RAPITA SYSTEMS

A DANLAW COMPANY

Safety through quality

HANDBOOK

Efficient verification through the DO-178C life cycle

Contents

1. Introduction to DO-178C

The safety assessment processes used in all functional safety domains rely on demonstrating that the probability of system failure that could cause harm is below an acceptable threshold.

When a system is made up of mechanical and electronic components, for which the component failure rate is known, the probability of failure for the system can be calculated and achievement of the safety target can be demonstrated. For software, complex systems or electronic hardware, system failures can be caused by design errors (sometimes known as systematic failures) as well as component failures, but there is no agreed way of calculating the failure rate of these design errors. In the aerospace domain, the agreed approach for dealing with design errors is to implement design assurance processes that have specific activities to identify and eliminate design errors throughout the software development life cycle.

DO-178 was originally developed in the late 1970s and released in 1982 to define a prescriptive set of design assurance processes for airborne software that focused on documentation and testing. In the 1980s, DO-178 was updated to DO-178A, which suggested different levels of activities dependent on the criticality of the software, but the process remained prescriptive. Released in 1992, DO-178B was a total re-write of DO-178 to move away from the prescriptive process approach and define a set of activities and associated objectives that a design assurance process must meet. This update allowed flexibility in the development approaches that could be followed, but also specified fundamental attributes that a design assurance process must have, which were derived from airworthiness regulations. These included, for example, demonstrating implementation of intended function, identifying potential unintended function, and verification of an integrated build running on the target hardware.

Advances in software engineering technologies and methodologies since the release of DO-178B made consistent application of the DO-178 objectives difficult. In 2012, DO-178C was released, which clarified details and removed inconsistencies from DO-178B, and which also includes supplements that provide guidance for design assurance when specific technologies are used, supporting a more consistent approach to compliance for software developers using these technologies. DO-178C guidance also clarified some details within DO-178B so that the original intent could be better understood and more accurately met by developers.

1.1 Design Assurance Levels (DALs)

DO-178B introduced (and DO-178C continued to use) the fundamental concept of the Design Assurance Level (DAL), which defines the amount of rigor that should be applied by the design assurance process based on the contribution to Aircraft Safety. The higher the DAL, the more activities and objectives that must be performed and met as part of the Design Assurance process because of the more severe consequences to the aircraft should the software fail or malfunction. Design Assurance Level A (DAL-A) is the highest level of design assurance that can be applied to airborne software and is applied when failure or malfunction of the software could contribute to a catastrophic failure of the aircraft. The activities and objectives that must be met through the Design Assurance process gradually decrease with each level alphabetically until DAL-E, which has no objectives as there is no consequence to aircraft safety should such software fail or malfunction.

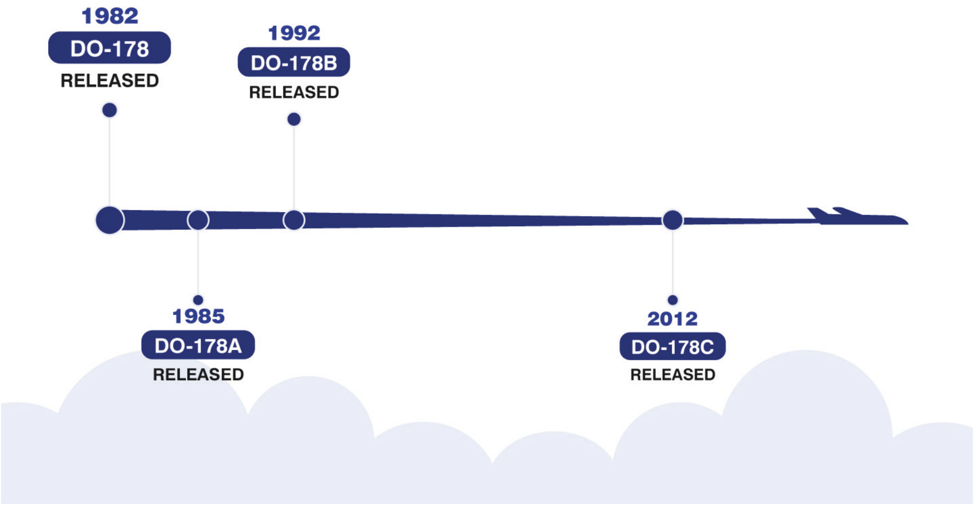


Figure 1 – Timeline of release of DO-178 guidance

1.2 Objectives and activities

The recommendations given in DO-178 fall into two types:

- Objectives, which are process requirements that should be met in order to demonstrate compliance to regulations
- Activities, which are tasks that provide the means of meeting objectives

In total, DO-178C includes 71 objectives, 43 of which are related to verification. The number of these objectives that must be met for compliance reduces as the Design Assurance Level of the system reduces (Figure 2).

DO-330

DO-330 provides tool-specific guidance for developing airborne and ground-based software. As DO-330 can be used independently, it is not considered as a supplement to DO-178C and is therefore titled differently from DO-178C's specialized technology supplements e.g. DO-331, DO-332 and DO-333.

1.3 Supplementary objectives and guidance

DO-178C introduced three technology supplements to provide an interpretation of the DO-178C activities and objectives in the context of using specific technologies. The three technologies are Model Based Development and Verification (DO-331), Object Oriented Technology and related technologies (DO-332), and Formal Methods (DO-333). Each supplement describes the technology, defines the scope of its use within airborne software, lists additional or alternative activities and objectives that must be met when the technology is used, and includes specific FAQs (Frequently Asked Questions) that clarify objectives and activities relating to the technology.

A further supplement was introduced in DO-178C, *Software Tool Qualification Considerations (DO-330)*, which gives guidance on the qualification of tools used in software development and verification processes. This guidance can be applied to any tools, not just those used for software development or verification, for example systems design or hardware development tools, and acts more like a stand-alone guidance document than the other supplements mentioned.

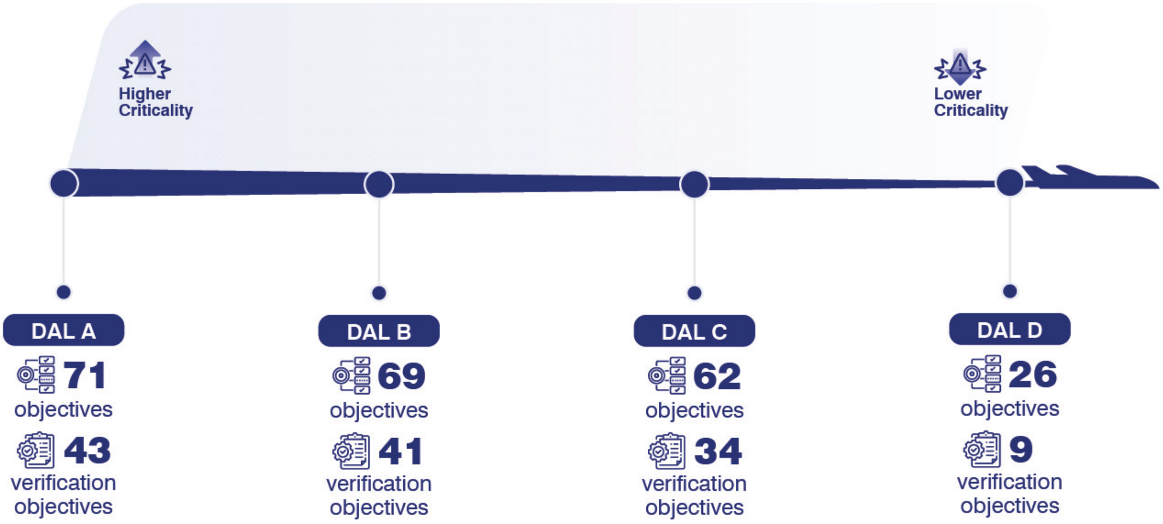


Figure 2 – Number of objectives and verification objectives in DO-178C Design Assurance Levels

Certification Authorities Software Team
(CAST)

Position Paper
CAST-32A

Multi-core Processors

COMPLETED November 2016 (Rev 0)

NOTE: **This position paper has been coordinated among representatives from certification authorities in North and South America, Europe, and Asia. However, it does not constitute official policy or guidance from any of the authorities. This document is provided for educational and informational purposes only and should be discussed with the appropriate certification authority when considering for actual projects.**

Figure 3 – Supplementary documents such as CAST-32A have provided additional clarification or explanations for DO-178C

Many other documents support DO-178C by providing additional clarification or explanations that can help developers to correctly interpret the guidance and implement appropriate design assurance processes. The Supporting Information (DO-248C) supplementary document includes FAQs relating to DO-178C, and the document is commonly referred to by the title Frequently Asked Questions. In addition to the FAQs in DO-248C, the document provides the rationale for the activities and objectives listed in DO-178C and includes discussion papers that provide clarification on specific topics related to software development and verification. A series of documents produced by the Certification Authorities Software Team (CAST) since the release of DO-178B provided information on specific topics of concern to certification authorities in order to harmonize approaches to compliance. These topics have had a greater scope than just Software concerns, and much of the content in CAST documents has been implemented in guidance updates such as DO-178C, or formed the basis of authority publications, such as A(M)C[1] 20-193 to address the use of multicore processors in avionics and A(M)C 20-152A on the development of airborne electronics hardware. CAST has remained inactive since October 2016 and links to most previous CAST papers have been removed from the FAA's website. The FAA plans to remove the link to CAST-32A after the publication of A(M)C 20-193.

The CAST team

The Certification Authorities Software Team (CAST) is a team of software specialists from certification authorities in the United States, Europe and Canada. The team released a number of documents that provide supplementary compliance guidance since the release of DO-178B, but the team has remained inactive since October 2016.

1.4 Demonstrating compliance

The basic structure of a Design Assurance process consists of three components:

1. Planning
2. Development
3. Integral processes (*Verification, Configuration Management, Quality Assurance,* and *Certification Liaison*)

[1] A(M)C refers to either EASA AMC (Acceptable Means of Compliance) or FAA AC (Advisory Circular) documents.

Planning should occur first and this follows the basic design assurance principle that you say what you are going to do before you do it so you can ensure that what you plan to do will meet the required DO-178C objectives and provide evidence to demonstrate this.

The **Development** process comprises the engineering activities needed to incrementally understand and decompose the system requirements into a software design and implementation that can be executed.

The **Integral** processes cover the activities needed to ensure that the products of the Planning and Development processes meet their requirements and are well managed, and that the processes you follow match those you planned to follow. The Certification Liaison activities focus on ensuring that there are sufficient data and checkpoints for the certification authorities to be able to determine compliance and approve the software.

The typical process for the certification authority to determine compliance is based on four Stage Of Involvement (SOI) reviews. These reviews are:

1. **SOI#1** or **Planning Review**
2. **SOI#2** or **Development Review**
3. **SOI#3** or **Verification Review**
4. **SOI#4** or **Certification review**

Each of these reviews focuses on an aspect of the process and evaluates the evidence that demonstrates compliance incrementally throughout the development life cycle. We discuss each of the SOIs in more detail later. Generally, certification authorities require that each SOI is passed before a project can proceed to the next SOI. SOIs thus mark key milestones in a DO-178C project.

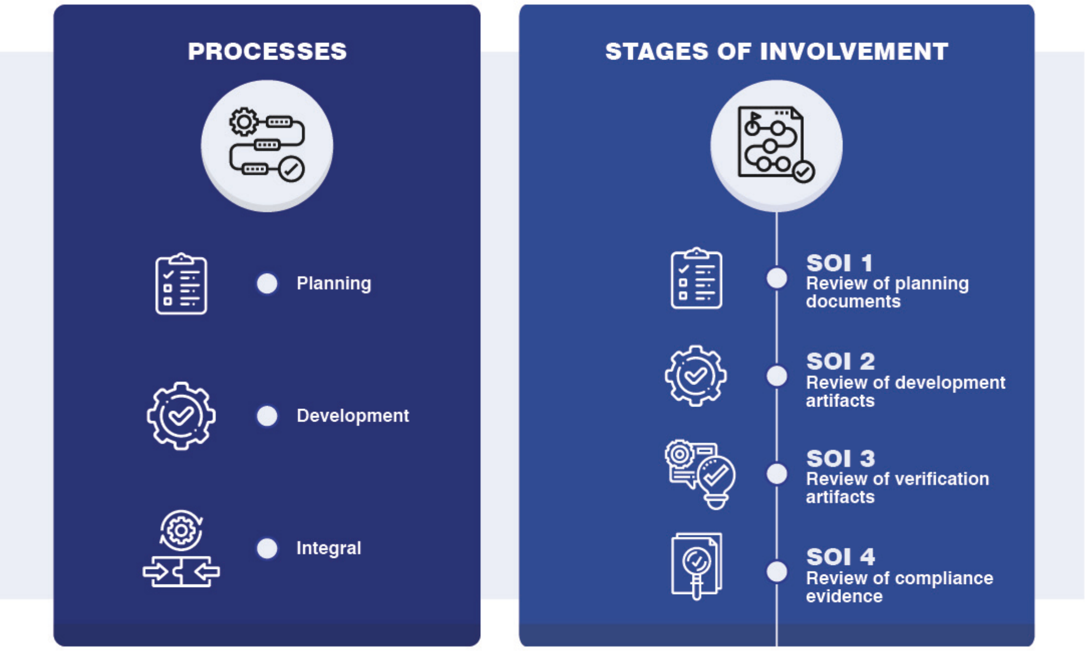
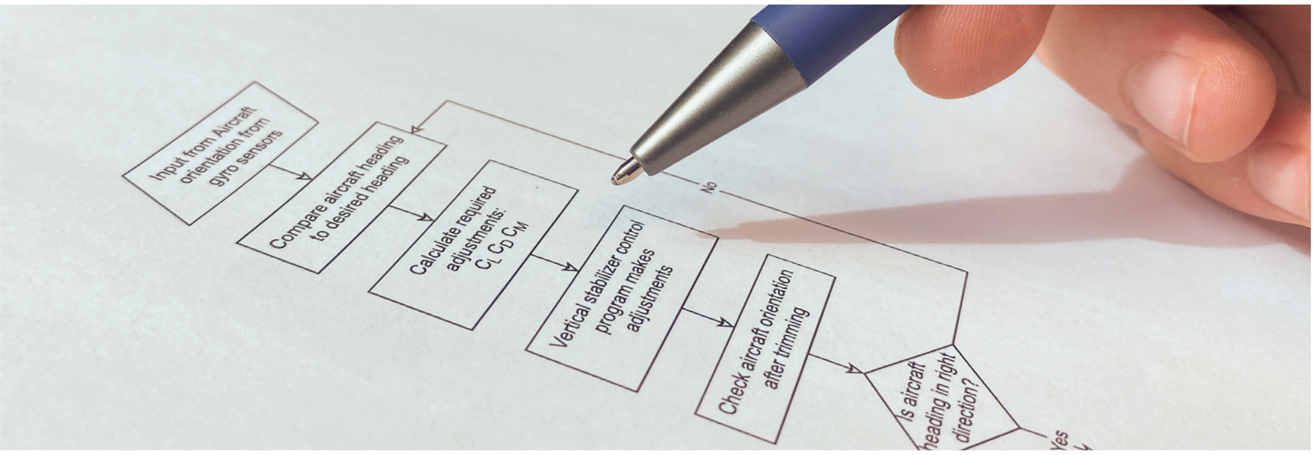

PROCESSES
Planning
Development
Integral
STAGES OF INVOLVEMENT
SOI 1
Review of planning documents
SOI 2
Review of development artifacts
SOI 3
Review of verification artifacts
SOI 4
Review of compliance evidence

Figure 4 – DO-178C processes and stages of involvement

2. Planning


Input from Aircraft orientation from gyro sensors
Compare aircraft heading to desired heading
Calculate required adjustments: C_L C_D C_M
Vertical stabilizer control program makes adjustments
Check aircraft orientation after trimming
Is aircraft heading in right direction?
No
Yes

TIP

Planning is a crucial part of DO-178C compliance and it sets the scene for your project's activities and the efficiency of those activities in later stages. You can save effort in your DO-178C planning by using a set of template planning documents as a starting point.

The development of a set of plans covering all components of the Design Assurance process is a cornerstone of DO-178C. As part of this activity, the following plans must be developed:

1. **Plan for Software Aspects of Certification** (**PSAC**): a description of the software you plan to develop, the hardware environment it will be used in, the design assurance processes you will follow, and how you will demonstrate compliance, including how you will verify your implemented code and any commercial tools you will use in your verification. This acts as the parent planning document.

2. **Software Development Plan** (**SDP**): a description of the software development processes and the software life cycle that is used to satisfy DO-178C objectives.

3. **Software Verification Plan** (**SVP**): a description of the verification processes (Reviews, Analyses and Tests) used to satisfy DO-178C objectives.

4. **Software Configuration Management Plan** (**SCMP**): a description of the methods and environment that will be used to configure all of the design data and compliance evidence needed to achieve DO-178C certification.

5. **Software Quality Assurance Plan** (**SQAP**): a description of the methods and associated records that will be used to ensure that DO-178C quality assurance objectives are satisfied.

