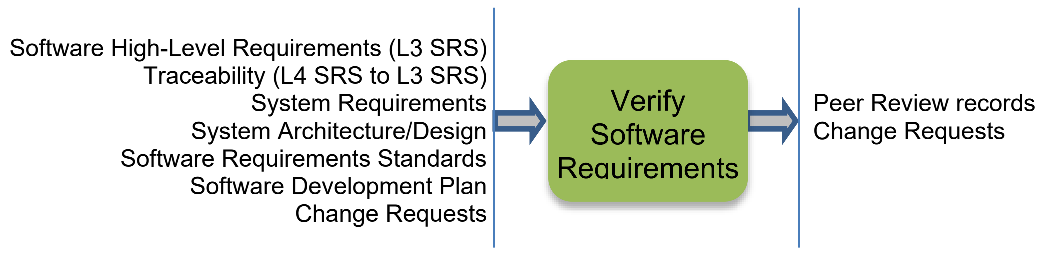
7.4.1 Verify Software Requirements

7.4.1.1 Overview

The purpose of this activity is to verify the software high-level requirements have been developed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A3-1	Software high-level requirements comply with system requirements
A3-2	High-level requirements are accurate and consistent
A3-3	High-level requirements are compatible with target computer
A3-4	High-level requirements are verifiable
A3-5	High-level requirements conform to standards
A3-6	High-level requirements are traceable to system requirements
A3-7	Algorithms are accurate

The verification method used for this activity will be the Review method (section 7.3.1).


Software High-Level Requirements (L3 SRS)
Traceability (L4 SRS to L3 SRS)
System Requirements
System Architecture/Design
Software Requirements Standards
Software Development Plan
Change Requests
Verify Software Requirements
Peer Review records
Change Requests

7.4.1.2 Entry Conditions

Software high-level requirements verification may begin once the relevant system requirements allocated to software have been reviewed and approved, and the software high-level requirements to be verified have been placed under developmental configuration control.

For small scope CRs, this activity may be done in conjunction with verifying the relevant system requirements. Reference the Glossary for a definition of a small scope CR.

7.4.1.3 Activity Tasks

Task 1: Conduct Peer Review

A peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing High-Level Requirements Review
- Checklist for Reviewing High-Level Requirements

7.4.1.4 Exit Conditions

This activity is considered complete when all findings in the peer review have been closed and the peer review itself is closed.

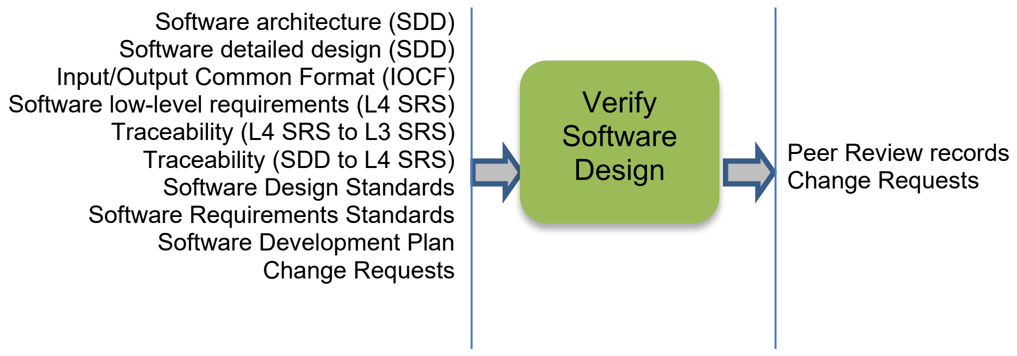
7.4.2 Verify Software Design

7.4.2.1 Overview

The purpose of this activity is to verify the software design description, which includes the software architecture, detailed design, and low-level requirements, have been developed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A4-1	Low-level requirements comply with high-level requirements
A4-2	Low-level requirements are accurate and consistent
A4-3	Low-level requirements are compatible with target computer
A4-4	Low-level requirements are verifiable
A4-5	Low-level requirements conform to standards
A4-6	Low-level requirements are traceable to high-level requirements
A4-7	Algorithms are accurate
A4-8	Software architecture is compatible with high-level requirements
A4-9	Software architecture is consistent
A4-10	Software architecture is compatible with target computer
A4-11	Software architecture is verifiable
A4-12	Software architecture conforms to standards
A4-13	Software partitioning integrity is confirmed

The verification method used for this activity will be the Review method (section 7.3.1).


Software architecture (SDD)
Software detailed design (SDD)
Input/Output Common Format (IOCF)
Software low-level requirements (L4 SRS)
Traceability (L4 SRS to L3 SRS)
Traceability (SDD to L4 SRS)
Software Design Standards
Software Requirements Standards
Software Development Plan
Change Requests
Verify Software Design
Peer Review records
Change Requests

7.4.2.2 Entry Conditions

Software design verification may begin once the relevant software high-level requirements have been reviewed and approved, and the software architecture, detailed design, IOCF document, and low-level requirements to be verified have been placed under developmental configuration control. Note that verification of artifacts related to software architecture, detailed design, IOCF document, and low-level requirements can be accomplished individually (i.e. it is not required to verify them collectively in a single peer review).

For small scope CRs, this activity may be done in conjunction with verifying the relevant software high-level requirements. Reference the Glossary for a definition of a small scope CR.

7.4.2.3 Activity Tasks

Task 1: Conduct Peer Review of Software Architecture and Detailed Design

A peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Software Architecture Review
- Checklist for Reviewing Software Architecture

Task 2: Conduct Peer Review of Software Low-Level Requirements

A peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Low-Level Requirements Review
- Checklist for Reviewing Low-Level Requirements

Exception: Checklists are not required when peer reviewing changes to software low level requirements when the only change made was a change to the Test Method attribute.

Task 3: Conduct Peer Review of IOCF Document

A peer review for the IOCF will be conducted as described in [11].

7.4.2.4 Exit Conditions

This activity is considered complete when all findings in each of the peer reviews have been closed and each peer review itself has been closed.

7.4.3 Verify Software Source Code

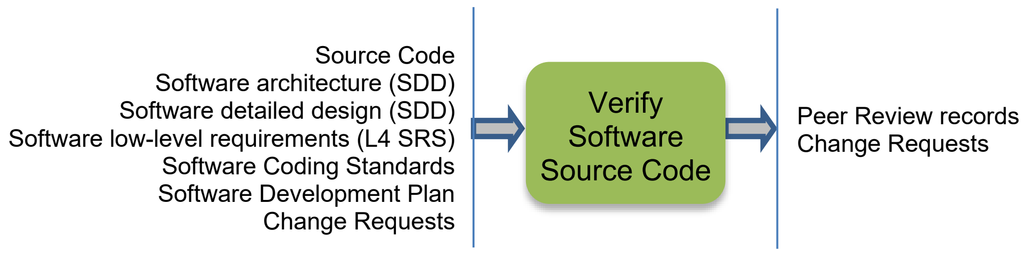
7.4.3.1 Verify Source Code implemented in C/C++/Assembly/APL

7.4.3.1.1 Overview

The purpose of this activity is to verify the software source code has been developed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A5-1	Source code complies with low-level requirements
A5-2	Source code complies with software architecture
A5-3	Source code is verifiable
A5-4	Source code conforms to standards
A5-5	Source code is traceable to low-level requirements
A5-6	Source code is accurate and consistent

The verification method used for this activity will be the Review method (section 7.3.1).


Source Code
Software architecture (SDD)
Software detailed design (SDD)
Software low-level requirements (L4 SRS)
Software Coding Standards
Software Development Plan
Change Requests
Verify Software Source Code
Peer Review records
Change Requests

7.4.3.1.2 Entry Conditions

Software source code verification may begin once the relevant software architecture, detailed design, and low-level requirements have been reviewed and approved, and the software elements to be verified have been placed under developmental configuration control.

For small scope CRs, this activity may be done in conjunction with verifying the relevant software architecture, detailed design, and/or low-level requirements. Reference the Glossary for a definition of a small scope CR.

7.4.3.1.3 Activity Tasks

Task 1: Conduct Peer Review

For peer reviews of C/C++/Assembly source code, a peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing C/C++/Assembly Source Code Review
- Checklist for Reviewing C/C++/Assembly Source Code

For peer reviews of application data sets, a peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing APL Code Review
- Checklist for Reviewing APL Code

7.4.3.1.4 Exit Conditions

This activity is considered complete when all findings in the peer review(s) have been closed and each peer review itself has been closed.