Importance of the PSAC

The Plan for Software Aspects of Certification (PSAC) is especially important and is essentially a compliance contract between your specific project and your certification authority. It must explain all features of the software and the associated processes that are salient to demonstrating compliance to the objectives of DO-178C and any applicable supplements.

Reducing planning costs

Creating plans and standards for a project working towards DO-178C compliance requires a lot of effort, but you can save effort by using a complete set of templates as a starting point. Some companies offer such templates for a fee.

For DALs C and higher, as well as producing the above planning documents, you will also need to establish and document the standards you will use for Requirements, Design, and Code, documenting the methods, rules, and tools you will use to ensure complete and consistent generation of these items in a way that will meet DO-178C and project objectives. It pays to put effort into writing your standards – having well thought out standards reduces the chance of incurring rework costs later due to issues that could be addressed through having better standards, such as producing unverifiable requirements.

2.1 Thinking about verification early

TIP

The earlier you begin to evaluate verification methods, the better. Knowing early on how you're going to satisfy each verification objective, including which verification tools you're going to use (if any), and how you're going to qualify any such tools, will reduce the cost of your verification.

As you need to describe the processes and tools you are going to use for verification during the Planning stage of a DO-178C project, you'll need to ensure that what you plan to do for your verification is feasible and efficient before you have designed any of your code, let alone implemented it. The earlier you invest in evaluating options for verification, the better. It pays to evaluate whether you will use manual processes or tools to satisfy each verification objective and evaluate the tools you intend to use while considering any tool qualification needs. Any changes to your verification processes after the completion of SOI#1 would require renegotiation of your compliance strategy with your certification authority, which could add add cost to and delay your project.

2.2 The Planning milestone (SOI#1)

When your plans are complete and have been reviewed and signed off internally, it will be time to take the first step in the compliance process by having the SOI#1 review with your certification authority. The SOI#1 review is typically a "desk" review performed by the authority, who will thoroughly review all plans and standards to ensure that all applicable objectives of DO-178C and any relevant supplements will be satisfied if the development proceeds as documented in your plans. This review will result in either a report documenting any observations or findings that must be resolved, or acceptance that your plans and standards are compliant. When your plans have been accepted, the next step is to follow them!

3. Configuration Management

Configuration Management, one of the Integral processes in DO-178C, is relevant throughout the DO-178C compliance process.

The Configuration Managment process requires you to ensure that you have robust processes for tracking the various compliance artifacts you will produce, tracking changes between different versions of those artifacts, and ensuring that your compliance artifacts are stored appropriately such that they can be retrieved later (this is often requested during SOIs). These processes are set up during Planning by producing a Software Configuration Management Plan (see *Planning* on page 6) and are followed throughout the DO-178C life cycle.

3.1 Control categories

Configuration items that you track through configuration management fall into two Control Categories, which determine the level of control that must be applied to items in that category. DO-178C specifies which items must be treated as Control Category 1 or 2 items based on the project's Design Assurance Level. Items treated as Control Category 1 (CC1) must undergo full problem reporting processes, and formal change review and release processes. Configuration items classified as Control Category 2 (CC2) items do not need to undergo these more formal processes, but items in this category must still comply with configuration identification and traceability needs, be protected against unauthorized changes, and meet applicable data retention requirements.

Configuration items

A configuration item is a software component or item of software life cycle data that is treated as a unit for software configuration management purposes.

3.2 Baselines

An important concept relating to Configuration Management is the baseline – a record of the state of specific compliance artifacts at a specific point in time. What counts as a baseline varies from project to project, but it always has the following characteristics:

- An artifact added to a baseline is immutable. The only way to work on it is to follow a formal change control process and create a new revision of the artifact, which you can then add to a later baseline.
- An artifact may be developed informally before you add it to the baseline.
- The artifacts within one baseline must be consistent with one another.

Typically, you will create a baseline for each phase of your software life cycle. For example, once high-level requirements have passed reviews and quality assurance checks, and you are about to transition to the software design phase, you may create a baseline representing the entire set of high-level requirements, reviews, analyses, the QA records, and traceability to system level requirements.

If your development is organized into modules, you can create a baseline per module. For example, you might establish a design baseline for each of three modules independently. Your configuration index submitted for approval should identify the relevant baseline, or baselines, of the accompanying compliance data.

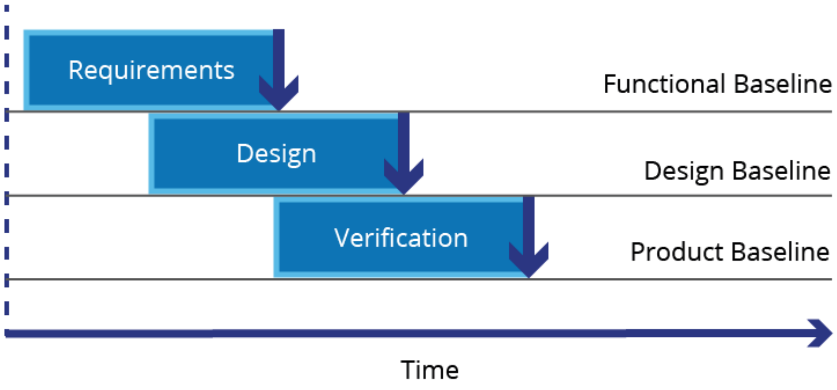


Figure 5 – Baseline progression

3.3 Problem reporting

DO-178C Control Category 1 items (see *Control categories* on page 8) are subject to a formal problem reporting process, which you will need to set up and implement. This process should capture how you plan to respond to problems identified for Control Category 1 items during your project, such as issues with implemented code not meeting requirements/standards, or verification activities not yielding results needed to demonstrate DO-178C compliance. You will need to write your Problem reporting process into your Software Configuration Management Plan.

Problem reporting can be considered as the process of managing changes to your artifacts. Your Problem reporting process should involve details on the process you will follow, typically including the following areas:

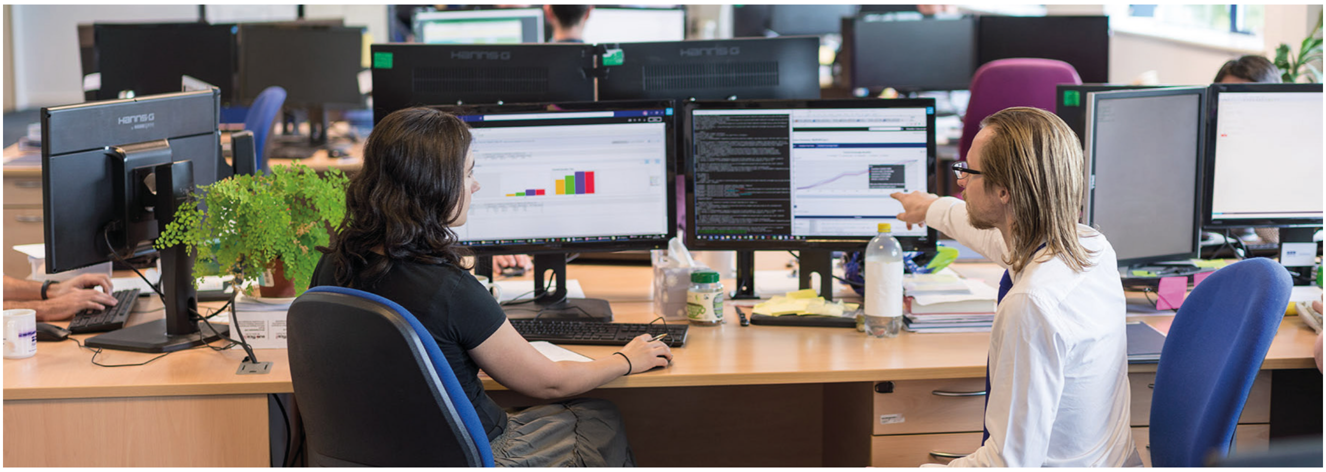
- How change requests, problem reports, issue reports or similar are raised. Your process should define the problem to be addressed or the need to be met. It must identify the configuration item and version affected, and especially in the case of a feature request, may include some notes on the expected solution.
- How you will triage requests to make sure that they adequately define scopes. How agreements are made on whether to proceed with a change, defer a change, or reject a change. Agreements to proceed with a change should capture specific resources and timescales. Such decisions are usually made by a Change Control Board.
- When changes are made, how you will update your documentation with details of the change, particularly identifying the configuration items and versions that resolve the issue or need.
- How you will close requests once all stakeholders agree that they are complete. This decision is also usually made by a Change Control Board.

Software Quality Assurance personnel should be involved in problem reporting to provide independent evidence that problem reports are being evaluated and resolved according to the Software Configuration Management Plan. SQA must have the authority to ensure that the engineering team act on problem reports.

Change Control Boards

A Change Control Board is a team of development experts (often including software engineers, testing experts, etc.) and managers (e.g. Quality Assurance managers), who make decisions on implementing proposed changes to a project.

4. Quality assurance



Software Quality Assurance (SQA), one of the Integral processes of DO-178C, provides the evidence showing that a project complies with processes, standards and guidance throughout the development life cycle. Broadly, there are six areas of software quality focus throughout a project.

At the beginning of a project:

1. Checking that the PSAC and the associated plans and standards align with the objectives of DO-178C.
2. Creating the Software Quality Assurance Plan (SQAP).
3. Checking that software life cycle processes comply with approved plans and standards.
4. Establishing supplier oversight.

During the project:

5. Checking that software activity follows the software life cycle processes – in particular, checking that transition criteria between life cycle processes are satisfied.

At the end of the project:

6. Conducting the conformity review of the software product.

These activities must be conducted whether software activities are internal to one organization or part of an integrator-supplier relationship. In the latter case, each organization that has further suppliers must coordinate SQA activities so that the integrator's evidence includes evidence from their supplier's SQA activities.

4.1 SQA independence

According to DO-178C, a project's SQA team must have the authority to ensure corrective action i.e. an organization must be arranged such that, if a software quality engineer discovers a problem, SQA cannot be overruled by some other part of the organization. This can be achieved both through processes (e.g. explicitly requiring SQA authorizations) and through the reporting structures within the organization (e.g. SQA should not report to engineering managers). SQA authorization or approval is typically needed for the data items (plans, standards, requirements, configuration index and so on) that form each baseline.

4.2 Initial SQA activities

Good software quality cannot be achieved as an afterthought. SQA should be involved in both defining plans and standards (see *Planning* on page 6) to make it as easy as possible to generate the evidence needed for SQA activities as well as checking through those plans and standards for alignment with DO-178C. When writing plans and standards, it makes sense to start with the end result you want to achieve – the software quality assurance records that you need for final submission – and work backwards to identify how you will generate each item. As well as informing the input into plans and standards, this should help produce checklists and identify supporting tools that you will use to help collect compliance evidence. You will typically end up planning to use a mix of tools and techniques and document this in the relevant planning documents.

SQA should be the primary authors of the Software Quality Assurance Plan (SQAP, see *Planning* on page 6), which should cover the initial, ongoing and conformity review activities that SQA will manage, as discussed here and below.

When supplier organizations are involved in your program, it is best to coordinate SQA activities between the organizations from the early stages of a project. For each SQA-specific item in your own plans, you should coordinate how you will obtain that information from the supplier, in what form, and at which milestones. This can be especially challenging for witnessing and participation activities (see *Ongoing SQA activities* on page 12), where you may need to delegate some authority to the supplier organization and then audit the responses that you obtain.

4.3 Ongoing SQA activities

The SQA Plan should include a mix of activities that will be performed throughout the project. These activities will generate the SQA evidence you need to show that your planned DO-178C processes were followed throughout the software life cycle.

These activities include:

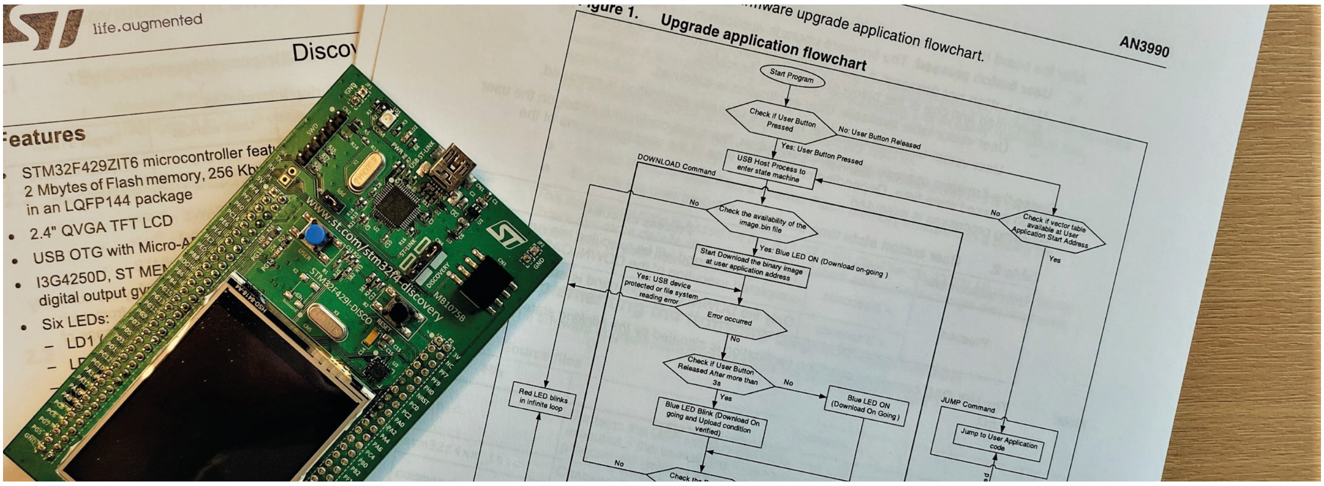
- **Independent assessment** – activities in which SQA independently (see *SQA independence* on page 12) assesses various activities specified in your plans and/or the outputs of those activities. Independent assessment can be applied to, for example, life cycle data, transition criteria, life cycle environment, SCM processes and records. Typical issues discovered through independent assessment activities include identifying software changes without corresponding requirements updates or identifying discrepancies in the versions of tools used by different parts of the engineering organization. It is sensible to build procedures, checklists and tools to support these activities, and consider what training your SQA team may need in independent assessment and auditing.
- **Participation** – activities in which SQA have a continuous active role. This can include, for example, participation in reviews, change control boards, and problem reporting (see *Problem reporting* on page 10). Software quality engineers would typically be present at some percentage of these meetings to help identify issues such as process deviations, missed checklist items, or confusing instructions. As well as generating software quality reports to explain attendance and identifying issues and corrective actions applied, another benefit of participation is to correct processes and update supporting materials as early as possible. This can help avoid the need for significant later rework to correct quality problems.
- **Witnessing** – this involves members of the SQA team witnessing activities such as testing and the build and load process. Witnessing can be in multiple forms, for example witnessing the original activity being performed, witnessing an engineer redo the activity, or independently performing the activity. Using a mix of these techniques will help to give a good spread between the efficiency of the checking process and the independence needed to find inconsistencies.

4.4 Conformity Review

The SQA team in a DO-178C project has the responsibility for conducting a conformity review. This review, typically performed as the final step going into SOI#4 (see *The Certification milestone (SOI#4)* on page 64), provides documentation showing that the software life cycle is complete. Your conformity review report will summarize and cross-reference evidence that you collect throughout the software life cycle, showing that:

- Planned processes were followed and generated traceable, controlled life cycle data, with complete and consistent baselines, including any data relating to changes from a previous baseline.
- Deviations and open problem reports were reviewed and approved.
- Software test, build and load activities were performed from repeatable, well-defined instructions.

5. Development


life.augmented
Discov
Features
STM32F429ZIT6 microcontroller feat
2 Mbytes of Flash memory, 256 Kb
in an LQFP144 package
2.4" QVGA TFT LCD
USB OTG with Micro-A
I3G4250D, ST MEI
digital output gy
Six LEDs:
www.st.com/stm32f4-discovery
STM32F429I-DISCO
MB1075B
Upgrade application flowchart
AN3990
Start Program
Check if User Button Pressed
No: User Button Released
Yes: User Button Pressed
USB Host Process to enter state machine
DOWNLOAD Command
Check the availability of the image.bin file
Check if vector table available at User Application Start Address
Yes: Blue LED ON (Download on-going)
Start Download the binary image at user application address
Yes: USB device protected or file system reading error
Error occurred
Check if User Button Released After more than 3s
Red LED blinks in infinite loop
Blue LED Blink (Download On going and Upload condition verified)
Blue LED ON (Download On Going)
JUMP Command
Jump to User Application code

After SOI#1, the next step is to develop your product. This involves writing requirements, designing software architecture and finally implementing the software itself.

5.1 High-level requirements

The first step in Development is to define the high-level requirements of your code. Later in Development, you will define low-level requirements. Developing high-level requirements involves refining the context of system requirements allocated to software and defining any software-specific requirements that do not map to system requirements as derived requirements. While defining both high and low-level requirements, you should ensure that you follow the processes you agreed to follow in your PSAC – your certification authority is likely to check that you have done so during future SOIs.

Requirements development for DO-178C projects

Engineers across your team will typically collaborate on requirements authoring and review, using requirements authoring tools and revision control systems to keep everyone synchronized. Pay attention to how you will meet major integral process objectives:

- For configuration management, make sure that you have unique identification of each version of each requirement. You don't need formal change control until you start later life cycle phases. For example, you can work informally on high-level requirements, discussing and amending as you go, but once you want to start the design work, you should create a baseline to store the reviewed and analyzed high-level requirements against which you perform the design work.
- Verification evidence at this level includes reviews and analyses of the consistency and completeness of the requirements.
- Quality assurance records from this stage can be generated by having SQA personnel involved in some of the requirements reviews, independently performing reviews and analyses, or having some proportion of the requirements re-reviewed or re-analyzed.

5.1.1 Your requirements should be verifiable

It is important to ensure that the requirements you write can be tested using the verification methodology you said you were going to use in your PSAC. This is important for both high and low-level requirements. If you generate requirements that can't be tested, you'll have two options: rewrite your requirements or change your verification methodology (and renegotiate the changes with your certification authority). The cost of both options is high, so it's best to ensure that your requirements are verifiable.

Common causes of unverifiable requirements include:

- Not including specific verifiable and quantitative acceptance criteria such as values or behaviors in requirements.
- Writing requirements that include implementation details without including enough detail on the behavior of a component to unambiguously define the aspects of its behavior that must be specified to support testing. An example of this is writing requirements for the behavior of a state machine without including information on all possible inputs and the outputs expected from different inputs. This information must be specified in requirements as it may not be possible to infer the behavior of a component during testing, depending on the level at which the testing occurs.

Having a well thought out Requirements standard should help your project avoid generating unverifiable requirements and reduce rework costs.

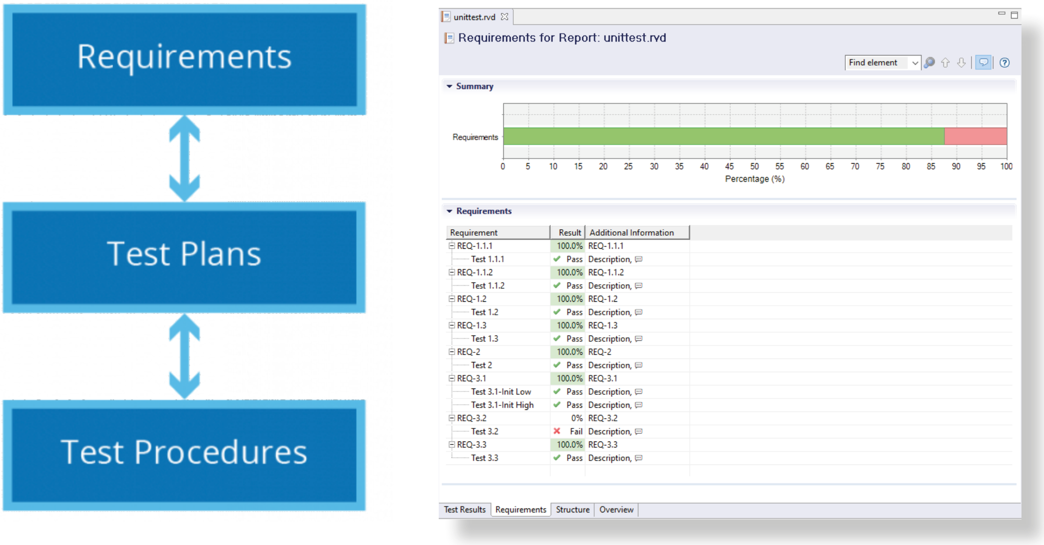
5.1.2 The importance of requirements traceability

TIP

Requirements traceability is crucial in DO-178C projects. Selecting a method (often a tool) to help you manage traceability information and automatically trace between artifacts will not only help you maintain an efficient workflow, but also support your SOIs. Some verification tools, such as the Rapita **Verification Suite**, can interface with some requirements-traceability tools, and it is worth considering this when selecting both requirements management and verification tools, as this could improve your efficiency.

Ensuring the correct management of traceability information (parent child relationships between requirements) is of paramount importance during the design assurance process. Aviation regulations dictate that airborne software must only implement Intended Aircraft Functionality. Traceability, and the validation of traceability links, is a key mechanism to ensure that only Intended Aircraft Functionality is implemented. The reason that DO-178C requires bi-directional traceability is that downstream traceability ensures that all Intended Aircraft Functionality is implemented, while upstream traceability ensures that all elements of the implementation exist to deliver Intended Aircraft Functionality. For this reason, traceability is a cornerstone of the SOI#2 and SOI#3 reviews.

Most projects working towards DO-178C compliance select to use a tool (either commercial or in-house) to manage requirements and generate traceability information. It is worth considering how well any tools you plan to use for verification interface with your requirements management tool. Some tools may offer features to import requirements information from your requirements management tool, thus helping you view your verification results in the context of your requirements. This makes it easier to map between your verification results and the requirements you're verifying, likely saving time and effort during your verification process.



Requirements for Report: unittest.rvd

Requirement	Result	Additional Information
REQ-1.1.1	100.0%	REQ-1.1.1
Test 1.1.1	✓ Pass	Description,
REQ-1.1.2	100.0%	REQ-1.1.2
Test 1.1.2	✓ Pass	Description,
REQ-1.2	100.0%	REQ-1.2
Test 1.2	✓ Pass	Description,
REQ-1.3	100.0%	REQ-1.3
Test 1.3	✓ Pass	Description,
REQ-2	100.0%	REQ-2
Test 2	✓ Pass	Description,
REQ-3.1	100.0%	REQ-3.1
Test 3.1-Init Low	✓ Pass	Description,
Test 3.1-Init High	✓ Pass	Description,
REQ-3.2	0%	REQ-3.2
Test 3.2	✗ Fail	Description,
REQ-3.3	100.0%	REQ-3.3
Test 3.3	✓ Pass	Description,

Figure 6 – Requirements traceability concept and dedicated requirements page in a Rapi**Test** report

5.2 Design

When your requirements are available, the next step is to design the architecture of your software and define its low-level requirements in order to define a system that can meet the high-level requirements you have identified. Design in DO-178C is the combination of architecture and low-level requirements. You will need to follow the design standards you said you would follow in your PSAC and provide evidence that you have done this through the SQA Integral process (see *Quality assurance* on page 11). Your certification authority is likely to check this evidence during future SOIs.

As software is designed, items that should have low-level requirements, but for which these requirements do not trace to high-level requirements, are often discovered. These requirements, known as derived requirements, can include requirements for things such as reusable libraries, scheduling behavior, or the numerical accuracy of the software. When considering bi-directional traceability, you must provide information to show where derived requirements come from. This could be a reference to the design process or a design standard, for example. Recording this information makes it easy to show that your software design is the product of a repeatable process. This type of traceability is also important for change impact analysis, as these decisions often affect more than one software component.

5.2.1 Developing software architecture

Software architecture defines the components that software will have, including the functionality that each component will achieve, and the relationships between components, specifically data flows and control flows. Your architectural decisions can have a significant impact on your project life cycle, not only for implementation but also for verification, particularly considering data coupling and control coupling.

Considering data coupling and control coupling

TIP

Architectural decisions you make can affect the efficiency of data coupling and control coupling coverage analysis during your verification. To make this analysis easier, you may want to:

- Limit the number of data and control couples in your design.
- Clearly define how each of the data and control couples in your design should operate, including specifying mode information.

Decisions you make in your software architecture will impact the verification effort needed for DO-178C compliance. As verification requires much more effort than implementation, it makes sense to consider the impact your decisions will have and choose a strategy to reduce total effort.

One way in which your design decisions impact verification is in the data coupling and control coupling of the interfaces that you plan to include in your implementation – to comply with DO-178C guidance at DALs C and above, you need to verify that you have exercised the data coupling and control coupling interfaces in the implementation (see *Data coupling and control coupling coverage analysis* on page 55), and your design decisions will affect the ease of this verification.

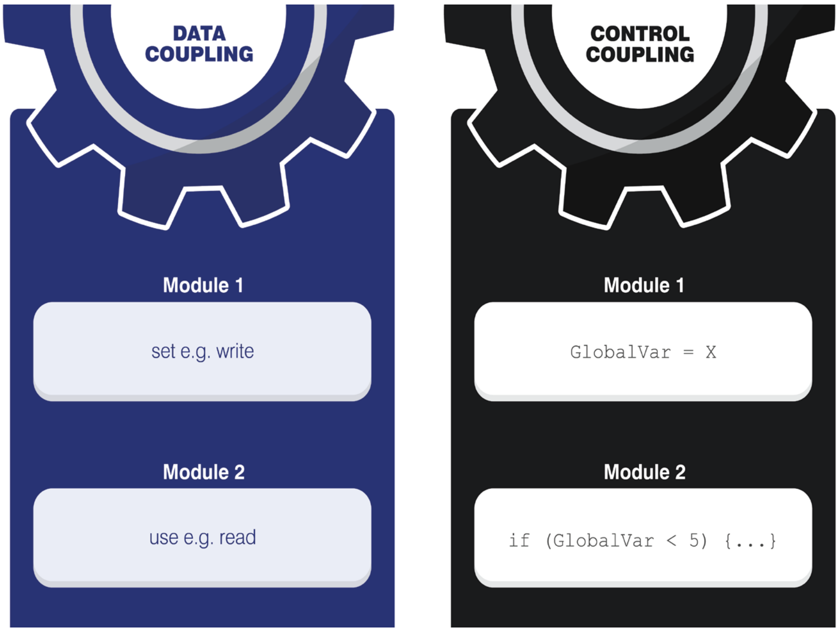


Figure 7 – Data coupling and control coupling illustration

As you need to verify that you have tested coverage of every data and control couple in your software for DO-178C compliance (see *Data and control coupling coverage analysis* on page 55), your verification will be much easier if you include fewer coupled components in your design rather than more.

The way that you define the couplings in your design will also impact the ease of your verification. To make verification of the data and control coupling in your system efficient, you may want to put extra effort into defining the data and control coupling interfaces thoroughly in your design by including information that will make it easier to verify the coverage later. Sensible items to include in such definitions include:

- Specifying data flow between components with detail such as Boolean and enumerated signals, numerical ranges and precisions, rates of change, update rates and fault flags.
- Describing components in terms of modes. If two components only interact in a certain way when one or both are in a particular mode, it makes sense to add that mode information to the component definition(s). Mode information helps to reduce complexity by limiting the number of interactions you need to consider.

Data coupling and control coupling

Data coupling and control coupling occur between components of your code that are separate within your architecture, but through which data can be passed or control decisions can be affected:

Data coupling occurs when data can be passed from one component of the code to another. The relevant data, its source and its destination form a data couple.

Control coupling occurs when a statement (such as a decision) in one component of the software affects which other software component(s) will execute (e.g. by choosing whether to call into the component or instructing the scheduler to start a task that uses that component). The components in which the control decision is made and from which the control decision is affected form a control couple.

- Specifying the control flow in terms of when and how control passes into and out of each component, including error cases and exceptions.
- Where software components interact with hardware, specifying data and control interactions in the same way as for software-software interactions.

The more such information that your interface definitions include, the easier your verification will be.

5.2.2 Developing low-level requirements

After developing the software architecture, the next step is to develop low-level requirements. This involves specifying requirements for the functionality you have assigned to each component in your software architecture. While most low-level requirements will typically map to high-level requirements, some components may include functionality not identified in high-level requirements and thus require that you develop derived requirements (see *Design* on page 17).

As with high-level requirements, low-level requirements must be developed according to your plans and requirements standards, and your requirements need to be both verifiable (see *Your requirements should be verifiable* on page 15) and traceable (see *The importance of requirements traceability* on page 16). When considering verifiability, you should consider how you will test the components in your software architecture, and if it looks like testing could be difficult (e.g. if it looks like testing will require a large number of mocks or stubs or a dependency injection approach), you may want to rethink your architecture.

Depending on your implementation decisions later on, you may need to develop new low-level requirements that do not trace to high-level software requirements, known as derived requirements.

5.3 Implementation

When your requirements and design are ready, the next step will be to implement your product in code. This is comparatively only a small part of the overall compliance process, usually taking 5% or less of overall effort.

As in every stage, you will need to ensure that your implementation follows the processes and standards you said you would follow in your PSAC and provide evidence that you have done this through the SQA Integral process (see *Quality assurance* on page 11). Your certification authority is likely to request evidence demonstrating that you have done so in future SOIs.

5.3.1 Implementation decisions can affect verification efficiency

TIP

Your implementation decisions can affect the efficiency of your verification. To keep your verification efficient, you should consider the impact that your choices (including programming language(s), coding standard(s) and choice of hardware platform and operating system) will have on your verification.

The decisions you make on how to implement your product can have major effects on your verification later on. These can make your verification much easier or much more difficult, and as a result cause your project to either run smoothly or incur delays. As verification takes much more effort than implementation, it's worth considering the effects your decisions will have and making decisions that will reduce your verification effort. Here are a few things you may want to consider:

- **Choice of programming language(s)** – if you're using any commercial tools for verification that need to analyze your source code (for example to apply instrumentation for structural coverage analysis), you should ensure that those tools support the language(s) you're using.
- **Choice of coding standard(s)** – DO-178C compliance requires that you define coding standards during your planning and follow them during your implementation. The standard(s) that you use can affect the ease of your verification:
    - Generally, the fewer and simpler the code constructs you use, the less effort you'll need to verify your software.
    - Some language constructs may not be supported by some commercial verification tools you intend to use, which may cause your project further costs or delay. If you evaluate verification tools, it's worth ensuring that they can analyze sample code written with the coding standard you plan to use.
    - Some coding standards require generated code to be written in a way that can cause development of untestable code, such as a standard requiring that default cases are included in every switch function. At higher DALs, you'll need to review this code even if you can't test it, which can incur extra effort. The cost of changing any of the items above grows larger the further you are through a DO-178C project, so you should ideally consider them before your DO-178C Planning.
    - DO-178C states that your coding standard should include code complexity restrictions. Limiting code complexity in general makes it easier to verify the code, while limiting the use of specific language features may be necessary for your source code to be compatible with the verification techniques and tools that you intend to use. If you are working at DAL A, you may need to take care to structure your low-level requirements so that your corresponding source code does not need an excessive number of conditions per decision. This will reduce the effort needed to achieve and maintain full MC/DC of your source code later (see *Structural coverage analysis* on page 46).

- **Choice of hardware platform and operating system** – your choice of hardware platform and operating system can have huge effects on the timing behavior and predictability of the final software, especially if you are using multicore processors. Some platforms and operating systems may have in-built features that make your verification easier – it's worth evaluating a range of options early and considering downstream effects on verification. For example, using hardware with on-chip tracing capabilities can make worst-case execution time analysis much easier than if only off-chip tracing capabilities are available.
- **Choice of compiler and compiler options** – the compiler and compiler options you use can affect how easy it is to verify your code. Your compiler and compiler options can especially affect the efficiency of your structural coverage analysis. Structural coverage analysis can be performed either by analyzing execution of source code or object code constructs (see *Source code vs. object code structural coverage analysis* on page 51). If object code is analyzed, additional verification may be needed to map the object code coverage to the source code. Compilers can optimize the code, making it require more effort to do this mapping. For DAL A software, it is also necessary to verify any additional code that is introduced by the compiler. In both cases, the more compiler optimizations that you allow to occur, the more effort you'll need to verify your code.
- **Use of emerging technologies** – if your implementation uses emerging technologies such as using GPU devices rather than CPU devices for computation, using multicore systems, or data-driven computation, your verification may be more challenging and time-consuming. While verification methodologies are well established for "standard" implementations, this is not the case for implementations using emerging technologies, and verification tool support is less available.

The cost of changing any of the items above becomes increasingly large the further you are through a DO-178C project, so you should ideally consider them before your DO-178C Planning.

GPUs for compute

In recent times, there has been an increasing use of Graphical Processing Units (GPUs) for computation in the avionics industry. Using GPUs for compute offers benefits when performing computationally intensive tasks due to high parallelization. By developing COTS GPU libraries to be used for safety-critical software, GPU manufacturers are supporting the use of GPUs by industries including the avionics industry.