7.4.3.2 Verify Source Code implemented in VAPS

7.4.3.2.1 Overview

The purpose of this activity is to verify that the VAPS layer files have followed the GCD[12] guidelines and have correctly and completely implemented the software low level requirements in accordance with the process defined in this SDP and to satisfy the following objectives from DO-178B. The graphical Binary Definition Files (BDF) are compiled from the VAPS layer files and are part of the review.

A5-1	Source code complies with low-level requirements
A5-2	Source code complies with software architecture
A5-3	Source code is verifiable
A5-4	Source code conforms to standards
A5-5	Source code is traceable to low-level requirements
A5-6	Source code is accurate and consistent

The verification method used for this activity will be the Review method (section 7.3.1).

7.4.3.2.2 Entry Conditions

VAPS layer files and the source code verification may begin once the relevant software architecture, detailed design, and low-level requirements have been reviewed and approved. The VAPS layer files and the source code to be verified have been placed under developmental configuration control.

7.4.3.2.3 Activity Tasks

Task 1: Conduct Peer Review

For peer reviews of VAPS layer files, TDF, BDF, and generated source code files (.cpp/.h) by python script (also referred as Genesis tool), a peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing C/C++/Assembly Source Code Review
- Checklist for Reviewing C/C++/Assembly Source Code

7.4.3.2.4 Exit Conditions

This activity is considered complete when all findings in the peer review(s) have been closed and each peer review itself has been closed.

7.4.4 Verify Software Integration

7.4.4.1 Overview

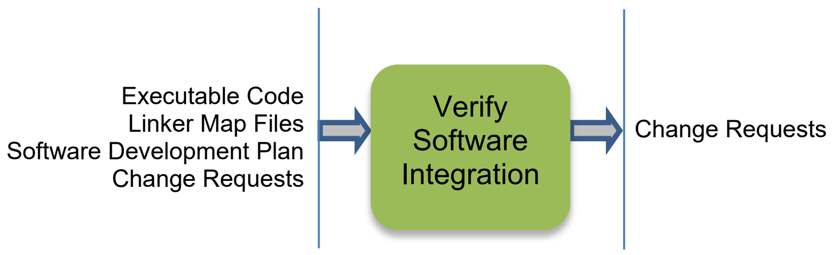
The purpose of this activity is to verify the integration process has been completed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A5-7	Output of software integration process is complete and correct

The elements of this objective are to ensure that the executable code has correct hardware addresses, contains no memory overlaps, and is not missing software components.

These elemental objectives can be met by demonstrating successful loading and startup on the target hardware and/or performing a link analysis on the linker map file.

The verification method used for this activity will be the Test method (section 7.3.3) and, if needed as described below, the Link Analysis Method (section 7.3.2.3).



7.4.4.2 Entry Conditions

Software integration verification may begin once the relevant software source code has been reviewed and approved, and a labeled software build has been delivered to the verification team and has been placed under developmental configuration control.

7.4.4.3 Activity Tasks

Task 1: Load and Run the Executable

If the loading operation is performed by a data loader application that validates and loads the executable software, *and* if there exists separate platform software in the target environment that monitors memory boundaries, then this task is sufficient to verify the integration process.

Task 2: Perform Link Analysis (if needed)

If the conditions specified in Task 1 are not present in the target environment, then a link analysis will need to be completed using the methods described in 7.3.2.3.

7.4.4.4 Exit Conditions

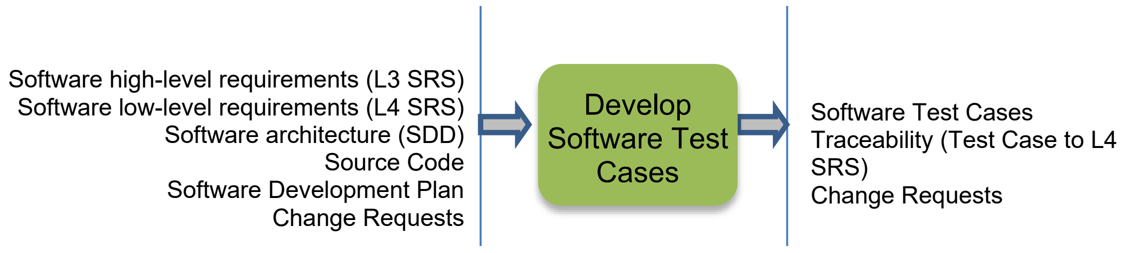
This activity is considered complete when:

1. A successful load and startup of the executable code is achieved
2. The results of the link analysis (if needed) are documented in the project Software Verification Procedures and Results document.

7.4.5 Develop Software Test Cases

7.4.5.1 Overview

The purpose of this activity is to develop the software verification test cases. The software verification test cases describe, at a high level, the actions (analysis, inspection, or test) that will be performed to verify that the software satisfies the software high-level and low-level requirements. To accomplish this, the test cases are developed from the software low-level requirements and the associated software high-level requirements.


Software high-level requirements (L3 SRS)
Software low-level requirements (L4 SRS)
Software architecture (SDD)
Source Code
Software Development Plan
Change Requests
Develop Software Test Cases
Software Test Cases
Traceability (Test Case to L4 SRS)
Change Requests

Note: safety related requirements will flow down from higher level system requirements and handled per section 4.2.1.5. Test Cases for safety related requirements are developed following the same process as all other high-level and low-level requirements.

7.4.5.2 Entry Conditions

This activity may begin once the relevant software high-level and low-level requirements are sufficiently understood. Normally, this means the relevant software high-level and low-level requirements have been documented, placed under developmental configuration control, and are ready for peer review. Activity tasks started prior to this are at risk of rework and should be coordinated with the PE.

7.4.5.3 Activity Tasks

Task 1: Identify the Verification Method

Individual requirement objects managed in DOORS modules will include an attribute which identifies the verification method by which the requirement is intended to be verified. For software high-level requirements, this attribute will defer to the software low-level requirements for the specific verification method(s) to be used. For software low-level requirements, this attribute will identify one of the following verification methods:

1. Target-based testing – This method will be used for executing the software on the target hardware. Target-based testing includes hardware/software and software/software integration testing. This method is the preferred method of verification. However, when target-based testing is not appropriate, achievable or feasible, an alternate verification method will be performed to verify the requirement.
2. Host-based testing – This method requires compiling the source code with the host compiler to run on the host developing computer. For Level C software, this method will be used for obtaining structural coverage when possible, based on engineering determination that the host based testing is equivalent to the target based testing.
3. Inspection – This method will be used primarily when only a visual observation of the software requirement's implementation is possible or necessary. Typically this method is used when the result of the visual observation is either compliant (i.e., pass) or non-compliant (i.e., fail) with no additional justification needed. Inspection test method can be used when the available test environment did not support the functional testing of the requirement and the inspection addresses the criteria in the inspection checklist in *Data Link Products Peer Review Checklists* [16].
4. Analysis – This method will be used primarily on algorithmic intense software requirements. Typically the analysis involves mathematics to assume the verification result of the software requirement's implementation. The result is based on a series of repeatable steps with each step deduced to be valid. This method is the least objective/structured of the verification methods.

Guidance on which verification method to select for a given requirement is based on the following considerations:

- Team member concurrence
- DO-178B guidelines
- The criticality of the software requirement
- The effort associated with the development of the verification method
- The resulting quality of the verification method in achieving verification of the software requirement

Regardless of the verification method chosen, test cases will be created and will be accompanied by a test procedure.

Task 2: Develop Software Test Cases

Test cases are defined based on a detailed analysis of the software low-level requirements with consideration toward satisfying the related software high-level requirements. Test cases are generally agnostic of the specific implementation and code structure, hence the name requirements-based testing. Requirements-based testing is used to show that the software performs its intended function and that the software meets its high-level and low-level requirements. Test cases will contain both normal range test cases and abnormal range (robustness) test cases to demonstrate the software robustness and capability to operate appropriately to all inputs and conditions. Test cases will stress boundary conditions (maximums, minimums, and any areas of discontinuity) both in normal and abnormal ranges. The most common requirements-based test cases are developed from the target-based testing verification method, which verifies software/software integration and software/hardware integration. The testing method will address correct and incorrect behavior in the following: algorithms, loops, logic decisions, combinations of input conditions, responses to missing or corrupted input data, handling of exceptions, computation sequences, and algorithm precision, accuracy, or performance.