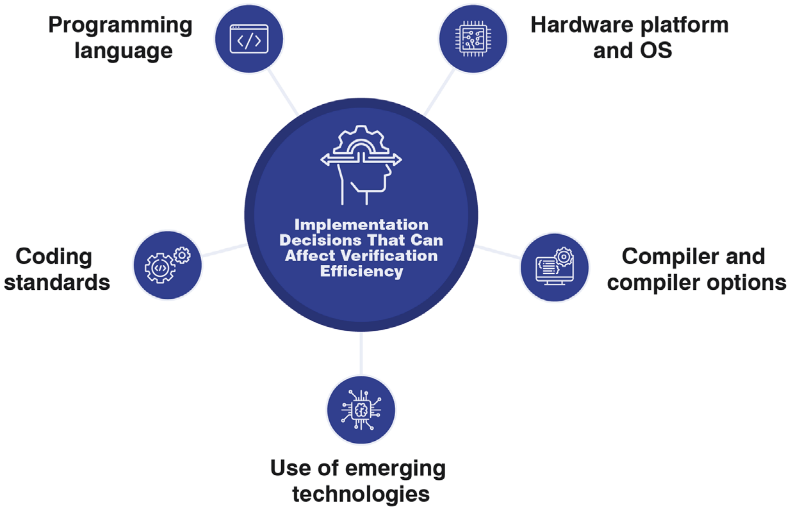

Programming language
Hardware platform and OS
Implementation Decisions That Can Affect Verification Efficiency
Coding standards
Compiler and compiler options
Use of emerging technologies

Figure 8 – Implementation decisions that can affect verification efficiency

5.3.2 It may pay to start verification early

Some verification activities, such as structural coverage analysis, take a long time. If your implementation generates code that can't be verified using your chosen methodology, you'll incur extra costs and potentially delays by needing to return to your implementation. This can be mitigated by verifying your code as you write it. A sensible approach is to write tests to ensure that you can get full coverage at the granularity you need it, then implement these as regression tests that are run throughout your development. While these are not requirements-based tests, and as such you won't be able to use them to support your compliance argument (see *Unit testing is not enough* on page 39), they give you confidence that you can test your code fully, and you'll be made aware of any changes to coverage from your regression tests so that you can rectify them. Some software verification tools can help you automatically generate test vectors to test your code in this way, reducing the effort needed to do so.

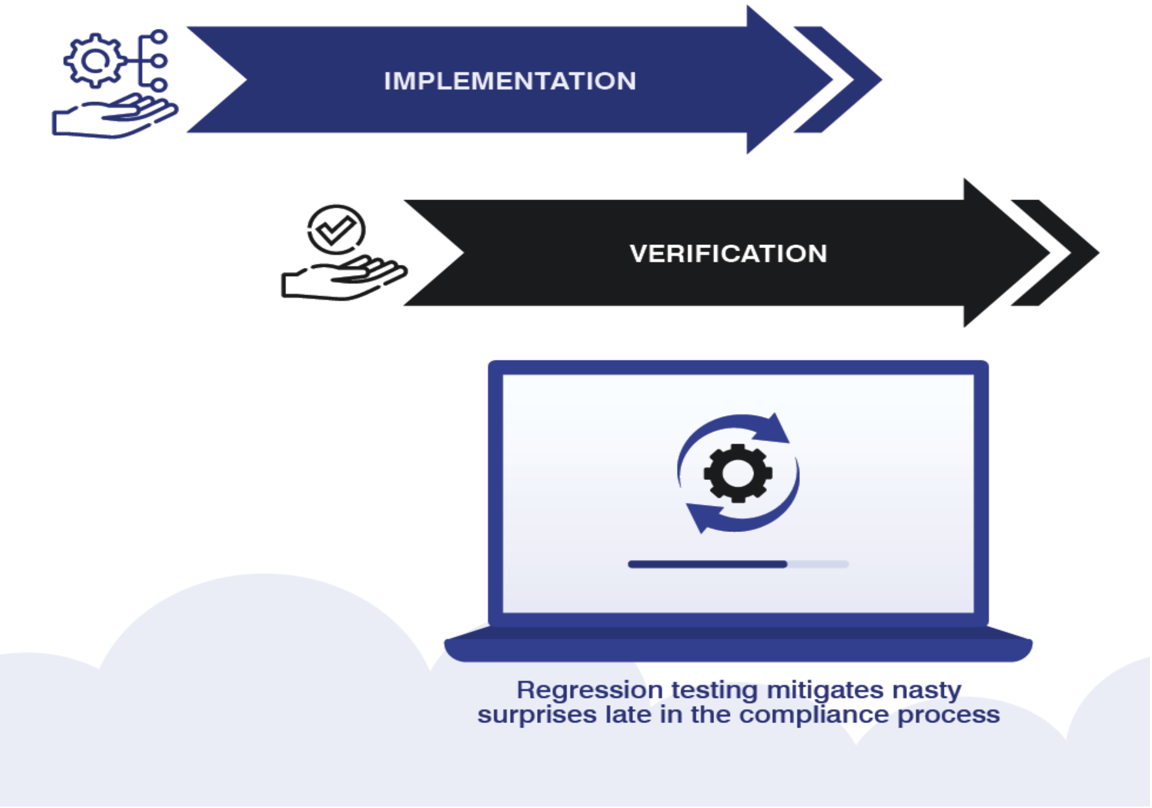

IMPLEMENTATION
VERIFICATION
Regression testing mitigates nasty surprises late in the compliance process

Figure 9 – Starting verification early and setting up regression tests can help you avoid unwanted suprises late in the compliance process

5.3.3 Using model-based design (DO-331)

Model-based design technologies can reduce the effort needed to design and test compliant software. Some benefits of using model-based design include being able to simulate software behavior, supporting unambiguous expression of requirements and software architecture, enabling automated code generation, and being able to perform some verification activities earlier in the software life cycle than would otherwise be possible.

RTCA DO-331 (Model-Based Development and Verification Supplement to DO-178C and DO-278A) provides additional objectives that apply when using model-based design in DO-178C projects, and clarification of how existing DO-178C objectives and activities apply to projects using model-based design.

One of the key additional verification activities discussed in DO-331 is model coverage analysis, which aims to detect unintended functionality in the model. As per DO-331, performing model coverage analysis does not eliminate the need to perform coverage analysis of the generated code that will actually execute.

The simulation tools provided by model-based design tools can reduce verification effort by providing a means to produce evidence that a model complies with its requirements. As per DO-331, using simulation does not eliminate the need to perform testing of the executable object code on the target hardware.

If you choose to use model-based design processes in a DO-178C project, you will need to understand the guidance in DO-331 and identify model-based design tools, as well as your verification and tool qualification strategies, in your DO-178C planning documents (see *Planning* on page 6).

Object-orientated programming

Object-oriented programming and related techniques offer benefits that can be attractive to software engineers, such as allowing encapsulation, increased code reusability and easier troubleshooting of code issues. As such, these technologies are often used in the aerospace industry.

5.3.4 Using object-oriented programming and related techniques (DO-332)

RTCA's DO-332 (Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A) provides additional objectives that apply for design, implementation and verification activities when using object-oriented programming and related techniques in DO-178C projects. It also clarifies how existing DO-178C objectives and activities apply to projects using these technologies and includes a discussion of vulnerabilities that may arise due to using object-oriented programming. While some organizations may avoid using object-oriented programming as this would require following DO-332, proper planning and use of the technology can lead to better software design and more maintainable and reusable implementations.

Some examples of extra objectives that must be achieved when using object-oriented programming include:

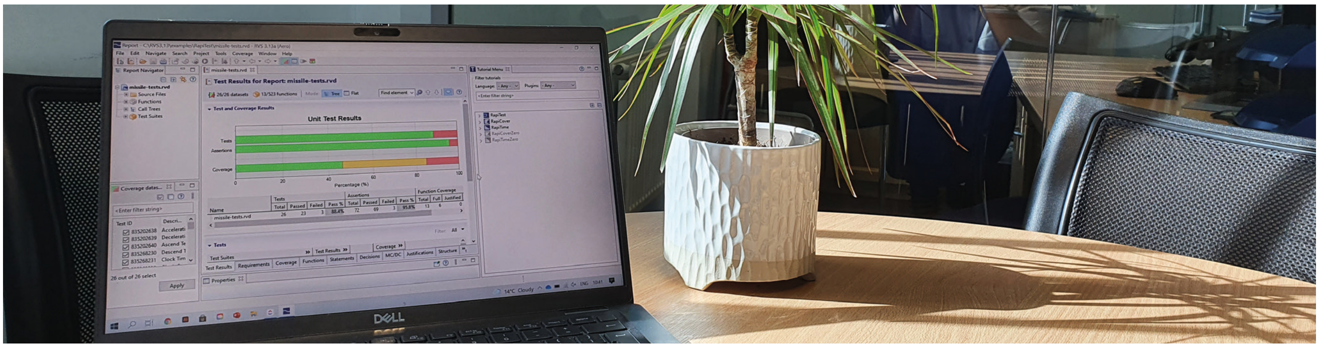
- Planning dynamic memory management and exception management strategies in your software architecture
- Verifying class hierarchy with respect to high-level requirements
- Verifying methods with respect to requirements
- Verifying local type consistency in the source code
- Verifying that dynamic memory management and exception management implementations are consistent with your software architecture and high-level requirements

If you choose to use object-oriented programming in a DO-178C project, you will need to understand the guidance in DO-332 and consider the impact that using the technology will have on all stages of your DO-178C life cycle, including verification. You will need to write up your implementation and verification strategies in your DO-178C planning documents (see *Planning* on page 6).

5.4 The Development milestone (SOI#2)

Before you have finished your Development, but when you have examples of each of your Development artifacts (typically 60%-80% of your total expected artifacts), you should conduct SOI#2 with your certification authority. SOI#2 focuses on the development process and artifacts, but the review also considers verification processes that should be running concurrently with Development – specifically, the review activities being implemented. SOI#2 may also look forward to the verification phase to see if there are any examples where verification activities have provided feedback to development activities. This may include, for example, test case development or test environment development providing feedback to your requirements and design processes to ensure that test activities can completely verify the functionality expressed in your requirements and design.

6. Verification



A number of verification activities are required by DO-178C. Some of these activities involve verification of the developed software (see *Verification of DO-178C software* on page 31), while some involve verification that your DO-178C verification process was correctly followed (see *Verification of the verification process* on page 46).

Many verification activities can be performed either manually or by using automated tools to help run the analysis. When automated tools are used to achieve a DO-178C objective without their output being verified, those tools must be qualified for use following the DO-330 Guidelines (see Qualifying verification tools (DO-330) on page 60). Qualification materials are often available from COTS tool providers.

As we discussed earlier, the number of verification objectives you must meet depends on the DAL level of your software:

- For DAL A systems, 43 of the 71 applicable objectives relate to verification
- For DAL B systems, 41 of the 69 applicable objectives relate to verification
- For DAL C systems, 34 of the 62 applicable objectives relate to verification
- For DAL D systems, 9 of the 26 applicable objectives relate to verification

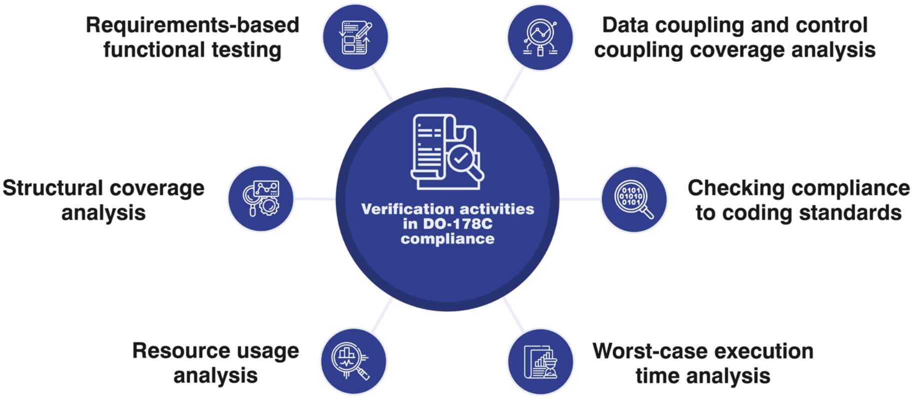

Requirements-based functional testing
Data coupling and control coupling coverage analysis
Structural coverage analysis
Verification activities in DO-178C compliance
Checking compliance to coding standards
Resource usage analysis
Worst-case execution time analysis

Figure 10 – Verification activities in DO-178C compliance

6.1 Preparing for SOI#3

TIP

To prepare for SOI#3, make sure that you begin verification early to produce compliance artifacts for every verification activity you need to perform.

During SOI#3, you will need to provide evidence that you have produced compliance artifacts for each of the verification activities you need to comply with based on your software's DAL (see *The Verification milestone (SOI#3)* on page 62).

Because of this, it makes sense to begin your verification activities early and not leave verification activities until late in your project life cycle.

6.2 Who does the testing?

Many DO-178C objectives that involve testing require independence in testing at some DALs. When this is required, the person (or group) implementing a particular item and the person (or group) verifying that item must be different. Independence requirements are identified within DO-178C's objective tables. Higher DAL developments require independence on a larger number of objectives than lower DAL developments.

Independence in DO-178C testing

RTCA defines independence as separation of responsibilities to ensure the accomplishment of objective evaluation. For software verification activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified.

6.3 Testing on-host vs. on-target

TIP

Access to test environments representative of the final system is often limited until later stages of DO-178C projects. Beginning verification early with on-host testing and migrating to on-target testing when on-target test environments become available is an excellent verification strategy.

The ideal method used to perform a final run for score (see *The final run for score* on page 63) to produce verification results is to run tests on an integrated hardware/software environment representative of your final flight system. This stems from the following activities in DO-178C:

- **4.4.3.b** – "The differences between the target computer and the emulator or simulator, and the effects of these differences on the ability to detect errors and verify functionality, should be considered. Detection of those errors should be provided by the software verification process and specified in the Software Verification Plan."
- **6.4.1.a** – "Selected tests should be performed in the integrated target computer environment, since some errors are only detected in this environment."

It is rarely practical to run all tests on the final system, however, due to two major limitations:

- **Cost** – on-target testing is much more expensive than on-host testing. Due to the cost of hardware test rigs, the number and availability of these can be a limitation, while the availability of systems for on-host testing is never limited in this way.
- **Availability of test rigs** – test rigs may not be available until relatively late in your Development, meaning that if you choose to only do on-target testing, your testing may be delayed by lack of test rigs, potentially introducing delays in your project.

Typically, a range of test environments is defined as part of the Software Verification Plan activities, for example the following:

- **On-host** – on-host testing environments are useful for early stages of verification to set up and evaluate testing processes, especially when targets may not yet be available. On-host tests provide less assurance that test behavior is representative of behavior on the final flight system for a number of reasons, for example because the compiler and compiler options used may not be representative of the final target. When running tests on-host, some devices may not be available and not all functionality can be tested.
- **On-host simulator** – simulator environments are again useful for early stages of verification. A simulator of the target will typically use a cross-compiler with the same compiler options that will be used on the final target and tests run on a simulator will thus be more representative than tests run on-host. While tests run on a simulator provide more assurance than those run on-host, they are still not enough to provide assurance of the final flight system – simulators may not be able to accurately simulate all devices, and not all functionality can be tested.
- A range of **on-target test environments** may be available during a project, which are more or less representative of the final flight system. While many types of environment may exist and may be given different names by different DO-178C applicants, we will describe some example environments as Model A-C environments:
    - **Model A (blue label)** – a Model A environment may be a development board that uses the same or a similar CPU and target compiler as the final system. Typically, these environments have other devices to aid in debugging and testing, enabling testing of some devices that will exist on the final target.
    - **Model B (red Label, open box)** – a Model B environment may be very similar to the final system e.g. including the intended RTOS, BSP and devices, but still in development. The hardware in these environments may be modified compared to that intended for the final flight system to support testing and debugging. For example, the Model B environment may have extra hardware set up to expose GPIO pins or debugger headers for testing and debugging purposes. All devices in the final system would typically be available.

- **Model C (black label, closed box)** – a Model C environment will be the final board itself. This may have limited or no connectivity to support debugging. Typically, all tests for the run for score would be performed on a Model C environment, but due to the cost of this testing, this would only be done after it is known that the system is working and that the tests are complete (after testing using previous test environments).

A sensible approach to testing is to start on-host testing early in your project (see *It may pay to start verification early* on page 22), then start on-target testing (e.g. Model A, B or C) later, when test rigs are available.

6.4 Manual vs. automated verification

TIP

The best option of whether to use a verification tool or not for a specific verification activity depends on the project. For relatively simple verification activities for which little rework is likely to be required, using manual processes may be the better option, while for complex and time-consuming verification activities for which rework is likely, an automated tool may be the better option. You should always consider whether your use of any tool will need to be qualified or not and if qualification kits are available for any tools you evaluate.

It is possible to do all of the verification needed for DO-178C compliance manually, but it would be extremely inefficient.

Verification activities can be performed entirely through manual processes, can be assisted by tools, or can be performed using tools alone. Tools can be developed internally or provided by external suppliers. Using tools can help you automate time-intensive verification activities, but there are factors to consider in deciding whether or not to use a tool rather than a manual process for a specific verification activity. These include the size and complexity of your code base, the complexity of the verification activity, and the time-saving that using a tool may be able to provide throughout the lifetime of your project.

Developing an internal tool for a specific verification activity can be a good idea as you can develop it to best meet your specific use cases, however the cost of developing and maintaining such a tool can be high, and if you need to provide tool qualification evidence for an internal tool you use (see *Qualifying verification tools (DO-330)* on page 60), you will need to either develop this internally or find another company to do the qualification for you.

If you have any specific needs from a verification tool and these aren't addressed by any of the tools on the market, it may pay to contact several vendors and discuss whether they are willing to develop features for you.