Equivalence Class

Exhaustive testing is impractical so the input requirement clauses and output requirement clauses will be partitioned into equivalence class test conditions (e.g. positive one-digit numbers, positive two-digit numbers, positive numbers less than 1, zero, negative one-digit numbers, etc.). For an equivalence class, a representative value of a class is equivalent to any other value in that class. There is some crossover with the boundary value analysis method.

Test cases include both normal and abnormal range test cases that show the software responds appropriately to all types of inputs and conditions. Normal range test cases should verify inputs at all minimum and maximum boundaries for data ranges, as well as a nominal value. All other values within the data range boundaries are considered part of the normal range equivalence class and do not need to be separately tested. Abnormal range test cases should verify inputs adjacent to the normal minimum and maximum values, but outside of the data range boundaries. All other values outside the data range boundaries are considered part of the abnormal range equivalence class and do not need to be separately tested.

Boundary Value

Test conditions may be chosen at the boundaries of the input and output domains as described in Table 7-1.

Table 7-1 Boundary Value Test Conditions

Input Or Output	Normal Range Test Conditions	Abnormal Range Test Conditions
Range of values	Each end of the input range	Beyond the ends of the range
Number of values	Minimum and Maximum number of output values	One beneath, one beyond
Ordered Set	First and last elements of the set	Anything out of the set
Time or Size	At the time/size limit	Above and below the limit

Stress Testing

Test conditions and test cases may be based on stress testing, which draws from past experience or intuition to speculate on certain probable types of errors. This could also include testing the negative of a requirement.

Software/Software Integration Test Cases

The objective of software/software integration test cases is to ensure the software components interact correctly with each other (i.e., module to module interfaces are correct) and satisfy the software requirements. The software/software integration testing will address correct and incorrect behavior in the following: initialization of variables and constants, parameter passing, ensuring data is not corrupt, end-to-end numerical resolution, and sequencing of events and operations. Test case selection for software/software integration testing will re-use system level and software level requirements-based tests, when possible. If existing tests are not sufficient for complete software/software integration, a new test(s) will be created. Test cases generated for software/software integration may be tested on the target or host.

Hardware/Software Integration Test Cases

The objective of hardware/software integration test cases is to ensure the software in the target hardware will satisfy the software requirements. The hardware/software integration testing will address correct and incorrect behavior in the following: hardware and external interfaces, interrupt handling, and access of hardware information (such as discrete inputs and outputs), execution time, software response to hardware transients or hardware failures, memory mapping, built-in test, feedback loops, memory management hardware, stack overflow, compatibility of field-loadable software, and software partitioning. Test case selection for hardware/software integration testing will re-use system level and software level requirements based tests, when possible. If existing tests are not sufficient for complete hardware/software integration, a new test(s) will be created.

Test cases will be created by choosing an analysis method to generate test conditions, test condition combinations, and hence, test cases. Valid analysis methods include equivalence classes, boundary value and stress testing as described above. Test condition combination analysis will be performed to ensure the effects of various combinations of input test conditions and output test conditions are considered, while test cases are being generated.

Task 3: Trace Software Test Cases to Software Requirements

Software verification test cases will be traced directly to the software low-level requirements by establishing direct links from each test case to applicable low-level requirements in the L4 SRS.

Traceability from the verification test cases to the software high-level requirements is accomplished indirectly through the traceability from software low-level requirements to software high-level requirements.

Task 4: Perform Configuration Control

Software verification test cases will be placed under developmental configuration control as described in section 8.3.1.

Task 5: Perform Change Control

Software verification test cases are subject to change control mechanisms as defined in section 8.3.4.

7.4.5.4 Exit Conditions

This activity is considered complete when all of the activity tasks have been successfully completed and the corresponding outputs and/or portions thereof have been placed under developmental configuration control.

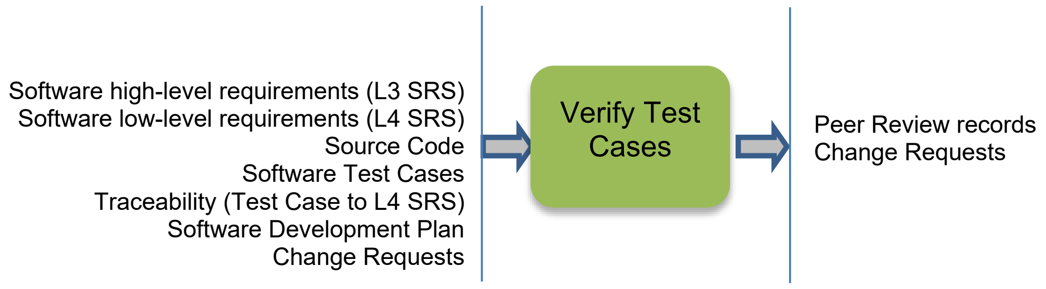
7.4.6 Verify Software Test Cases

7.4.6.1 Overview

The purpose of this activity is to verify the software test cases have been developed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A7-3	Test coverage of high-level requirements is achieved
A7-4	Test coverage of low-level requirements is achieved

The verification method used for this activity will be the Review method (section 7.3.1) and the Requirements Based Test Coverage Analysis method (7.3.2.1).


Software high-level requirements (L3 SRS)
Software low-level requirements (L4 SRS)
Source Code
Software Test Cases
Traceability (Test Case to L4 SRS)
Software Development Plan
Change Requests
Verify Test Cases
Peer Review records
Change Requests

7.4.6.2 Entry Conditions

Software test case verification may begin once the relevant software high-level and low-level requirements have been reviewed and approved, and the software test cases to be verified have been placed under developmental configuration control.

For small scope CRs, this activity may be done in conjunction with verifying the relevant software high-level and low-level requirements. Reference the Glossary for a definition of a small scope CR.

7.4.6.3 Activity Tasks

Task 1: Perform Requirements Based Test Coverage Analysis

This task is to ensure that the test cases under review are complete and sufficient to test each high-level and low-level requirement in the relevant SRS(s). A requirements based test coverage analysis will be performed, as described in section 7.3.2.1, to confirm and document that test cases exist for each high-level and low-level requirement and the requirements traceability is complete. Change Requests will be created against the requirements and/or test cases for incomplete coverage.

Task 2: Peer Review the Software Test Cases

A peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Test Case Review
- Checklist for Reviewing Test Cases
- Checklist for Producing Code Inspection if Inspection Test Method is used

The peer review is performed against the test cases to assure they are accurate prior to developing the corresponding test procedure(s). The test cases should be reviewed for completeness, accuracy, consistency, compatibility with target computer (when test cases are target based) and test environment, and traceability. The test cases should be reviewed for completeness and accuracy for both normal and abnormal test conditions.

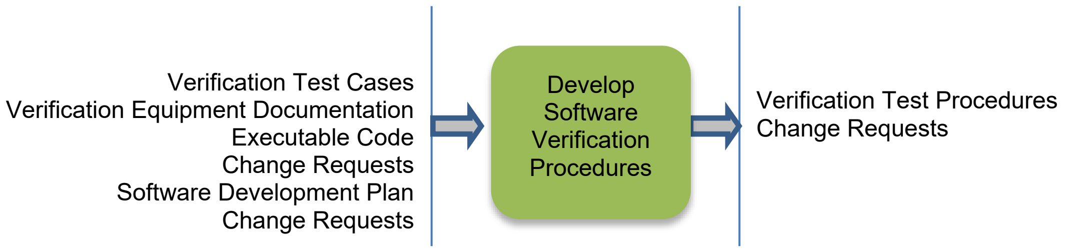
7.4.6.4 Exit Conditions

This activity is considered complete when all findings in the peer review have been closed and the peer review itself is closed.

7.4.7 Develop Software Test Procedures

7.4.7.1 Overview

The objective of this activity is to provide step-by-step instructions for how each test case is to be set up and executed, how the test results are evaluated, and the configuration of the test environment used to execute the tests.


Verification Test Cases
Verification Equipment Documentation
Executable Code
Change Requests
Software Development Plan
Change Requests
Develop Software Verification Procedures
Verification Test Procedures
Change Requests

7.4.7.2 Entry Conditions

This activity may begin once the relevant test cases are sufficiently understood. Normally, this means the relevant test cases have been documented, placed under developmental configuration control, and are ready for peer review. Activity tasks started prior to this are at risk of rework and should be coordinated with the PE.

7.4.7.3 Activity Tasks

Task 1: Develop Software Verification Test Procedures

This task is to develop software verification test procedures based on the software verification test cases. A verification test procedure consists of a test procedure document and optional executable test programs (for non-manual test procedures).

Test procedures for each test case or set of test cases will be developed and documented in the Software Test Procedure document. The test procedures should be implemented as executable procedures wherever possible. If possible, automated executable test programs should be created to allow for faster regression testing and retesting.

Automated executable test programs would be considered if

- the test environment supports the capability to execute automated tests (e.g. scripting) and
- The output of the tool is verified through another DO-178B process or the tool has been qualified.

The test environment and the specific tools used will be defined in the project specific PSAC.

Each test procedure document will include or record the following to assure executing the test procedure is deterministic and repeatable:

- Identification or reference to the test environment configuration, including target hardware configuration (if applicable), necessary equipment, and software build
- Identification or reference to the person and date of test procedure execution
- Step-by-step instructions on the set up and the execution of the procedure and/or reference to the executable test programs to be executed for each test case
- Instructions and criteria for the evaluation of the test pass/fail results
- Collection and retention of the test results

Task 2: Dry Run Test Procedures (optional)

If the software executable code is available at the time the test procedures are being developed, then a dry run of the test procedures should be performed to ensure correctness of the procedures before using them for run-for-score testing.

Task 3: Perform Configuration Control

Software verification test procedures will be placed under developmental configuration control as described in section 8.3.1.

Task 4: Perform Change Control

Software verification test procedures are subject to change control mechanisms as defined in section 8.3.4.

7.4.7.4 Exit Conditions

This activity is considered complete when all of the activity tasks have been successfully completed and the corresponding outputs and/or portions thereof have been placed under developmental configuration control.

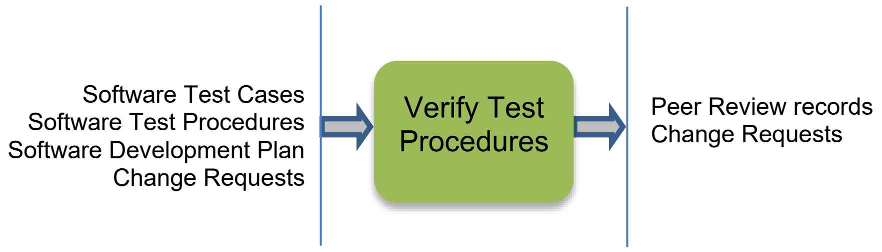
7.4.8 Verify Software Test Procedures

7.4.8.1 Overview

The purpose of this activity is to verify the software test procedures have been developed in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A7-1	Test procedures are correct

The verification method used for this activity will be the Review method (section 7.3.1).



7.4.8.2 Entry Conditions

Software test procedure verification may begin once the relevant software test cases have been reviewed and approved, and the software test procedures to be verified have been placed under developmental configuration control.

For small scope CRs, this activity may be done in conjunction with verifying the relevant software test cases. Reference the Glossary for a definition of a small scope CR.

7.4.8.3 Activity Tasks

Task 1: Peer Review the Software Test Procedures

A peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Test Procedure Review
- Checklist for Reviewing Test Procedures

The peer review is performed against the test procedures to ensure that:

- Test Cases are accurately implemented
- Test Procedures include a means to capture the identification of the test environment configuration, target hardware configuration, and software build
- Test Procedures include a means to capture the identification of the person running the tests and the date of each test procedure execution
- Test Procedures include detailed instructions for setup and execution
- Test Procedures include instructions for evaluation of the test results
- Test Procedures identify the test environment
- Test Procedures identify how to collect and retain the test results

7.4.8.4 Exit Conditions

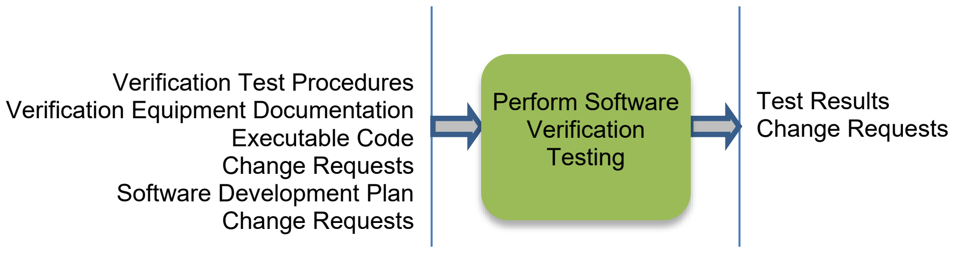
This activity is considered complete when all findings in the peer review have been closed and the peer review itself is closed.

7.4.9 Perform Software Verification Testing

7.4.9.1 Verification of Requirements implemented using C/C++/Assembly/APL coding

7.4.9.1.1 Overview

The objectives of this activity are to demonstrate that the software satisfies its high-level and low-level requirements and to demonstrate with a high level of confidence that errors which could lead to unacceptable failure conditions have been removed.


Verification Test Procedures
Verification Equipment Documentation
Executable Code
Change Requests
Software Development Plan
Change Requests
Perform Software Verification Testing
Test Results
Change Requests

Software verification testing is performed to demonstrate functional correctness of the software product and, for Level C software, to collect structural coverage data. Note that verification testing to collect structural coverage data is run in *addition* to verification testing for functional correctness.

Testing for functional correctness will be performed in the target environment to the maximum extent possible, with exceptions for host-based testing in cases where a host-based test harness is required.

For Level C software, testing to collect structural coverage data will be performed primarily in the host environment, augmented by some testing in the target environment as necessary to collect coverage on target-only code. The rationale for host-based structural coverage testing is twofold: *i*) it facilitates code instrumentation and data collection techniques that are otherwise limited by target resources, and *ii*) testing to collect structural coverage data is performed in addition to target-based testing for functional correctness.

7.4.9.1.2 Entry Conditions

Software testing activities may begin once the applicable inputs, or portions thereof, have been placed under developmental configuration control, reviewed and approved.

7.4.9.1.3 Activity Tasks

Task 1: Develop Verification Strategy for Configured Software Build

This task will identify the set of software test procedures to be executed for a build based on an analysis of the changes. Integration testing and regression testing are performed on various baselines. Integration testing is incrementally verifying changes and/or additional functionality. Regression testing is verifying that the original functionality has not been impacted. The regression tests are previously used integration tests (i.e., test procedures). The change analysis considers the new and/or modified source code, the new and/or changed software high-level requirements, and the new and/or changed software low-level requirements.

Task 2: Dry Run Verification Procedures

A dry run of the applicable test procedures should be performed to catch any problems prior to formal (run-for-score) execution of the test procedure. This task may have been completed during development of the verification test procedures.

Task 3: Software Verification Readiness Review (SVRR) (optional)

A Software Verification Readiness Review is an optional activity that is highly recommended for new developments and major revisions. The SVRR allows coordination and communication on the developmental status among the software team and other engineering teams on the project. The SVRR is held to ensure that all elements are in place and of proper status to allow for the formal software verification testing to be performed. The review will cover the following items:

- Results of the dry run including the version of the software, hardware, and verification procedures used to perform the dry run
- Status of the software (e.g. Change Request status, configuration identification)
- Status of the hardware
- Status of the verification test procedures
- Configuration status of the test environment
- Verification schedule

Task 4: Perform Testing

This task is to execute the software test procedures and capture the results. When testing for functional correctness, the test procedures are executed using an un-instrumented software integration build. When testing to collect structural coverage data (Level C software), the test procedures are executed using an instrumented software integration build. The results file from executing the test procedure should contain at a minimum the following information:

- Date
- Verifier
- Software integration build reference
- HW reference (if applicable)
- Actual Results reference (if applicable)
- Overall test result (Pass/Fail)

During the test procedure execution it may also be necessary to debug the test procedures themselves. Any changes to the test procedures will be documented and a review of the changes will be conducted.

Actual test results generated will be compared with the expected results documented in the test procedure typically as the procedure is being executed. A test procedure will only pass when the actual test results agree with the expected test results. Change Requests will be created for each failed test procedure to determine and correct the cause of failure (i.e., source code errors, procedural errors, high-level and/or low-level requirement flaws).

A record will be kept of which test procedures are run against which software builds. All test cases, test procedure documents, test procedure executable test programs, actual test/verification results, and structural coverage results (Level C software) will be captured in the project SVPR Computer Program Configuration Item (CPCI) and referenced in the project Software Verification Procedures and Results (SVPR) document, as described in section 7.4.10.3.