Ultimately, the decision of whether to use tools or not should depend on whether this will make your verification more efficient. While tools may make some verification projects more efficient, every project has different needs, and in some cases, for example in a small project, using a tool may not improve your efficiency. It is best to consider the specific needs of your project and compare the efficiency of using manual processes vs. using an automated tool.

6.5 Tool qualification and DO-330

The *DO-330: Software Tool Qualification Considerations* supplement to DO-178C gives specific guidance on situations in which you need to qualify any verification tools you use in order to demonstrate that they work as intended.

For more information on Tool qualification and DO-330, see *Qualifying verification tools (DO-330)* on page 60.

6.6 Documenting verification results

Certification authorities will expect that you document your verification results in a series of documents, some of which you will submit in your final compliance demonstration data. The documents that are typically expected include:

- **Software Accomplishment Summary (SAS)**, which gives the overall compliance position, states the timing and memory margins, and summarizes any agreed process deviations and open or deferred problem reports. This document is always submitted.
- **Software Verification Results (SVR)**, which lists the verification activities conducted (reviews, analyses and tests), gives the results, summarizes problem reports for any failed verification activities, and typically also includes traceability and coverage data. This document is sometimes submitted and sometimes just made available for external review.
- **Software Verification Cases and Procedures (SVCP)**, which gives the design of each review, analysis and test to conduct. This includes details such as test environment setup, schedules, staffing, auditing, and efficiency concerns. It isn't expected that this document is submitted.

Other documents may be expected if you are using emerging technologies in your development.

The above list makes a one-to-one correspondence between a named DO-178C output and a document. In practice, you can structure your deliverables into whatever document arrangement best suits your development approach. This includes splitting out one output into multiple documents (e.g. to put all your resource usage verification planning into a separate document) or supporting one document with another more detailed document (such as a general execution time test strategy document supporting the resource usage part of the SVCP).

The form of the SVR will depend on the verification activities performed.

- **For reviews**, the SVR would typically summarize the completion of checklists, while the checklists themselves stored under appropriate configuration control with unique identifiers.
- **For analyses**, you will often create dedicated analysis reports. For example, your resource usage results and structural coverage analysis results might be best presented in their own reports and then summarized in the SVR. For consistency and maintainability, consider generating the summary as part of the detailed document, and then copying that across to the SVR. This will make it easier to deploy review effort: experts reviewing the detailed document can check the detailed meaning of the summary, while the consistency of the text between the documents can be handled with a review checklist item.
- **For tests**, you will keep the results of test execution (e.g. log files or test tool output files) and summarize those in the SVR. Make sure that you can trace each test execution result back to the corresponding test procedure and to the verification environment configuration that was used for the testing.

If part of your SQA activity involves witnessing or repeating verification activities (see *Ongoing SQA activities* on page 12), you will typically document those as part of the SQA record. If the result of a repeated activity is different to the original result, you should invoke your problem reporting process (see *Problem reporting* on page 10) to analyze and repeat the activity.

6.7 Multicore systems

The use of multicore systems brings extra complexity to software behavior, and as a result to the verification activities that must be performed to provide sufficient design assurance. As multicore systems were not used in avionics when DO-178C was released, DO-178C itself includes no guidance on certification concerns for multicore systems. This was addressed by the CAST team in CAST-32A[2] , which is due to be superseded by A(M)C[3] 20-193 in 2021. For more information on DO-178C compliance of multicore systems, see *Verifying multicore systems (A(M)C 20-193/CAST-32A)* on page 57.

[2] https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf

[3] A(M)C refers to either EASA AMC (Acceptable Means of Compliance) or FAA AC (Advisory Circular) documents.

DO-178C tests, reviews, and analyses

DO-178C verification activities can include reviews and analyses as well as tests.

According to DO-178C, a **test** *exercises a component to verify that it satisfies requirements* (both normal range and robustness requirements), while a **review** *provides a qualitative assessment of correctness*, and an **analysis** *provides repeatable evidence of correctness.*

6.8 Verification of DO-178C software

For DO-178C compliance, software must be verified against both functional and non-functional requirements.

DO-178C software must be shown to comply with the coding standards you have agreed to follow in your DO-178C planning (see *next section*). Functional requirements in DO-178C projects should be verified by requirements-based functional testing (see *Requirements-based functional testing* on page 33), while non-functional requirements are typically verified with a combination of test, review, and analysis, including analysis of the resource usage of the DO-178C system (see Resource usage analysis on page 39).

6.8.1 Checking compliance to coding standards

For DO-178C compliance, you must demonstrate that you have followed the coding standards that you said you would follow in your Software Development Plan (see *Planning* on page 6).

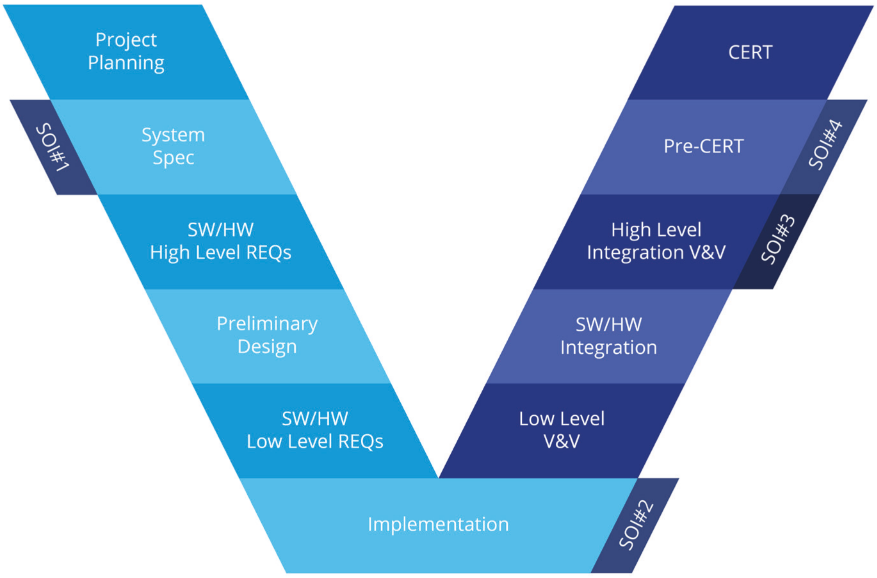


Figure 11 – Software develpoment model showing likely locations of SOI audits

TIP

It pays to check your code's compliance to coding standards early, and this should ideally be a continuous process throughout your implementation. This is because any deviations of your implementation from your coding standard will necessitate changes to your code, which will have a downstream effect on all other verification activities, requiring rework.

Manual vs. automated compliance checking

TIP

Using a tool to check compliance against coding standards is generally preferred for DO-178C compliance as code compliance checking is a highly computational activity. When evaluating compliance checking tools, it makes sense to consider the following:

- Does the tool support the coding standard you plan to use?
- If you're not using a commonly used standard, does the tool let you define custom rules?
- Does the tool integrate with the continuous integration software you are using?
- Is the tool qualifiable for DO-178C?

Compliance to coding standards can be checked manually or supported through the use of automated compliance checking tools.

Compliance can be analyzed manually by reviewing every section of the code against your coding standard. This can take a long time, the process is prone to human error, and if any changes are made to the code after a specific component is checked, it may incur extra rework. Nonetheless, if your implementation uses relatively few lines of code, following a manual approach may be preferred.

Due to the factors above, most companies working towards DO-178C compliance use automated tools to check compliance to coding standards, which significantly reduces the effort needed to check for compliance. Most tools support checking against commonly used coding standards such as MISRA C™ for C and C++ code[4]. If the coding standard you plan to follow includes rules not present in these standards, some tools may offer features to let you define custom rules to check your code against. Most compliance checking tools will integrate with any continuous integrations tools you use, so compliance checking is automatically performed with each update of the source code. If you choose to use a compliance checking tool, you should consider whether the tool is qualifiable for DO-178C (see *Qualifying verification tools (DO-330)* on page 60) – if not, you'll need to manually review the results generated by the tool.

[4] https://www.misra.org.uk/MISRAChome/tabid/181/Default.aspx

6.8.2 Requirements-based functional testing

DO-178C requires evidence to demonstrate that the function of the code has been tested adequately with respect to the system's high and low-level requirements in two ways:

- **Normal range testing** provides evidence that the software will provide intended aircraft function and that it complies with high and low-level requirements
- **Robustness testing** provides evidence that the software will provide intended aircraft function for all foreseeable operating conditions

Both types of testing should be done without examining the implementation.

With the exception of DAL D software, for which testing need only be done against high-level requirements, the expectation when following DO-178C guidance is that every high and low-level requirement in the system has been tested by both normal range and robustness tests. When tests are complete, one or more test cases should be directly traceable from every requirement in the system.

The execution of requirements-based tests can be done manually, but this is a laborious process, so automated functional testing tools are often used.

TIP

As you'll need to document the method by which you plan to perform requirements-based testing in your DO-178C planning documents, you should evaluate methods early during your DO-178C planning. If you choose to use any functional testing tools, you should also evaluate these early.

Requirements-based functional testing in DO-178C

The need to perform requirements-based testing for DO-178C compliance is listed in 4 DO-178C objectives – 6.4.a, b, c and d (Table 1).

Table 1. DO-178C objectives relating to requirements-based testing
(Source: RTCA DO-178C)

	Objective		Applicability (DAL)				Output	
ID	Description	Ref.	A	B	C	D	Description	Ref.
A-6, 1	Executable Object Code complies with high-level requirements.	6.4.a	❍	❍	❍	❍	Software Verification Results Trace Data	11.14 11.21
A-6, 2	Executable Object Code is robust with high-level requirements.	6.4.b	❍	❍	❍	❍	Software Verification Results Trace Data	11.14 11.21
A-6, 3	Executable Object Code complies with low-level requirements	6.4.c	●	●	●	●	Software Verification Results Trace Data	11.14 11.21
A-6, 4	Executable Object Code is robust with low-level requirements	6.4.d	●	●	●	●	Software Verification Results Trace Data	11.14 11.21

❍ Objective required at DAL
● Objective required with independence at DAL

For DAL D software, functional testing is only required against high-level requirements. For higher DALs, it is required against both high and low-level requirements. For DAL B software, there must be independence between those implementing the code and those verifying that it complies with low-level requirements, while for DAL A software, there must be independence between those implementing the code and those verifying that it both complies with and is robust with low-level requirements.

Normal range and robustness testing

Normal range testing involves testing how the software responds to normal inputs and conditions. Each normal range test used to claim compliance must trace to a high or low-level requirement. As normal range tests used to claim DO-178C compliance must be requirements-based, unit testing is not appropriate for demonstrating compliance, though it can still be useful (see *Unit testing is not enough* on page 39).

Robustness testing involves stressing the system, including comprehensively testing it with the full range of equivalent data classes, and testing its behavior when abnormal conditions and faults are present. In robustness testing, abnormal conditions should be tested, such as testing a function with input values outside the expected range.

When testing a component for robustness with respect to data inputs, not all possible values must be tested – DO-178C defines the concept of ***equivalence classes***, or classes of data that behave in an identical way, and it is only necessary to test a single member of each equivalence class. To make your generation of robustness tests more efficient, you should ensure that you identify the equivalence classes in your system early, and you may want to design the interfaces and data dictionary of your system to limit the number of equivalence classes you will need to test.

Testing on-host vs. on-target

To comply with DO-178C guidance, you will need to collect evidence that the Executable Object Code meets its requirements while it executes on a target as representative of the final environment as possible. Typically, access to such a target may not be available until later stages of the project (see T*esting on-host vs. on-target* on page 26).

While you may not be able to claim credit for on-host testing, using this approach can still be sensible as it allows you to de-risk your on-target testing as much as possible until you have access to test rigs for on-target testing. In such an approach, you may develop and run tests on-host during the early stages of the project (see *It may pay to start verification early* on page 22), then repeat them on a Model A/B/C system (see *Testing on-host vs. on-target* on page 26) when the tests are relatively complete and these test environments are available. It may not be possible, however, to run tests against some high-level requirements on-host, as these tests may need the devices intended for the system to be available.

Manual vs. automated functional testing

TIP

Using a tool to run requirements-based tests is generally preferred for DO-178C compliance as functional testing is a highly computational activity. When evaluating functional testing tools, it makes sense to consider the following:

- Which test formats are available with the tool, and how can they help you reduce test authoring effort?
- Can the tool help you write and run robustness tests through forcing abnormal software behavior and injecting faults?
- Can the tool run tests on-target?
- Will the tool integrate with your existing system, or do you need to modify your system to use it?
- Can the tool help you merge results from different tests and builds?
- Does the tool integrate with any continuous integration tools you are using so you can view your results throughout your project?
- Is the tool qualifiable for DO-178C, and are any of its features that you want to use qualifiable (if needed)?

Functional testing can either be performed mostly manually (the software will of course still need to be compiled and run) or can be supported by using an automated functional testing tool.

Manual functional testing can take a huge amount of time, the process is prone to human error, and if any changes are made to the code after the functional testing is performed, it may incur significant extra rework. For this reason, most companies working to DO-178C guidance use automated functional testing tools, which improve testing efficiency by automating or supporting some or all of the following:

- **Writing functional tests** – most tools provide one or more methods to write tests that are more efficient for functional testing than using raw code in common programming languages such as C. Testing formats are typically designed to make it easy to write and review tests. The test formats supported by different tools differ, and tools offered by some vendors may better fit your project and use cases, so it pays to consider your options. You should consider whether the tool you select can help you test abnormal conditions by forcing abnormal software behavior and injecting faults.

- **Convert tests into test harnesses that can be run on the host computer or target hardware** – functional testing tools convert test cases written in an appropriate input format into a test harness that can be executed to collect results. As on-target testing is crucial for DO-178C compliance (see *Testing on-host vs. on-target on* page 26), you should ensure that any tools you evaluate create a test harness that can be run on-target.

- **Run the tests on-host or on-target to collect results** – some functional testing tools may integrate with your build system to automatically run generated tests on your local machine or target system, saving effort as you don't need to do this manually. As on-target testing is crucial for DO-178C compliance (see *Testing on-host vs. on-target* on page 26), you may want to ensure that any tools you evaluate can run tests on your target hardware. Note that it should be possible for a tool to integrate with your build system to run tests – if a tool you're evaluating isn't able to offer this and requires you to integrate with a specific build system or compiler, for example, you may want to consider other options.

- **Analyze results** – functional testing tools will typically provide features that help you efficiently analyze test results and make any modifications needed, for example filtering options that let you quickly identify failing tests. Some tools may provide further features that can help you analyze results, such as integrating with any continuous integration system(s) you may be using so test results can be automatically collected during every build of the code, letting you track test results throughout the project. As mentioned above (Writing functional tests), the testing format(s) used by a tool can help reduce review effort. When selecting a tool, it is sensible to consider how the available features may support your verification.

- **Merge results** – you'll likely produce test results from a large number of different builds. Some functional testing tools can help you merge results from those tests into a single report. If you use such a feature, you should consider whether the feature is qualified for DO-178C (see *Qualifying verification tools (DO-330)* on page 60) – if not, you'll need to manually review the results.

- **Traceability** – traceability information is crucial in DO-178C compliance, and you may be asked to demonstrate traceability artifacts during SOI audits (see *The importance of requirements traceability* on page 16). Functional testing tools can make it much easier to trace between your artifacts quickly. Some tools support traceability by integrating with requirements management tools so you can efficiently import requirements information to add to your test results and potentially export requirements information back to your requirements management tool.

- **Export results and traceability results** – when providing DO-178C compliance evidence for functional testing, you will need to include a summary of your test results, and you'll likely have many such results. Most functional testing tools will include features to export your results much more efficiently than using manual processes. In some cases, exports will also include traceability results. If you plan to use export features to generate your final compliance results, you should consider whether any export features you plan to use are qualified for DO-178C (see *Qualifying verification tools (DO-330)* on page 60) – if not, you'll need to manually review the exports.

Merging test results

Some functional testing tools, like the Rapita **Verification Suite**, enable users to merge results from separate test runs. An example use case of this feature is merging results collected from different testing strategies (system, integration and lower level testing) and different builds to combine test results into a single report. Along with test results, structural coverage and timing data can also be merged.

When evaluating functional testing tools, it is important to consider whether any features you plan to use are qualifiable for DO-178C (see *Qualifying verification tools (DO-330)* on page 60). If they aren't and they need to be, using such a feature may be a false economy as you may need to manually review any relevant artifacts generated by the tool.

You need to test what you fly

A common approach to functional testing is to use a commercial functional testing tool to create a functional test harness and collect results. Often, structural coverage analysis (see *Structural coverage analysis* on page 46) is run at the same time. The methods used by most structural coverage analysis tools to collect coverage results involve adding additional "instrumentation" code to the source code. As this instrumentation code may interfere with the function of the code, the final run for score (see *The final run for score* on page 63) should be run with an executable compiled without additional instrumentation code or any other code that is not directly traceable to a high or low-level requirement.

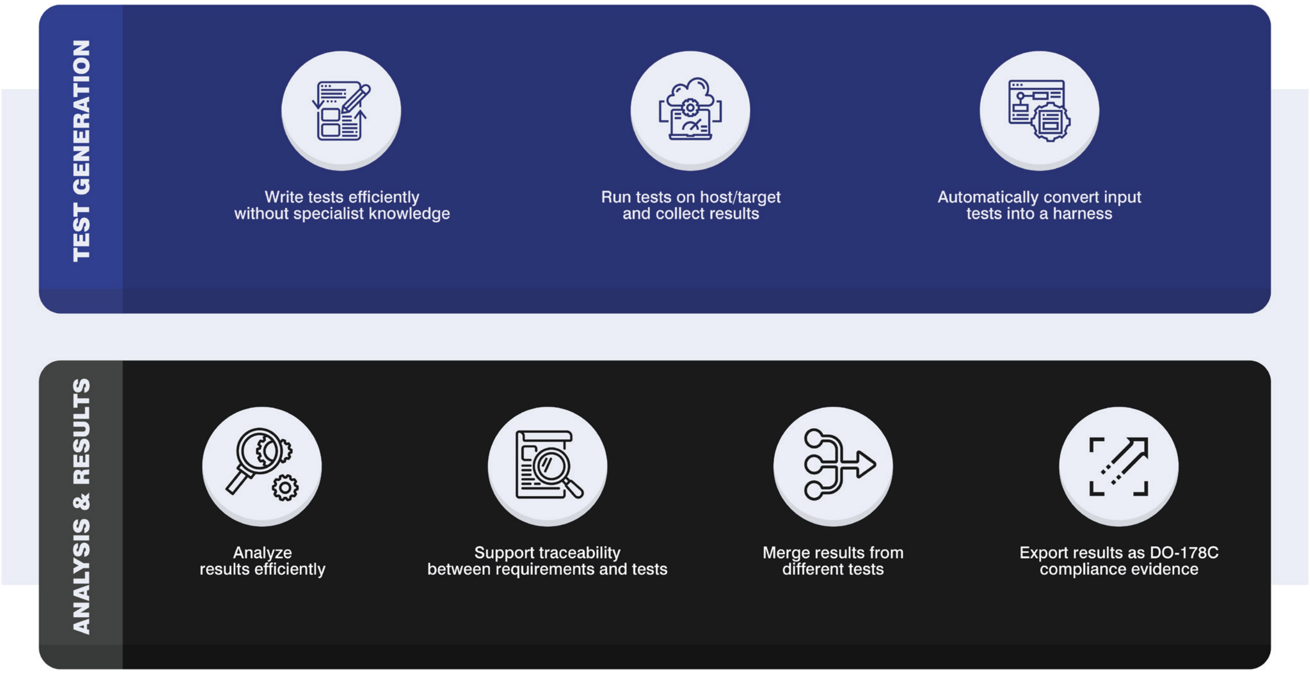

TEST GENERATION
Write tests efficiently without specialist knowledge
Run tests on host/target and collect results
Automatically convert input tests into a harness
ANALYSIS & RESULTS
Analyze results efficiently
Support traceability between requirements and tests
Merge results from different tests
Export results as DO-178C compliance evidence

Figure 12 – How functional testing tools can support verification activities

Unit testing is not enough

TIP

While unit tests can't be used to claim credit for DO-178C compliance, they can be a good way to informally test your code and set up a suite of regression tests that can help you identify issues earlier and reduce overall verification effort.

In many software industries, functional testing is achieved in part by traditional unit testing (testing a specific subcomponent) of the code. Traditional unit tests may, for example, test that expected output values are returned from a subcomponent of the code when a range of input values are tested. When developing software according to DO-178C guidance, however, traditional unit tests are not an appropriate means of compliance – functional tests used to claim compliance must be written to satisfy high or low-level requirements of the system.

While units test results can't be submitted for DO-178C compliance, it may still be useful doing unit testing, as unit tests can be created at the same time the code is implemented (while this may not be the case for requirements-based tests), they can be used to ensure that it is possible to get 100% coverage of the implemented code, and they can contribute to a suite of regression tests (see *It may pay to start verification early* on page 22).

Ensuring completeness

As you approach your run for score (see *The final run for score* on page 63), you should ensure that all of your requirements have been completely covered by tests. This means not only that you should have tests traceable to every requirement, but also that you should have all of the tests needed to fully test the requirement, including any robustness tests.

6.8.3 Resource usage analysis

When following DO-178C guidance, it is necessary to provide evidence that resource usage in software has been considered and analyzed. DO-178C software has requirements that relate to usage of a range of resources, including:

- Memory requirements such as memory usage, stack usage and cache management
- Timing requirements (response time and execution time)
- Requirements related to contention in areas of the system such as bus loading and input/output hardware

The ability of software to meet these requirements does not typically decompose easily across components. Instead, resource usage requirements relate to emergent properties of the fully integrated software. While you cannot completely determine these properties before formal verification, you can take steps to control and monitor the ability to meet these requirements (and hence likely save overall compliance effort) throughout the software life cycle. As with most verification activities, it makes sense to begin considering how you will analyze resource usage early in your DO-178C process.

Resource usage analysis in DO-178C

DO-178C includes three objectives related to resource usage analysis, as shown in Table 2.

Table 2. DO-178C objectives relating to resource usage analysis (Source: RTCA DO-178C)

	Objective		Applicability (DAL)				Output	
ID	Description	Ref.	A	B	C	D	Description	Ref.
A-4, 13	Software partitioning integrity is confirmed.	6.3.3f	●	❍	❍	❍	Software Verification Results	11.14
A-5, 6	Source Code is accurate and consistent.	6.3.4f	●	❍	❍	❍	Software Verification Results	11.14
A-6, 5	Executable Object Code is compatible with target computer.	6.4.3a	❍	❍	❍	❍	Software Verification Cases and Procedures Software Verification Results	11.13 11.14

❍ Objective required at DAL
● Objective required with independence at DAL

Budgeting and measuring execution time is the most challenging aspect of resource usage management. We focus on specific issues relating to execution time analysis in *Worst-case execution time analysis* on page 43.

Analyzing resource usage

TIP

To efficiently comply with DO-178C's resource usage objectives, you should consider resource usage analysis throughout the DO-178C process by:

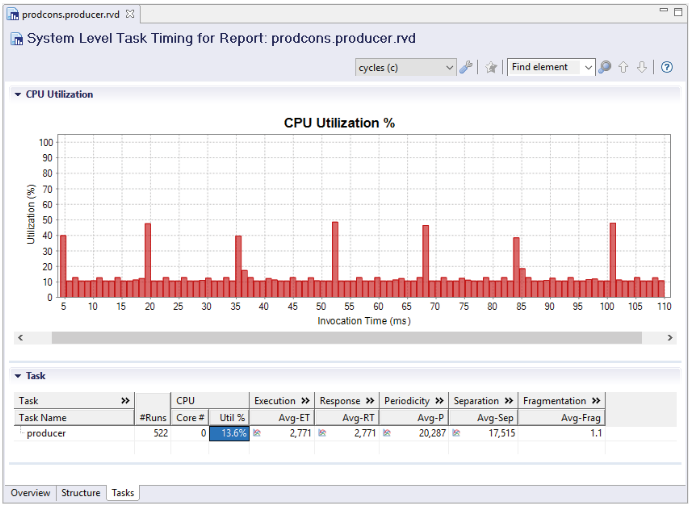
- During planning, developing a requirements standard that ensures that you can mitigate against unwanted surprises later.
- When designing your software, identifying high-level requirements that relate to resource usage and decomposing these to specific software components can help you mitigate the risk of late discovery of resource usage issues.
- Collecting evidence on your software's resource usage during your requirements-based testing. Using a consistent approach for monitoring resource usage at all levels of testing can help you reduce analysis and review effort.
- Throughout your DO-178C process, recording your resource usage analysis progress in a resource usage report. This is an efficient way of developing the evidence you'll need to demonstrate compliance against resource usage objectives.

As is the case with all verification activities, development and implementation of a robust, efficient methodology for analyzing resource usage involves considering the activity throughout the DO-178C life cycle, beginning with Planning (see *Planning* on page 6).

For DO-178C compliance, you will need to be able to check that your requirements are feasible when you review them. As such, when you create and review your requirements standard, you should ensure that it explains how to write requirements for resource usage that you will be able to verify (see *Your requirements should be verifiable* on page 15). In your processes, you should also include the steps you will take if you discover that resource usage requirements are infeasible. If you discover this early, you may be able to negotiate with system design and safety processes, or the hardware design, to implement an alternative approach. If you only discover these issues later in the process, however, you will have fewer options, and may even be forced to re-allocate budgets and re-work unrelated parts of the software to meet resource usage requirements.

When you develop high-level requirements (see *High-level requirements* on page 14), you should identify which of your requirements relate to emergent resource usage. This will allow you to track the status of your resource usage analysis by using traceability from these requirements.

When you develop the architecture for your software (see *Design* on page 17), decomposing your resource usage requirements and allocating them to each component in your software architecture can help you to de-risk late discovery of resource usage issues.



prodcons.producer.rvd

System Level Task Timing for Report: prodcons.producer.rvd

cycles (c)

CPU Utilization

Task

Task		CPU		Execution	Response	Periodicity	Separation	Fragmentation
Task Name	#Runs	Core #	Util %	Avg-ET	Avg-RT	Avg-P	Avg-Sep	Avg-Frag
producer	522	0	13.6%	2,771	2,771	20,287	17,515	1.1

Figure 13 – Rapi**Task** displaying CPU utilization metrics

There is no general analytical method to apply when allocating resources to components, but the accepted practice is to divide up your available resources with a tentative budget for each component, and monitor component resource usage against that budget. Your Software Verification Plan should explain how verification at lower levels relates to the achievement of requirements at higher levels. For memory usage, including all of the different linker sections as well as stack and heap size budgets can help to manage and trade off budgets throughout development. Your target and/or tool chain may impose limits here as well – for example, your linker may be unable to handle data storage that spans multiple on-target memory segments. If your software architecture includes partitioning, you will need to demonstrate that partition breaches are prevented. Even if you are not using partitioning, you should still consider how to mitigate the effects of exceeding your component budget.

Your review processes for your software architecture should include feasibility checks after your architecture has been defined but before your software has been implemented. For execution time analysis, you can use schedulability analysis to show that, if tasks fit within their budget, the software as a whole will meet its deadlines. For static memory usage, you can usually just add up the allocated sizes to show that they fit, but you should be aware that some memory may need to be allocated in specific devices to be able to support your software's functionality. For dynamic memory usage, you should consider the call tree within and between components as well as any dynamic allocations made during start-up. In all cases, you should consider how you will demonstrate a repeatable analysis process to be able to meet SQA objectives.

When developing your requirements-based tests (see *Requirements-based functional testing* on page 33), you will need to develop test cases that trace to the resource allocations you made during your software design. Even though these logically fit within the software integration level of testing, you will find it useful to be able to execute these scenarios during hardware-software integration testing to verify resource usage (and hence compatibility with the target computer) in that environment. When you run requirements-based tests, you should ensure that you monitor resource usage consistently at all levels of testing, as this will help you reduce analysis and review effort. This may mean, for example, using the same approach that you use for execution time testing against low-level requirements for your software integration testing and hardware-software integration testing.

Collating your current progress and achievements for verifying your software's resource usage in a Resource Usage Report throughout your DO-178C project can be an efficient way of showing that your development meets the relevant DO-178C objectives (see *Resource usage analysis* on page 39). This document should include links to your resource usage requirements.

Worst-case execution time

The Worst-case execution time (WCET) of a piece of software is the longest time that that piece of software could possibly take to execute. As such, if the WCET of a piece of software is lower than its timing deadline, that piece of software will execute before its deadline every time it executes.

Worst-case execution time analysis

To adhere to DO-178C guidance, you will need to produce evidence that demonstrates that your software meets its timing deadlines as defined in high and low-level requirements. This is demonstrated by performing worst-case execution time (WCET) analysis.

Two main methods are used to analyze software WCET: *static analysis* and *measurement-based* approaches.

WCET analysis can be performed manually, but this approach is laborious, so automated WCET analysis tools are often used.

TIP

As you'll need to document the method by which you plan to analyze the worst-case execution time of your code in your DO-178C planning documents, you should evaluate methods early during your DO-178C planning. If you choose to use any automated analysis tools, you should also evaluate these early.

Static analysis vs. measurement-based approaches to WCET analysis

Two major approaches are used for worst-case execution time analysis: static analysis and measurement-based approaches.

In the static analysis approach, a model of the system's hardware and software is generated and used to estimate* the worst-case execution time of software components. A major benefit of using this approach is that when a model is available, the analysis can be run automatically by verification tools, and that in this case there is no need to generate tests for software timing behavior. A major drawback is the need for a model of the system – if no model is available for the system, it can be difficult, time-consuming and costly to generate an accurate model. Model accuracy is also a factor – the more complex the system, the more costly and less feasible it is to generate an accurate model of the system. Furthermore, a system may demonstrate behavior that only occurs infrequently and is unlikely to be modelled, such as a system's execution time being affected by the temperature of the hardware.

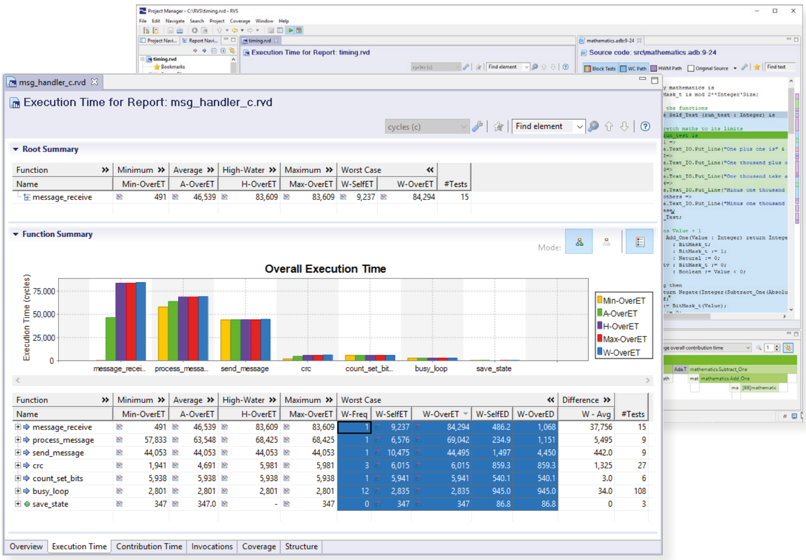
For very complex systems such as multicore systems, it is infeasible to generate an accurate model. Worst-case execution time estimates from static analysis can also be very pessimistic – estimated worst-case execution times are the true* worst-case execution time of the software but they generally do not take into account factors that may reduce the actual worst-case execution time such as mutually exclusive paths in the code. As a result, estimated values may exceed the timing deadlines defined in requirements for the software.

** While the calculated worst-case execution time for each software component is a true worst-case execution time for the model, it is intractable to create a model that accurately reflects any but the simplest system, so the real worst-case execution time will differ from that estimated from the model.*

In the measurement-based approach, tests that exercise the software are run in an environment representative of the final flight system (on-target tests), and the timing behavior of the software is measured during these tests. The worst-case execution time of the software is then estimated± from the tests that take the longest time to run. Major benefits of this approach include that the results it generates are less pessimistic than those generated using a static analysis approach and that the results reflect the actual hardware environment. A major drawback is that extensive testing of the software must be done (and must be done on the target system, which may have limited availability) to use this method of testing. Another drawback is that as the results do not actually reflect the software's pathological worst-case execution time±, they may be optimistic, yielding a value less than the actual worst-case execution time of the software. The better the testing, the more assurance that results are not optimistic.

± This method does not actually identify the worst-case execution time of software, but its high water mark execution time. The accuracy of the results depends on the quality of testing. Typically, engineers performing timing analysis using this method will apply a safety margin on top of any results generated using this method (e.g. add 20% to the calculated high water mark execution time to generate an estimated worst-case execution time).

A hybrid approach can also be used that combines the best of the static analysis and measurement-based approaches. This approach, used by some commercial WCET tools such as Rapi**Time**, uses results from both static analysis of the software and on-target testing to generate results that are not optimistic and are less pessimistic than those generated by static analysis approaches. In this approach, timing tests are run to generate a high water mark execution time, and the software's control flow is then used to estimate a worst-case execution time from this.