7.4.9.1.4 Exit Criteria

This activity is considered complete when all outputs and/or portions thereof have been placed under developmental and/or production configuration control.

7.4.9.2 Verification of Requirements implemented using VAPS

7.4.9.2.1 Overview

The objectives of this activity are to demonstrate that the software satisfies its high-level and low-level requirements and to demonstrate with a high level of confidence that errors which could lead to unacceptable failure conditions have been removed.

Test cases for VAPS are developed from the software low-level requirements (L4 SRS – HMI) and the associated software high-level requirements. Widget Mapping and VAPS figures are documented in low level requirement document (L4 SRS – HMI).

Verification activities will be performed as per the process steps explained in Section 7.

7.4.9.2.2 Entry Conditions

This activity may begin once the relevant software low-level requirements, architecture, standards (GCD[12]) are sufficiently understood. Normally, this means the relevant software low-level requirements have been documented, placed under developmental configuration control, and are ready for peer review

7.4.9.2.3 Activity Tasks

All the activity tasks listed in section 7.4.9.1.3 will be followed. Manual test cases and procedures are used for the VAPS verification. Verification test cases and procedures include comparing the display of the image on the host and target environment to that in the software low level requirements and then exercising the HMI Components to ensure their behavior is correct. This has been done by loading the graphical BDF files (generated from VAPS layer files and widget library) into the display application simulation and target, and the manual test cases/procedures are used for verification.

7.4.9.2.4 Exit Conditions

This activity is considered complete when the test result data is analyzed and documented in the project Software Verification Procedures and Results document.

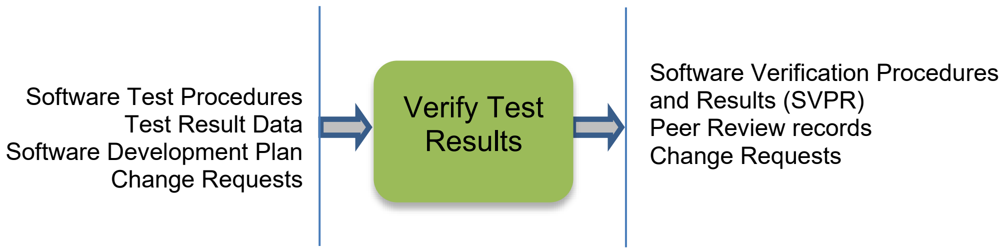
7.4.10 Verify Test Results

7.4.10.1 Overview

The purpose of this activity is to verify the test results produced from software testing process were generated in accordance with the process defined in this SDP and in accordance with the following objectives defined in DO-178B.

A6-1	Executable Object Code complies with high-level requirements
A6-2	Executable Object Code is robust with high-level requirements
A6-3	Executable Object Code complies with low-level requirements
A6-4	Executable Object Code is robust with low-level requirements
A6-5	Executable Object Code is compatible with target computer
A7-2	Test results are correct and discrepancies explained
A7-7	Test coverage of software structure (statement coverage) is achieved (Level C software)
A7-8	Test coverage of software structure (data coupling and control coupling) is achieved (Level C software)

The verification method used for this activity will be the Review method (section 7.3.1) and, for Level C software, the Structural Coverage Analysis method (7.3.2.2).



7.4.10.2 Entry Conditions

Software test result verification may begin once the relevant test procedures have been executed and the corresponding test result data has been collected and placed under developmental configuration control.

7.4.10.3 Activity Tasks

Task 1: Perform Structural Coverage Analysis (Level C software)

Structural coverage data will be collected as a result of performing high level and low-level requirements based testing on a software build that has been "instrumented" by a structural coverage tool . The coverage data collected will be fed back into the structural coverage tool and various reports will be generated depicting which source statements have been covered and which statements have not during the course of performing high level and low-level requirements based testing.

Structural Coverage Analysis (SCA) is the process of analyzing the coverage reports produced by the structural coverage tool, and further analyzing the source statements that were not covered during the course of performing high level and low-level requirements based testing.

For link2k, in some cases special builds called White Box instrumented builds will be created that includes code designed to trigger error conditions for specific sections of code and special behaviors that are difficult to trigger under black box based operations.

SCA will be performed per the specific methods as described in section 7.3.2.2, Structural Coverage Analysis Method.

Based on the analysis performed, CR's may be written to improve structural coverage by:

- Adding test cases and/or test procedures
- Adding high level and/or low-level requirements that result in new test cases and test procedures
- Modifying code to eliminate unnecessary conditional branches
- Modifying code to eliminate dead code

A project should plan for at least two structural coverage runs, such that an initial SCA can be performed to generate CR's as described above, and a final SCA can be performed to document the final results and analyses in the project SVPR after RFS testing is complete.

Task 2: Perform Control and Data Coupling Analysis (Level C software)

The objective of Control and Data Coupling analysis is to confirm that the coupling (data and control) between the code components have been exercised using the requirement-based tests.

- Data Coupling: The dependence of a software component on data not exclusively under the control of that software component.
- Control Coupling: The manner or degree by which one software component influences the execution of another software component.

Link2k

Control coupling analysis will be performed in conjunction with SCA, where statements that were not covered will be analyzed to ensure that no adverse effects on the logical control of execution would occur if the statements were to execute.

Likewise, data coupling analysis will be performed in conjunction with SCA, where statements that were not covered will be analyzed to ensure that no adverse effects on shared data would occur if the statements were to execute.

Other datalink products

Requirements-based tests will be executed in an integrated target environment. The same requirements-based tests will be executed for structural coverage analysis on the host environment. See section above (Task 1 in Section 7.4.10.3) for Structural Coverage Analysis for more details about SCA. The structural coverage analysis results will demonstrate if all functions in the software utilized, and it was called in the proper order. This will verify the coverage of control coupling.

Similarly, SCA results will be used to demonstrate data coupling based on the statement coverage. This will show if all data within those statements were used, which in turn will demonstrate that all data inputs and outputs were exercised by the requirements-based tests or not.

Apart from the coverage analysis, the code review will be performed on the newly added or updated code to identify any deficiency (if any) in the Data and Control Coupling. Change Requests will be written as needed for any identified deficiency(s) in the structural coverage.

Task 3: Record Test Results in SVPR

All actual test procedure results, and corresponding results of the analysis tasks described in this section, will be documented in the project Software Verification Procedures and Results (SVPR) document.

Development of the software verification summary will include identifying the versions of all hardware/software used, test case versions, who performed and witnessed the verification testing, and when the verification testing was performed. The summary data will be contained in the project Software Verification Procedures and Results (SVPR) document. The project SVPR document is associated with a project SVPR Computer Program Configuration Item which is a collection of the electronic copies of test artifacts; test procedure documents, test procedure executable test programs, result files, and results of all analyses performed.

Task 4: Perform Configuration Control

The SVPR is placed under developmental configuration control prior to being peer reviewed as described in section 8.3.1.

Task 5: Conduct Peer Reviews

The test procedure results will be reviewed against the software test procedures to ensure they are correct, that the pass/fail results are accurate and match the expected results, and that discrepancies between actual and expected results are explained. The review for test results will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Test Results Review
- Checklist for Reviewing Test Results

Where analysis was performed as a verification method, the specific analysis will be reviewed to ensure it is correct and that discrepancies between actual and expected results are explained. The software analysis and results will be reviewed to ensure that:

- The analysis was comprehensive and covered all aspects of the area under analysis
- The actual analysis results match the expected analysis results
- The analysis results concur with corresponding test results where appropriate

If the test results include structural coverage data (Level C software), then the structural coverage results will also be reviewed for accuracy and completeness. The test procedure results will be reviewed for correctness with respect to the structural coverage data collected. The review for structural coverage results will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing Structural Coverage Results Review
- Checklist for Reviewing Structural Coverage Results

When the individual test result reviews are completed and all results have been captured in the SVPR, a final peer review will be conducted as described in section 7.3.1 using the following checklists:

- Checklist for Producing SVPR Review
- Checklist for Reviewing SVPR

Task 6: Perform Change Control

The SVPR is subject to change control mechanisms as defined in section 8.3.4.

7.4.10.4 Exit Conditions

This activity is considered complete when:

1. The test result data is analyzed and documented in the project Software Verification Procedures and Results document.
2. All findings in the peer review(s) have been closed and each peer review itself has been closed.

7.5 Verification Environment

The verification environment includes the use of both a host/simulator and the target hardware. See section 7.4.5.3 for a description of host and target test methods and section 4.3 for a full description of the verification environment. The test procedure identifies the test execution environment. When test procedures are executed on the host for structural coverage purposes, any differences in results from target execution of that same test procedure will be handled via the CR/CCB process.

7.5.1 Tool Qualification

Tools that must be qualified are those used to reduce, eliminate, or automate a software life–cycle process and that tool's output is not verified as part of the project software verification process. Two classifications of tools exist, software development tools and software verification tools. Software development tools provide output that becomes part of the airborne software and can introduce errors. Software verification tools do not introduce errors, but may fail to detect errors.

Any qualified software development tool(s) and/or verification tool(s) will be documented in respective qualification plans, operational requirements, verification results, and accomplishment summary. In addition, any qualified software development tools will have tool qualification development data (design, traceability, code, etc.).

Refer to the project PSAC for a description of any planned tool qualification activities.

7.6 Transition Criteria

The transition criteria for entering the verification activities are described in the entry condition sections throughout the verification processes documented in section 7.4.

7.7 Partitioning

Reference the project-specific PSAC for a description of partitioning and if/how it is used within the project software product.

In general, Data Link software products developed for IMA-based systems do not themselves use partitioning, as this is a function of the underlying platform software.

Partitioning is used, however, on certain software products used in the CMU-900, CMU-4000, and RIU-40X0. The hardware partitioning is accomplished by means of the protected mode capabilities built into the 486 processor. The 486 processor provides the following mechanisms:

- Application specific Local Descriptor Table (LDTs) provide bounded access to code and data
- Through the use of 486 privilege levels (0-3), tight control of inter-partition access to memory and instructions is achieved.

In addition, the software monitoring protection mechanisms employed within System Services are to prevent adverse effects from erroneous operation through the following:

- Power on self-test
- Software Watchdog
- Protected Mode CPDLC – Only for products that contain PM CPDLC
- System Data Privileged Access

If partitioning is used, the partition design will be fully documented in the software architecture, including the means for providing integrity among the partitions and the prevention of software breaches. The peer review process will be used to verify the software architecture to ensure that software partition integrity is confirmed, and requirements-based test procedures will be developed to verify the protection mechanisms in the target hardware.

7.8 Compiler Assumption

The verification of high-level and low-level requirements and their associated outputs encompasses the verification of the compiler correctness. Compiler and linker options will be selected early in the initial integration phases of a new software product, and are not anticipated to change throughout the remaining lifecycle of that product. If any changes do occur with the compiler and linker options such that the executable image is modified, then re-verification of the new executable will be required.

The compiler and linker options will be documented in the project specific Software Environment Configuration Index (SECI) document and maintained under development configuration control.

The external loader is expected to be ARINC 615-3 compliant. The internal loading software within the unit is composed of L2 software which resets the hardware to a known state and validates the software load by performing a CRC check of all the loadable software components. If no errors are detected the normal startup procedures are allowed to continue to execute to full operational status.

7.9 Re-verification

Re-verification is the process of performing the software verification activities against a subset of a previously verified software product, where the artifacts associated with that subset have been modified as result of one or more Change Requests.

When a Change Request results in the modification of a software artifact, traceability will be used to determine the impact analysis on all affected artifacts, both higher level and lower level. If the impact analysis results in related changes to either the source code or the test procedures, the relevant test procedures will be rerun against the software product and new test result data will be collected.

7.10 Previously Developed Software

Refer to the project specific PSAC for the description and the applicability of the Previously Developed Software.

7.11 Multiple-Version Dissimilar Software

No multiple-version dissimilar software is used in any Data Link software product.

8 Software Configuration Management

This section describes the objectives and activities of the Software Configuration Management (SCM) process for the Data Link projects regarding *developmental* configuration management. For information regarding formal *production release* configuration management, reference the Rockwell Collins *Software Configuration Management Plan* (SCMP) [6].

8.1 Responsibilities

In general, all project personnel that contribute to the development of formal project artifacts are responsible for configuration management of those artifacts. In this context, a formal project artifact is one that is a deliverable to a consumer outside the Data Link organization, and/or is a required artifact as defined by the processes in this SDP.

The Software Control Library (SCL) provides production release configuration management as described in the *Software Configuration Management Plan* [6].

DAC representatives are responsible for auditing the SCM process.

8.2 Environment

Rational ClearCase and/or Subversion will be used to maintain developmental configuration control of all project artifacts, except for DOORS modules, which are configuration controlled within the DOORS tool itself. See *DOORS Documentation Method for the Commercial Systems Data Link Organization* [2].

The change tracking tool, JIRA, provides the capability to record and track problem reports (work packages) subject to configuration management, including software life cycle processes. JIRA is a purchased tool and can be used for scheduling and tracking change requests (problem reports). The change tracking tool database is an on-line database designed to collect data and track the progress of product changes. A product change will be captured as a Work Package (WP). A Work Element (WE) will be used to track and document the product change to artifacts under configuration management. WEs will be linked to WPs that drive the product change within the tool. The change tracking tool will enforce valid state transitions. It will also force required data fields to be completed for each WP based on its current state. For a detail description of JIRA fields and states, see *Change Request and CCB Process for the Commercial Systems Data Link Organization Using JIRA* [5].

The change tracking tool, ClearQuest, is an enterprise change tracking tool and provides equivalent capability as JIRA but uses Change Requests (CRs) and Change Tasks (CTs) rather than Work Packages and Work Elements.

Rational ClearQuest and/or JIRA will be used to record and control additions and changes to project artifacts through the use of Work Packages or Change Requests (CRs).

For simplification purposes, the term ‘Change Request’ or ‘CR’ will be used generically to describe the change tracking mechanism for the change tracking tools. For ClearQuest, see *Change Request and CCB Method Using ClearQuest for the Commercial Systems Data Link Organization* [4]. For JIRA, *see Change Request and CCB Process for the Commercial Systems Data Link Organization Using JIRA* [5].

PREP is the Rockwell Collins enterprise peer review tool. PREP is utilized to conduct peer reviews on life cycle data produced by this development plan. See section 7.3.1 Review Method.

8.3 Activities

8.3.1 Configuration Identification

During the development process, all formal project artifacts will be maintained under configuration control. Any high level and low-level requirements, specification, or other materials used during the development process that are not under another configuration control system will be maintained under project configuration control. This ensures the capability to reconstruct any version of a document or software file for any stage of the development process.

All verification activities will be performed against material that has been placed under configuration control. All verification artifacts and tools that are used to perform formal verification activities will be under configuration control prior to starting formal verification activities.

For each Black Label release all components necessary to reconstruct the version must be archived in the development configuration control system. These items will be released through the production Software Control Library (SCL) to support Black Label delivery. The project will ensure that the following items are under configuration control:

- All documents which place requirements on the deliverables. This includes any documents referenced by the SRS and SDD as well as the SRS and SDD itself. It may also include any plans or standards, which it references that affect the deliverable or the software development process.
- The source code for all deliverables. Object code or other intermediate forms between the source and the load module will not be under engineering configuration control since they can be recreated from the source code.
- Executable images for software under test
- All test cases, procedures, and results.
- All software or other tools required to maintain the deliverables.
- As requested by the certification office, any additional software documentation or artifacts in support of certification approval.

For DOORS modules a baseline can be used to identify a specific revision of a module. See the *DOORS Documentation Method for the Commercial Systems Data Link Organization* [2] for additional information on DOORS baselines.

Source control tools provide the mechanism of applying a version label to identify all of the components of a specific software build. The version label is a unique string that can be associated with a unique version number for each controlled archive file. Once the target version of each component in the software build has been “tagged” with a version label, source control operations can use the label to globally manipulate all of the components of the build.

8.3.2 Baselines and Traceabiity

Reference the Glossary in Appendix C for a definition of the term “baseline”.

Once an artifact has been baselined, it can only be modified through a Change Request as described in 8.3.3.

Note that the term “baseline” is merely a definition applied to the maturity level of an artifact, with respect to the review and approval of that artifact. It should not be misconstrued with like terms used to represent the physical state of an artifact within a particular tool. For example, a

DOORS baseline may or may not relate to the same definition of a baselined artifact as defined in this section.

Traceability to baselined items will be managed in the PREP peer review tool, where artifact version identification is captured for each reviewed item as described in the *Peer Review Process Using PREP for the Commercial System Data Link Organization [3]*.

When comparing baselines of a given artifact, consideration should be given to the capability within the tools to facilitate easier baseline comparisons. For example, comparing source code baselines is often accomplished with difference listings produced by the source code control system or an external utility compare tool such as Beyond Compare. Another example for Microsoft Word document comparisons is to use the change tracking mechanism within Microsoft Word itself.