Execution Time for Report: msg_handler_c.rvd

Root Summary

Function	Minimum	Average	High-Water	Maximum	Worst Case		
Name	Min-OverET	A-OverET	H-OverET	Max-OverET	W-SelfET	W-OverET	#Tests
message_receive	491	46,539	83,609	83,609	9,237	84,294	15

Function Summary

Function	Minimum	Average	High-Water	Maximum	Worst Case					Difference	
Name	Min-OverET	A-OverET	H-OverET	Max-OverET	W-Freq	W-SelfET	W-OverET	W-SelfED	W-OverED	W - Avg	#Tests
message_receive	491	46,539	83,609	83,609	1	9,237	84,294	486.2	1,068	37,756	15
process_message	57,833	63,548	68,425	68,425	1	6,576	69,042	234.9	1,151	5,495	9
send_message	44,053	44,053	44,053	44,053	1	10,475	44,495	1,497	4,450	442.0	9
crc	1,941	4,691	5,981	5,981	3	6,015	6,015	859.3	859.3	1,325	27
count_set_bits	5,938	5,938	5,938	5,938	1	5,941	5,941	540.1	540.1	3.0	6
busy_loop	2,801	2,801	2,801	2,801	12	2,835	2,835	945.0	945.0	34.0	108
save_state	347	347.0	-	347	0	347	347	86.8	86.8	0	3

Figure 14 – Worst Case Execution Time analysis report in Rapi**Time**

Sensible things to consider when selecting an approach to worst-case execution time analysis include the availability of a model for your system, the complexity of the system, and the experience and ability of your engineers to write code to optimize worst-case, rather than average case, timing behavior.

Manual vs. automated WCET testing

Worst-case execution time analysis can be performed entirely manually, or can be supported by the use of automated verification tools.

Software worst-case execution time can be analyzed manually by developing and running a test suite that intends to exercise worst-case execution time behavior for the software (including considering variability in behavior across different execution paths through the code) and reviewing results. This can take a huge amount of time, the process is prone to human error, and if any changes are made to the code after the analysis is run, it may incur significant extra rework. For this reason, most companies working towards DO-178C compliance use automated verification tools to support the process, which significantly reduce the effort needed to analyze software execution time behavior regardless of whether they use a static analysis, measurement-based, or hybrid approach (see *Static analysis vs. measurement-based approaches to WCET analysis* on page 43). Regardless of whether WCET testing is done manually or supported through an automation tool, typically, a safety margin is added to recorded high-water mark execution times to provide additional assurance that the software will meet its timing deadlines.

Timing analysis on resource-limited systems

If you perform worst-case execution time analysis using an automated tool, your verification efficiency may be affected by the resources available on your system. This is because of the way that most worst-case execution time analysis tools work, by adding additional "instrumentation" code to your source code and then using a mechanism to store or capture additional data while your code runs. Limitations in the availability of the following resources may cause additional verification overheads:

- **Code size** – as most tools add additional code to "instrument" the native code, it may be possible that the tool cannot add instrumentation to all of your code at once, and you may need to split your analysis over multiple builds.
- **Memory** – one method of collecting timing results on-target is to have data written to an area of memory and later output the data from this area of memory. If you use such a mechanism to collect timing results but do not have enough available memory to collect data for your entire code base at once, you may need to split your analysis over multiple builds, pause your software's execution during testing to capture data before clearing the memory location so more data can be captured, or use a different method to collect data.
- **Bandwidth** – another commonly used method to collect timing results on-target is to have timing data written to an external device such as a debugger or datalogger and collected on-the-fly. If your system is not capable of writing accurate and thread-safe instrumentation data to an IO port fast enough, or capable of writing this data to the device capturing the data fast enough, you may need to split your analysis over multiple builds or use a different method to collect timing data.

Zero instrumentation timing analysis

Two main approaches exist for automated timing analysis: techniques that apply instrumentation to source code to measure timing behavior during execution, and instrumentation-free approaches. The latter use methods such as interpeting branch traces collected from a platform during execution to understand the timing behavior of an application.

If you plan to use a commercial tool to perform worst-case execution time analysis, it makes sense to consider how well suited the tools you evaluate are in handling each of the limitations above, especially the ones you are likely to encounter on your system. The amount of additional code needed for instrumentation can vary greatly between different tools, so using some tools may reduce the number of builds you need to fully test your code. Across a project, this alone can make a huge difference in the cost of on-target testing. Some tools may include features that help you automatically apply partitions, reducing the effort needed to verify your entire code base.

6.9 Verification of the verification process

DO-178C includes a range of verification objectives that are used to demonstrate that DO-178C processes, including the Verification process itself, have been followed appropriately throughout a DO-178C development project.

These objectives include, for example:

- **Structural coverage analysis** (see *Structural coverage analysis* on page 46), which demonstrates that the DO-178C software has been thoroughly tested such that it can be claimed that the software includes no unintended functionality
- **Data coupling and control coupling coverage analysis** (see *Data and control coupling coverage analysis* on page 55), which demonstrates that the data coupling and control coupling interfaces in DO-178C software have been sufficiently exercised during testing

These activities must be carried out by analyses or reviews.

6.9.1 Structural coverage analysis

DO-178C requires evidence to demonstrate that the code has been covered by requirements-based testing. Either source code or object code can be tested, and each type of testing has its benefits and drawbacks, as described in *Source code vs. object code structural coverage analysis* on page 51.

The purpose of coverage (often called structural coverage to be distinguished from requirements coverage) testing is to ensure that an application has no unintended functionality.

The expectation when applying for DO-178C compliance is that 100% of the code has been executed during testing or that, in cases where this is not possible (e.g. for defensive code that cannot be executed under normal conditions), the reason this code is untested is justified.

TIP

As you'll need to document the method by which you plan to verify the structural coverage of your software testing in your DO-178C planning documents, you should evaluate methods early during your DO-178C planning. If you choose to use any structural coverage analysis tools, you should also evaluate these early.

Structural coverage can be analyzed at different granularities. The broadest granularity of analysis considers the proportion of functions in the code that have been called, while the narrowest granularity considers the proportion of conditions in the code that have been shown to independently affect the outcome in the decisions in which they occur.

Structural coverage testing can be done manually, but this is a laborious process, so automated structural coverage analysis tools are often used. Coverage can either be tested at the source code or object code level, and each type of testing has benefits and drawbacks.

Structural coverage analysis in DO-178C

The need to perform structural coverage analysis for DO-178C compliance is listed in 3 DO-178C objectives – 6.4.4.2, 6.4.4.2a and 6.4.4.2b (Table 3).

Table 3 - DO-178C objectives relating to structural coverage analysis (Source: RTCA DO-178C)

	Objective		Applicability (DAL)				Output	
ID	**Description**	**Ref.**	**A**	**B**	**C**	**D**	**Description**	**Ref.**
A-7, 5	Test coverage of software structure (modified condition/ decision) is achieved.	6.4.4.2	●				Software Verification Results	11.14
A-7, 6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●			Software Verification Results	11.14
A-7, 7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●	❍		Software Verification Results	11.14

❍ Objective required at DAL

● Objective required with independence at DAL

For DAL C, independence between the individuals implementing and verifying the code. For DALs A and B, independence is required for all structural coverage analysis activities.

Granularities of structural coverage

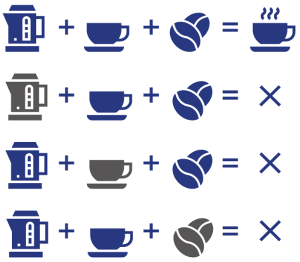
DO-178C lists 3 different types of structural coverage analysis that may need to be performed, depending on the Design Assurance Level (DAL) of the tested application. These are:

- **Statement Coverage** – this analyzes whether statements in the code have been executed, and the overall coverage percentage indicates the proportion of statements that have been executed.
- **Decision Coverage** – this analyzes whether decisions in the code have been executed and observed to have both a true and false outcome, and the overall coverage percentage indicates the proportion of decisions that have been executed and observed to have both true and false outcomes.
- **Modified Condition/Decision Coverage** – this analyzes whether conditions in each decision in the code have been executed and shown to independently affect the outcome of the decisions in which they occur, and the overall coverage percentage indicates the proportion of conditions that have been executed and shown to independently affect the outcome of the decision in which they occur (see below).

Modified Condition/Decision Coverage

Modified Condition/Decision Coverage (MC/DC) is intended to ensure that each condition in every decision independently affects the outcome of that decision. As an example of how this works, consider making a cup of coffee. To make a cup of coffee, you'd need each of the following: a kettle, a cup and some coffee beans.

```
if (kettle && cup && coffee){
        return cup_of_coffee;
} else {
        return none;
}
```



(continued on next page)

Modified Condition/Decision Coverage (Continued)

Pairs of test vectors (shown below) can show that each condition (kettle, cup and coffee) independently affects the outcome of the decision:

- Tests 4 and 8 show that `kettle` independently affects the outcome
- Tests 6 and 8 show that `cup` independently affects the outcome
- Tests 7 and 8 show that `coffee` independently affects the outcome

	Inputs			Output
Test	Kettle	Mug	Coffee	Results
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	0	1	0
7	1	1	0	0
8	1	1	1	1

To fully test this code for MC/DC, you'd need to run tests 4, 6, 7 and 8.

Sources of structural coverage

Structural coverage can be collected from system tests, integration tests, and lower level (component) tests, so long as each type of testing represents requirements-based tests. Higher level testing (such as system testing) generally produces more structural coverage per test than lower-level testing (such as component testing) does. As such, it is common practice to generate as much coverage as possible through higher-level tests before running lower-level tests to cover sections of the code that haven't already been covered.

Testing on-host vs. on-target

DO-178C recommends that structural coverage analysis is performed on a target as representative of the final environment as possible, but this testing is likely to be expensive and not available until later stages of the project (see *Testing on-host vs. on-target* on page 26).

A sensible approach to structural coverage analysis is to develop and run tests on-host during the early stages of the project (see *It may pay to start verification early* on page 22), then repeat them on a Model A/B system (see *Testing on-host vs. on-target* on page 26) when the tests are complete and such a system is available. When collecting structural coverage results from system level tests, it may only be possible to do so in Model B or C environments as system tests need all devices in the system to be available.

Manual vs. automated structural coverage analysis

TIP

Using a tool to perform structural coverage analysis is generally preferred for DO-178C compliance as functional testing is a highly computational activity and the analysis must typically be repeated. When evaluating structural coverage analysis tools, it makes sense to consider the following:

- Does the tool let you mark sections of your code as covered by manual analysis?
- Will the tool integrate with your existing system, or do you need to modify your system to use it?
- Can the tool help you merge results from different tests and builds?
- Does the tool integrate with any continuous integration tools you are using so you can view your results throughout your project?
- Is the tool qualifiable for DO-178C, and are any tool features you may want to use qualifiable (if needed)?

Structural coverage can either be analyzed manually, or automatically by using a structural coverage analysis tool.

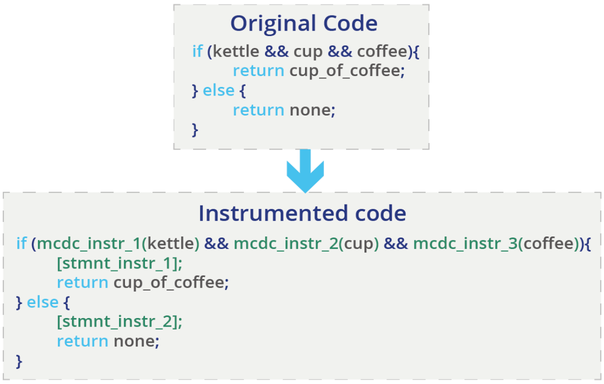
Structural coverage can be analyzed manually by following the execution trace of test runs of the code, analyzing the source code and the test steps needed to test the code, and marking every section of code that was covered. This can take a huge amount of time, the process is prone to human error, and if any changes are made to the code after the structural coverage analysis is run, it may incur significant extra rework, which some structural coverage analysis tools can mitigate through various features. Because of the above, most companies working towards DO-178C compliance use automated structural coverage analysis tools, which significantly reduce the effort needed to analyze structural coverage by automatically applying coverage instrumentation to code and analyzing the code segments that are executed during testing (see *How coverage instrumentation works* info box on page 51).

Structural coverage analysis tools can provide further benefits over manual analysis approaches:

- They often let you justify the reason that any untested code has not been tested and let you manage these justifications through an intuitive interface.
- They usually let you automatically export your analysis results into a format that is appropriate for providing to a certification authority.
- They may be able to integrate with continuous integration tools so you can automatically collect coverage results with every build.
- They can significantly reduce the number of tests you need to run to reverify your code if it changes through features such as merging results from multiple reports and code change analysis.

How coverage instrumentation works

Coverage instrumentation works by adding instrumentation code to the original code, such that coverage results can be collected when specific areas of the code are reached during execution. Instrumentation code may support data collection through different means depending on the execution environment, such as writing results to a file for on-host testing or writing results to an on-target buffer for on-target testing. Returning to our previous cup of coffee example from the *Modified Condition/Decision Coverage* info box on pages 48 and 49), here's how the code would look when instrumented for statement, decision and MC/DC coverage analysis.



Original Code

```
if (kettle && cup && coffee){
        return cup_of_coffee;
} else {
        return none;
}
```

Instrumented code

```
if (mcdc_instr_1(kettle) && mcdc_instr_2(cup) && mcdc_instr_3(coffee)){
        [stmnt_instr_1];
        return cup_of_coffee;
} else {
        [stmnt_instr_2];
        return none;
}
```

In this example, the instrumentation at `[stmnt_instr_1]` and `[stmnt_instr_2]` instruments the code for statement coverage. As this instrumentation occupies both true and false branches of the decision, if both statements are observed to execute, this also implies that the decision has been covered. The instrumentation at `[mcdc_instr_1]`, `[mcdc_instr_2]` and `[mcdc_instr_3]` allows Modified Condition/Decision coverage of the decision to be determined.

Source code vs. object code structural coverage analysis

TIP

There are benefits to both source code and object code structural analysis, but source code analysis is used much more often in DO-178C projects. When selecting a methodology to collect coverage results, you should consider:

- Whether you have access to source code for all of the software in your project.
- Whether your hardware environment supports coverage analysis without needing to apply instrumentation to the code.
- The DAL of your software – for DAL A software, if you perform coverage analysis on object code, you will need to provide additional evidence to show that any additional code introduced by your compiler has been verified.

Structural coverage analysis can be performed on the source code, the object code, or both. There are benefits and drawbacks to both types of analysis.

Coverage analysis is most often performed on source code, especially when using automated tools for the analysis. One major benefit of analyzing source code is that this often makes it easier to modify tests to produce 100% coverage (though these tests should still be written to test any relevant requirement(s), not just to produce coverage!) – while most testers can analyze source code to improve tests, this is not necessarily the case for object code. Another benefit is that most structural coverage analysis tools are designed to analyze source code and can't analyze object code, and using these tools can significantly reduce effort as we saw in *Manual vs. automated structural coverage analysis* on page 50. A major drawback of source code coverage analysis is that the typical approach of instrumenting the source code to collect coverage results can necessitate running multiple builds, especially for resource-limited systems (see *Coverage analysis on resource-limited systems* on page 52).

Object code structural coverage analysis is performed less often than source code coverage analysis in the aerospace industry. A major benefit of this type of analysis is that it can be performed even when there is no access to source code (for example if you are using a third-party library for which you don't have access to source code). Another benefit is that some methods for object code coverage analysis do not require instrumentation of the code, allowing testing of unmodified code. These methods typically impose restrictions on compatible hardware environments, however. A major drawback of this type of analysis is that if you choose to analyze coverage of object code, you will need to demonstrate that your analysis provides the same level of confidence that you would get from analysis at the source level. Further guidance on this was given in the CAST-12 position paper. Another drawback is that less support is available from commercial tool vendors to perform coverage analysis of object code as most structural coverage analysis tools cannot do this.

Regardless of whether you perform structural coverage analysis against source code or object code, if you are working on a DAL A development, you will need to provide additional evidence to show that any additional code introduced by your compiler has been verified (see *Verifying additional code introduced by compilers* on page 54).

Coverage analysis on resource-limited systems

If you perform structural coverage analysis using an automated tool, your verification efficiency may be affected by the resources available on your system. This is because of the way that most structural coverage analysis tools work, by adding additional "instrumentation" code to your source code and then using some mechanism to store or capture additional data while your code runs.

Zero instrumentation coverage analysis

Two main approaches exist for automated coverage analysis: techniques that apply instrumentation to source code to measure coverage achieved during test execution, and instrumentation-free approaches. The latter use methods such as interpeting branch traces collected from a platform during execution to measure the coverage achieved. Thus, they analyze the object code rather than source code for an application.

Limitations in the availability of the following resources may cause additional verification overheads:

- **Code size** – as most tools add additional code to "instrument" the native code, it may be possible that the tool cannot add instrumentation to all of your code at once, and you may need to split your analysis over multiple builds.
- **Memory** – a commonly used method to collect structural coverage results on-target is to have coverage data written to an area of memory and later output the data from this area of memory. If you use such a mechanism to collect coverage results but do not have enough available memory to collect data for your entire code base at once, you may need to split your analysis over multiple builds, pause your software's execution during testing to capture data before clearing the memory location so more data can be captured, or use a different method to collect coverage data.
- **Bandwidth** – another commonly used method to collect coverage results on-target is to have coverage data written to an external device such as a debugger or datalogger and collected on-the-fly. If your system is not capable of outputting data fast enough, you may need to split your analysis over multiple builds or use a different method to collect coverage data.

If you plan to use a commercial tool to perform structural coverage analysis, it makes sense to consider how well suited the tools you evaluate are in handling each of the limitations above, especially the ones you are likely to encounter on your system. The amount of additional code needed for instrumentation can vary greatly between different tools, so some tools may require you to run fewer builds than others. Across a project, this alone can make a huge difference in the cost of on-target testing. Some tools may include features that help you automatically apply partitions, reducing the effort needed to verify your entire code base.