Traceability between baselined items maintained in the project configuration management system (e.g. Subversion repository) and the enterprise configuration management system (i.e. SCL) is accomplished through the use of labels, such that version labels containing a text string representation of the formal release identifier used in the enterprise configuration management system (e.g. part number) is applied to the corresponding items maintained in the project configuration management system.

8.3.3 Change Requests/Problem Reporting

The Rational ClearQuest and/or JIRA tool will be used to record and track Change Requests for any item subject to configuration management. Problem Reporting will be managed by creating Change Requests. Refer to the *Change Request and CCB Method Using ClearQuest for the Commercial Systems Data Link Organization* [4] or *Change Request and CCB Process for the Commercial Systems Data Link Organization Using JIRA* [5] for details on generating and tracking Change Requests.

All software project personnel will have access to ClearQuest and/or JIRA. Change Requests are entered as they are discovered throughout the product’s life cycle. Change requests are controlled and closed by the project Change Control Board (CCB) as described in 8.3.4.

8.3.4 Change Control

A developmental Configuration Control Board (CCB) is responsible for managing all submitted Change Requests throughout a product’s life cycle. Change Requests may be the result of a discovered error (problem report), or an enhancement request for the product. The CCB is responsible for managing Change Requests based on the project cost, schedule, and scope. For Change Requests that are accepted, the CCB assigns each one to a team member for action and establishes criteria such as criticality, priority, and desired completion dates. Refer to the *Change Request and CCB Method Using ClearQuest for the Commercial Systems Data Link Organization* [4] for details on the CCB methods. While the CCB Method refers specifically to the ClearQuest tool and Change Request (CR) states, the JIRA tool, when used, will follow the same processes and intent but the use of the tool and Work Package (WP) states will differ. For specifics on the use of JIRA see *Change Request and CCB Process for the Commercial Systems Data Link Organization Using JIRA* [5].

Artifacts which have been released through the SCL will only be changed through another release resulting in a part number version or revision change and are controlled as defined in the *SCMP* [6].

8.3.5 Change Review

The Change Review activity will utilize the Change Request process, described in section 8.3.3, for handling feedback changes as a result of the life cycle processes. These Change Requests will be managed by the development Configuration Control Board (CCB) as described in section 8.3.4.

8.3.6 Configuration Status Accounting

Configuration reporting is available to all project personnel and is used to identify and help control product revisions throughout the development and verification cycles.

Reports can be obtained from the project configuration management system to track and status configuration items and baselines. Additionally, reports are available from the change tracking tool to track and status problem reports and change history. And finally, a project specific database is used to track and status the formal release of items submitted to the enterprise configuration management system.

8.3.7 Archive, Retrieval, and Release

The SCL ensures that the software life cycle data associated with the software product is archived and can be retrieved. Only software released through the SCL is used in manufacturing products. The processes used by the SCL are defined in the *SCMP* [6].

8.3.8 Software Load Control

Software products associated with the federated hardware (CMU-900, CMU-4000, RIU-40X0) utilize a data loader function to reprogram those units in accordance with ARINC 615. Software products associated with IMA hardware (DLCA) will be loaded into the IPC as part of a systems media set. Additional information about software load control safeguards can be found in the project specific PSAC.

8.3.9 Software Life Cycle Environment Controls

The tools used to develop, build, verify, and load the software are required to be released through SCL according to the *Software Control Library Release Procedure* [7]. This is considered essential to product development, verification, or configuration control and is managed throughout the life cycle of the product. Section 4.3 provides a consolidated list of the tools for the project.

8.3.10 Software Life Cycle Data Controls

The Project Engineer is responsible for determining the appropriate control category based on the software level and data item.

All outputs categorized as Control Category 1 (CC1) in DO-178B will be released through SCL.

Data items generated at the program/project/product or corporate level, such as CR/PR database, peer review records, SQE records, Software Configuration Management (SCM) records, etc. are generally categorized as Control Category 2 (CC2) as described in DO-178B. All outputs categorized as CC2 will be maintained by their cognizant organization using enterprise backup/data integrity processes for tools that do not allow for simple archival of their data. Note: The cognizant organization may, at their discretion, release these artifacts through the SCL to CC1 criteria.

All electronic outputs from the development and verification processes will be maintained as uniquely identified items via an approved configuration management system providing traceability and protection against unauthorized changes, using problem reporting, change control, and change review procedures.

8.4 Transition Criteria

The entry condition for the software configuration management process is the start of the project. Upon project start–up, development tools are selected and project repositories are established.

Data Items must be placed under development configuration control for the item to be considered baselined.

The criterion for entering the activity to manage change to a baselined item is to enter a CR.

8.5 Software Configuration Management Data

Data items submitted to the SCL must meet requirements established in the *Software Control Library Release Procedure* [7]. The data items required to be controlled as *DO-178B* [17] CC1 are part of the release package submitted to the SCL.

All project CM records, including developmental CM version history records, and change requests/problem reports, are maintained under developmental configuration control (CC2) electronically within the development environment.

8.6 Supplier Control

Software from suppliers that is incorporated into a product is subject to the same software configuration management practices defined in this document.

9 Software Quality Assurance

The *Commercial Systems Software Quality Engineering Assurance Plan* [7] defines the roles and responsibilities of the Design Assurance Center for Quality and the processes that the DAC representatives follow. The Software Quality Plan applies to the Data Link development projects and defines the methods to be used to achieve the objectives of the SQE process. The CS SQE Plan includes descriptions of the following:

- Environment
- Authority
- Activities
- Transition Criteria
- Timing
- SQE Records
- Supplier Control

10 Software Deployment

This section describes the typical artifacts that are generated in relation to a formal release of the software product. These artifacts include the Software Accomplishment Summary (SAS), Version Design Description, Software Emod, and a Footprint document. Reference the sections below for applicability requirements against the specific Data Link product.

10.1 Software Accomplishment Summary

A Software Accomplishment Summary (SAS) will be created in accordance with the guidelines defined in *DO-178B* [17] for all Data Link software products governed by this SDP.

As part of the software characteristics documented in the SAS, timing margins will be obtained as described in section 7.3.2.4 and a stack analysis will be performed as described in section 7.3.2.5.

The SAS will be placed under developmental configuration control as described in section 8.3.1 and subject to change control as described in section 8.3.4.

The SAS will be reviewed per the method described in section 7.3.1 using the following checklists:

- Checklist for Producing SAS
- Checklist for Reviewing SAS

Upon completion of the peer review and all findings resolved, the SAS will be formally released to the SCL as described in section 8.3.1.

10.2 Version Design Description

The Version Design Description (VDD) document captures the software product identification (version) information along with a description of the changes for the identified version. The VDD is required for each formally released version of all Data Link software products governed by this SDP. While there is no specific template required for the VDD, the document should include the software identification and change history information as described in the SAS (reference *DO-178B*[17] section 11.20, items h and i). The Data Link software development team is responsible for creating the VDD document.

10.3 Footprint Document

The footprint document is created for IMA applications to characterize the software footprint in relation to the available resources in the IMA computing environment. This document is not required for federated LRU software products. Through the incremental series of application integration processes, a footprint document that defines the processing resource parameters for the software application in the IMA will be captured and formalized at project completion. The Data Link software development team is responsible for creating the Footprint document in accordance with the *Integrated Modular Avionics (IMA) Footprint Process Document [9].*

10.4 Emod Document

The Engineering Modification (Emod) document is a Rockwell Collins internal document intended to provide a flexible method for configuration control and tracking of design changes, from a baseline configuration to final product configuration, in order to establish and maintain company conformity for a product. The Emod document will be generated for BRS Data Link products to be delivered external to the Data Link organization. The Data Link software development team is responsible for creating the Emod document in accordance with the *Configuration Management Plan for Developmental Avionics Equipment (Emod Procedure) [10]*.

Appendix A Data Link Software Products

Table 10-1 below lists the Data Link software products, along with the associated target equipment types for each product, for which this SDP is intended.

Table 10-1 Data Link Software Products

	DAL	CMU-900	CMU-4000	RIU-40X0	Fusion IMA	EDS IMA	GS IMA	3rd Party IMA
Core SW Products								
DLNKX1 (Link2000+)	C,D	●	●	●				
DLNKX2 (SC-214)	C,D	●						
DLNKX3 (No Local Apps)	D		●	●				
AOC SW Products								
EDS AOC	D					●		
VM SW Products								
VM Data	D	●	●	●				
IMA SW Products								
DLCA-6000	C				●		●	
DLCA-6500	C				●	●		
DLCA-6510	C					●		
CSA-3500	D					●		●
ATS-3500	C							●

Appendix B Correlation to DO-178B

The following is a correlation between RTCA/DO-178B and the applicable section in this SDP or reference.

DO-178B Section and Title		SDP Section or Reference
11.2	Software Development Plan	This document
11.2a	Standards	4.2
11.2b	Software Life Cycle	5,6,7
11.2c	Software Development Environment	4.3
11.3	Software Verification Plan	7
11.3a	Organization	7.1
11.3b	Independence	7.2
11.3c	Verification Methods	7.3
11.3d	Verification Environment	7.5
11.3e	Transition Criteria	7.6
11.3f	Partitioning Considerations	7.7
11.3g	Compiler Assumptions	7.8
11.3h	Re-Verification Guidelines	7.9
11.3i	Previously Developed Software	7.10
11.3j	Multiple-Version Dissimilar Software	7.11
11.4	Software Configuration Management Plan	8
11.4a	Environment	8.2
11.4b1	Configuration Identification	8.3.1
11.4b2	Baselines and Traceability	8.3.2
11.4b3	Problem Reporting	8.3.3
11.4b4	Change Control	8.3.4
11.4b5	Change Review	8.3.5
11.4b6	Configuration Status Accounting	8.3.6
11.4b7	Archive, Retrieval, and Release	8.3.7
11.4b8	Software Load Control	8.3.8
11.4b9	Software Life Cycle Environment Controls	8.3.9
11.4b10	Software Life Cycle Data Controls	8.3.10
11.4c	Transition Criteria	8.4
11.4d	SCM Data	8.5
11.4e	Supplier Control	8.6
11.5	Software Quality Assurance Plan	9
11.6	Software Requirements Standards	4.2.1
11.6a	Requirement Methods	4.2.1
11.6b	Requirement Notations	4.2.1
11.6c	Requirement Tool Constraints	4.2.1
11.6d	Derived Methods	4.2.1
11.7	Software Design Standards	4.2.2
11.7a	Design Methods	4.2.2
11.7b	Naming Conventions	4.2.2
11.7c	Conditions on Design Methods	4.2.2
11.7d	Design Tool Constraints	4.2.2
11.7e	Design Constraints	4.2.2
11.7f	Complexity Restrictions	4.2.2
11.8	Software Code Language	4.2.3
11.8a	Programming Language	4.2.3
11.8b	Source Code Presentation	4.2.3
11.8c	Naming Conventions	4.2.3
11.8d	Source Code Constraints	4.2.3
11.8e	Source Code Tool Constraints	4.2.3

Appendix C Glossary

The terms defined here are intended to be consistently applied throughout this document. Where these terms are also used in DO-178B, the *italicized* text is taken straight from the definition provided in DO-178B, and then expanded upon to provide further clarification as it relates to the processes described in this SDP.

Algorithm – *DO-178B: A finite set of well-defined rules that give a sequence of operations for performing a specific task.* This definition is further clarified as a specific set of ordered computations for producing a well-defined output based on a set of inputs, often corresponding to mathematical formulas (e.g. a CRC calculation or a set of encoding/decoding rules). In this context, an algorithm is not meant to represent general program flow or pseudo code descriptions.

Baseline – *DO-178B*: *The approved, recorded configuration of one or more configuration items, that thereafter serves as the basis for further development, and that is changed only through change control procedures*. The term “basis for further development” implies two cases for declaring an artifact as baselined:

1. The artifact itself is being further developed. In this case, if the previous version was reviewed and approved, then the previous version is considered “baselined” by definition, such that the subsequent review need only focus on the changes.
2. The artifact is being used as an input to produce another artifact. In this case, the output artifact cannot be reviewed and approved until the input artifact has been reviewed and approved; hence, once the input artifact is reviewed and approved, it is, by definition, considered “baselined”.

Level 1 (L1) Requirements – For Data Link projects described in this SDP, L1 refers to requirements at the aircraft system level, which is external to the Data Link organization. For Air Transport programs, this usually refers to OEM aircraft requirements. For BRS programs, this usually refers to the Rockwell Collins platform system engineering organization responsible for the BRS aircraft platform.

Level 2 (L2) Requirements – For Data Link projects described in this SDP, L2 refers to requirements at the Data Link subsystem engineering level, which is internal to the Data Link organization. L2 requirements are typically generated for Data Link LRU products that involve both HW and SW (e.g. CMU-900).

Level 3 (L3) Requirements – For Data Link projects described in this SDP, L3 refers to high-level software requirements at the Data Link software engineering level, which is internal to the Data Link organization. L3 requirements are generated for all Data Link software products.

Level 4 (L4) Requirements – For Data Link projects described in this SDP, L4 refers to low-level software requirements at the Data Link software engineering level, which is internal to the Data Link organization. L4 requirements are generated for all Data Link software products that are DO-178B Level D or higher.

Relevant – Throughout this document, the term relevant is used to denote a proper subset of an artifact that is used or referenced in completing a process activity. Because the life cycle processes can be iterated upon, there will be partially completed artifacts and/or extraneous content within the artifact that does not pertain to the specific activity. In this context, the relative content includes the full subset directly applicable and pertaining to the specific activity.

Reviewed and Approved – Denotes a particular revision of an artifact for which a peer review has been successfully completed per the method defined in section 7.3.1. The artifact is considered approved by virtue of achieving acceptance from the peer review process. This term is synonymous with “Baseline”.

Small Scope CR – A CR for which the resulting change to all impacted life cycle artifacts is very limited, such that it is practical to combine multiple artifacts into a single review. An example is changing a timer value in a system requirements document, where it makes practical sense to review all affected artifacts concurrently. If there is uncertainty as to what is considered “practical”, the PE has authority to make the determination.

Validation – *DO-178B: The process of determining that the requirements are the correct requirements and that they are complete. The system life cycle process may use software requirements and derived requirements in system validation.* The typical method used in validation is testing. The objective of validation can be summed up asking the following question:
Are we building the *right* product?
Verification – *DO-178B: The evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process.* Verification is then much more than just testing. It is an integral process that includes activities performed concurrently with the development processes (as evidenced by the verification processes in Figure 3-1 Life Cycle Model). Typical methods employed are reviews, analyses, and inspection. The objective of verification can be summed up asking the following questions:
Are we building the product *right*?
Are we building the product according to the specified process?

Appendix D Acronyms and Abbreviations

Acronym	Definition
A661	ARINC 661
AAR	Airtel ATN Router
AFD	Advanced Flight Display
AGS	A661 Graphical Server
APL	Application Programming Language
ARINC	Aeronautical Radio, Inc.
ASM	Assembly
ATC	Air Traffic Control
ATN	Aeronautical Telecommunications Network
BDF	Binary Definition File
CCB	Change Control Board
CMU	Communications Management Unit
CNS/ATM	Communication, Navigation, Surveillance/Air Traffic Management
CPCI	Computer Program Configuration Item
CR	Change Request
CSCI	Computer Software Configuration Item
DAC	Design Assurance Center
DAL	Design Assurance Level
DLCA	Data Link Communications Application
EPA	Engineering Project Assistant
GCD	GUI Conventions Description
GUI	Graphical User Interface
HW	Hardware
IMA	Integrated Modular Avionics.
I/O	Input/Output
IOCF	Input Output Common Format
LRU	Line Replaceable Unit
LTC	Lead Technical Contact
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
PE	Project Engineer
PSAC	Plan for Software Aspects of Certification
RCPN	Rockwell Collins Part Number
RFS	Run For Score
RIU	Radio Interface Unit
RTCA	Radio Technical Commission for Aeronautics

Acronym	Definition
SARD	Systems Architecture Requirement Document
SAS	Software Accomplishment Summary
SCA	Structural Coverage Analysis
SCL	Software Control Library
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SDD	Software Design Document
SDP	Software Development Plan
SME	Subject Matter Expert
SQE	Software Quality Engineering
SRS	Software Requirements Specification
SVP	Software Verification Plan
SVPR	Software Verification Procedures and Results
SW	Software
TCP	Technical Consistent Process
TDF	Textual Definition File
TPM	Technical Project Manager
TPR	Technical Performance Review
VAPS	Virtual Application Prototyping System