If you do need to run your analysis across multiple builds, it pays to consider how the tools you are evaluating can help you combine those results into a single report that you can analyze and provide to your certification authority. Some tools may include features to combine results, while some may not. For tools that do include such features, the result merging feature may or may not be included in the qualification kits provided by the tool vendor. If it isn't, you would either need to qualify the feature yourself or not use it for your final run for score (see *Qualifying verification tools (DO-330)* on page 60).

Structural coverage resolution

If your project is DAL C or above, you will need to provide evidence that your requirements-based testing and analysis has covered or justified 100% of your code. Most code should be coverable by requirements-based tests, but often some code cannot be, or cannot easily be, covered by such tests. Some causes of untestable, or not easily testable code, include:

- **Coding standard** – your coding standard may require the use of untestable coding structures such as default case statements that can never be executed, or destructors.

- **Programming language** – your choice of programming language may lead to the generation of code structures that cannot be executed, such as default base class implementations in object-oriented programming languages.
- **Defensive programming** – some code may only be executable in fault conditions that cannot be tested. Typically, you will need to provide a rationale for this defensive code in your Software Coding Standards.
- **Unused library code** – a general-purpose library may include routines that are never called during your tests.
- **Initialization code and shutdowns** – some code executes before you can set up the testing environment, making it difficult to test directly. Other tests may need to invoke resets of the target hardware during the test (e.g. to check that non-volatile memory is used correctly on a warm restart first pass), making it hard to record test successes and associated structural coverage.

To comply with DO-178C guidance, a justification must be provided for any code that hasn't been tested at the relevant coverage granularities for the DAL. This activity must include a manual component, but it can be supported by features of commercial structural coverage analysis tools that let you assign a rationale for why specific sections of code cannot be tested and export these results along with those for coverage achieved through testing. This applies across all types of source structural coverage (function entry and function exit, decisions, statements, and MC/DC).

Even in the best planned DO-178C project, implementation changes are often needed after initial rounds of verification, when these justifications are often applied. A major consideration when planning how you will handle untestable code is how much extra effort implementation changes will cause. Some commercial structural coverage analysis tools include features that can mitigate the extra effort needed by helping you migrate any justifications that still apply to the new location in your code and automatically discarding justifications that no longer apply.

Justifications for coverage resolution

Structural coverage analysis tools can help with structural coverage resolution by letting users apply justifications to areas of the code that haven't been tested. Tools such as Rapi**Cover** can also help reduce review effort when code changes by supporting the migration of justifications to new code elements.

Verifying additional code introduced by compilers

TIP

If you are working on a DAL A project, you need to verify any additional code introduced by your compiler. Hence, it makes sense to mitigate the effort needed to verify additional code introduced by your compiler by:

- Investigating the behavior of your compiler given a range of compiler options and programming language constructs that you expect you may want to include in your project. This is known as compiler verification, and verification services suppliers may be able to help you with the investigation.
- Having completed this investigation, selecting programming language standards that limit the generation of additional code by compiler.

Compilers transform source code into equivalent object code. Along the way, they can add and remove code as needed. For example, they can perform optimizations to make best use of the facilities offered by the target CPU and platform, which typically removes branches, avoids repeated calculations, or creates copies of object code (inlining) instead of creating calls to existing object code. Compilers can also add new code structures that have no equivalent in the original source code. For example, additional code may be generated that includes loops to copy data in and out of functions, decision structures for array bounds checking, or calls to library functions to implement operators (e.g. modulus) that are not supported in a CPU's native instruction set.

If your software is DAL A, then you must verify the correctness of any additions made to your code by your compiler. This is covered in objective 6.4.4.2.b of DO-178C. Such verification can include analysis, review, and testing as necessary:

- **Analysis**: to identify the combinations of code constructs and compiler options that you use that lead to the insertion of additional code. This typically involves extensive review of the object code generated from sample input source files. Based on this analysis, you may decide to update your coding standard to limit the use of some language constructs. Accordingly, you should perform this analysis early on to ensure that you are aware of the behavior of your compiler environment before verification begins in earnest.
- **Review**: for example, to identify whether you are following your coding standard and keeping the analysis up to date whenever you change compiler settings, or to show that all added code falls into the categories that you identified in your analysis.
- **Testing**: to demonstrate the correctness of the generated additional code structures that are still present in the object code. This involves derivation of suitable testing criteria depending on the nature of the additional code. For example, if the compiler uses a library to implement the modulus operator, you could provide additional testing of the library implementations and use an equivalence-class test design to derive appropriate test cases.

Verification specialists may be able to help with all three aspects of verifying additional object code.

6.9.2 Data coupling and control coupling coverage analysis

DO-178C requires evidence that the data coupling and control coupling interfaces in software have been tested. As we saw in *Considering data coupling and control coupling* on page 17, data coupling interfaces and control coupling interfaces are points in the code at which one component provides data to, or affects the choice to execute, another component. DO-248C: Supporting Information for DO-178C and DO-278A provides the following definitions for data coupling and control coupling, defining:

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

Data coupling and control coupling coverage are metrics of the completeness of testing of data coupling and control coupling interfaces in the software. By demonstrating that your testing has exercised 100% of the data coupling and control coupling interfaces in the software architecture, you are also demonstrating the sufficiency of your software-software and hardware-software integration testing.

TIP

As you'll need to document the method by which you plan to verify the data coupling and control coupling coverage you have achieved of your code in your DO-178C planning documents, you should evaluate methods early during your DO-178C planning. If you choose to use any automated analysis tools to produce coupling coverage results, you should also evaluate these early.

Data coupling and control coupling coverage analysis in DO-178C

The need to analyze coverage of data coupling and control coupling interfaces is listed in a single DO-178C objective – 6.4.4.d (Table 4).

Table 4 - DO-178C objectives related to data coupling and control coupling coverage (Source: RTCA DO-178C)

	Objective		Applicability (DAL)				Output	
ID	Description	Ref.	A	B	C	D	Description	Ref.
A-7, 8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.d	●	●	❍		Software Verification Results	11.14

❍ Objective required at DAL
● Objective required with independence at DAL

This objective applies to DAL A-C systems, and the analysis should be with independence for DAL A and B systems.

Testing data coupling and control coupling coverage

TIP

The key to efficient data coupling and control coupling coverage analysis is defining data coupling and control coupling interfaces with enough information with which to write test cases to test those interfaces in your requirements-based tests. You can then satisfy the objective for data coupling and control coupling coverage analysis by analyzing the coverage you achieve during your tests.

For DO-178C compliance, you need to ensure that your requirements-based testing has exercised 100% of the data coupling and control coupling interfaces in your code.

To make your data coupling and control coupling coverage analysis efficient, it is important to include relevant interface requirements in your software architecture description (see *Considering data coupling and control coupling* on page 17). These requirements describe the control (including sequencing, timing and concurrency) and data interchange dependencies between components. This will make it easier to write requirements for both normal range and robustness testing (see *Requirements-based functional testing* on page 33) that test the interfaces in your code. You can then collect coverage results during these tests to demonstrate that the data coupling and control coupling interfaces in your code are exercised during them. Typically, this will involve analyzing statement coverage of relevant read/write and procedure call statements.

Data coupling and control coupling coverage analysis can be achieved manually or can be supported by the use of automated tools. If you use an automated tool to support your data coupling and control coupling coverage analysis, you should ensure that the tool is flexible and able to adapt to the way you have mapped between your requirements and software.

6.10 Verifying multicore systems: A(M)C 20-193/CAST-32A

Multicore systems are becoming more widely adopted in DO-178C projects. There is a major drive to move towards using them for two major reasons:

- Their improved size, weight and power (SWaP) characteristics allow system developers to add more functionality to systems per unit volume, satisfying the increasing expectations of the industry.
- It is slowly becoming more difficult to source single core processors as the majority of the wider embedded software industry has moved towards using multicore processors due to their better SWaP characteristics.

While using multicore systems offers great benefits, it also necessitates additional verification activities (and additional planning activities) to meet the design assurance required for DO-178C certification.

[5] https://www.faa.gov /aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf

[6] A(M)C refers to either EASA AMC (Acceptable Means of Compliance) or FAA AC (Advisory Circular) documents.

While DO-178C itself offers no specific guidance on the use of multicore systems (they were rarely if ever used in DO-178C applications when DO-178C launched), it has been addressed in supplementary documents. First, supplementary guidance was given in CAST-32A[5]. This guidance is due to be superseded by A(M)C[6] 20-193 in 2021.

6.10.1 Additional activities required by CAST-32A

CAST-32A lists 10 additional objectives that should be followed when developing DO-178C software hosted on multicore systems. The first 2 objectives relate to additional planning activities:

- **MCP_Planning_1** – this objective requires that the multicore system (hardware and RTOS) is described and that development and verification methodologies are planned.
- **MCP_Planning_2** – this objective requires that the shared resources in the multicore system are identified along with any methods used to mitigate interference effects from contention on those resources. It also requires the identification of any active dynamic hardware features used on the multicore system such as DVFS.

The remaining 8 objectives listed in the document relate to verification activities:

- **MCP_Resource_Usage_1** – this objective requires that the critical configuration settings used in the multicore system are identified and documented.
- **MCP_Resource_Usage_2** – this objective requires an analysis to be run on the impact of inadvertent changes to the critical configuration settings of the multicore system occurring, and for a verification mechanism that mitigates such changes to be developed.
- **MCP_Resource_Usage_3** – this objective requires that the interference channels through which software execution time on the multicore system can be affected are identified.
- **MCP_Resource_Usage_4** – this objective requires that the resources on the multicore system are verified to meet requirements on timing behavior and that any mechanisms used to mitigate the possibility of exceeding these requirements are verified.
- **MCP_Software_1** – this objective requires the execution time of the multicore software to be verified through worst-case execution time analysis (this is in addition to the verification needed for all DO-178C projects, see *Worst-case execution time analysis* on page 43, and the objective provides additional guidance on how to perform this analysis).
- **MCP_Software_2** – this objective requires the data coupling and control coupling of the multicore system to be verified (this is in addition to verification of the data coupling and control coupling needed for all DO-178C projects, see *Data coupling and control coupling coverage analysis* on page 55).
- **MCP_Error_Handling_1** – this objective requires the error handling mechanisms in the multicore system to be verified.

- **MCP_Accomplishment_Summary** – this objective requires evidence from the verification activities above to be incorporated into a Software Accomplishment Summary (*Documenting verification results* on page 29) so as to describe how each CAST-32A objective was met.

Worst-case execution time analysis of the multicore software will in most cases be the most challenging and time and resource-consuming activity when complying with CAST-32A guidance. While the timing behavior of single core systems is deterministic, the timing behavior of multicore systems is non-deterministic as it is affected by interference as separate cores access shared resources (see below).

Multicore interference

Multicore interference occurs when the execution characteristics (such as execution time) of software hosted on a single core of a multicore platform is affected by what is happening on other cores. This can be due to, for example, one core needing to access a shared resource that is currently in use from a task running on another core. Multicore interference makes execution of software non-deterministic and can greatly impact software execution characteristics including timing behavior.

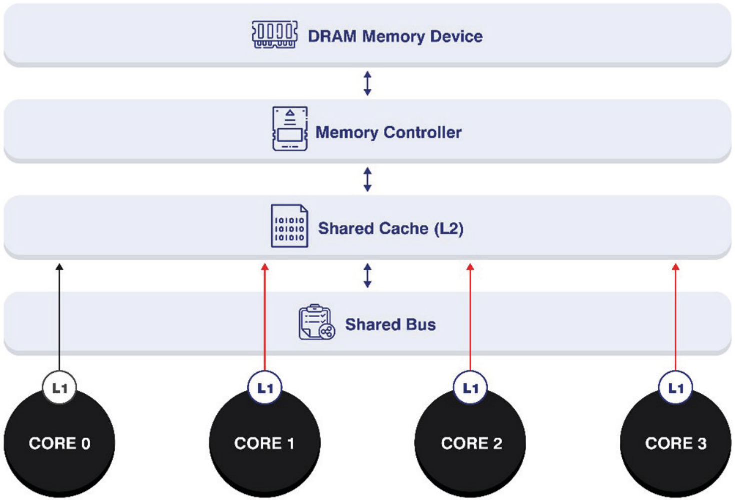


Figure 15 – Multicore interference affects software execution characteristics – the execution of tasks running on Cores 1-3 can be affected by Core 0, which is currently using the shared L2 cache

6.10.2 What's in store for A(M)C 20-193?

While A(M)C 20-193 has not yet been published, the objectives of the new document are expected to largely follow the guidance provided in CAST-32A, while also providing insight from the certification authorities on approaches that can be used to comply with the objectives.

6.10.3 Complying with CAST-32A / AM(C) 20-193

TIP

If you begin a multicore DO-178C project and need to comply with CAST-32A / AM(C) 20-193, it makes sense to consider the following:

- How can your choice of hardware and RTOS help you mitigate interference?
- How will you design your software architecture to mitigate interference?
- How will you perform worst-case execution time analysis on your multicore system?

Due to the challenging nature of compliance with CAST-32A objectives, the large number of tests that must be run for compliance, and uncertainty in the scope of how much testing must be performed, compliance with the additional objectives needed to follow CAST-32A can be very expensive. Due to this high potential cost, it pays to make multicore verification as efficient as possible.

As is the case with single core systems, decisions on which hardware to use will have an impact on the cost of multicore certification. In recent years, silicon manufacturers, board suppliers and RTOS vendors have been developing and commercializing new mechanisms that can help to mitigate the effects of multicore interference.

Discover multicore verification

You can learn more about CAST-32A objectives and strategies for multicore verification at rapitasystems.com/cast-32a.

The design decisions you make will also affect the worst-case execution time and predictability of timing behavior of your software. This includes, for example, your allocations of software functionality into components and of hardware I/O communication into software components.

Worst-case execution time analysis is likely the most challenging part of CAST-32A / A(M)C 20-193 compliance. You will need to evaluate potential methods of doing this analysis early in your DO-178C project. In recent years, methods have arisen to perform this analysis, which use interference-generating applications to stress specific shared resources on the multicore system so "worst-case" timing behavior can be analyzed.

6.11 Qualifying verification tools (DO-330)

TIP

As you'll need to describe how you plan to qualify any tools you use that replace or mitigate DO-178C processes and for which you don't manually verify the output in your Plan for Software Aspects of Certification (PSAC), you should consider tool qualification early during your DO-178C planning. If you plan to use any commercial verification tools that need to be qualified, you would benefit from finding out whether qualification kits are available for them.

As per DO-178C, you need to *qualify* any software tool you use that replaces or mitigates any DO-178C process and for which the output is not manually verified. The qualification process ensures that such software tools can be relied upon to produce appropriate and repeatable results.

DO-178C itself describes when a tool must be qualified, but does not go into detail on how this should be done. The *DO-330: Software Tool Qualification Considerations*[7] supplement to DO-178C expands on this guidance by defining corresponding objectives for the specification, development and verification of qualified tools.

6.11.1 Tool qualification levels

DO-178C defines 3 sets of tool assessment criteria which, when combined with the DAL level of your software, are used to classify tools at one of 5 different Tool Qualification Levels (TQLs) as shown in Table 5.

Table 5 - Assessment criteria for tool qualification levels (Source: RTCA DO-330)

Criteria	DAL			
	A	B	C	D
Criteria 1: • output is part of the airborne software • could insert an error	TQL-1	TQL-2	TQL-3	TQL-4
Criteria 2: • could fail to detect an error • elimination or reduction of other processes	TQL-4	TQL-4	TQL-5	TQL-5
Criteria 3: • could fail to detect an error	TQL-5	TQL-5	TQL-5	TQL-5

For example, a code generator tool that converts an architectural description of the software into package or class structures fulfils criteria 1. Verification tools typically fall into Criteria 3 (and are thus classified at TQL-5) as they neither create airborne software nor eliminate or reduce any processes other than the ones for which they are intended. Criteria 2 typically applies in cases such as model-based testing with a qualified code generator. In this case, the task of verifying the generated code is eliminated or reduced in favor of testing the model, and so the model-based testing tool meets criteria 2.

[7] The DO-330 document is available for purchase from various sources online, including https://standards.globalspec.com/std/1461615/RTCA%20DO-330

6.11.2 Tool qualification for commercial tools

TIP

If you plan to use commercial verification tool(s), you should consider:

- Is a qualification kit available for the tool?
- Are all of the tool features that you want to use included in the tool qualification kit?

What additional support and services are available to help complete the tool user activities?

If you use any commercial verification tools to automate DO-178C verification processes and don't plan on manually reviewing output from the tools, they will need to be qualified at the appropriate tool qualification level.

DO-330 defines some tool qualification activities that must be performed by the tool developer and some that must be performed by the tool user (you).

Many commercial verification tools have supporting qualification kits, which include evidence needed to demonstrate that the activities the tool developer must perform have been performed. Generally, not all features of verification tools are qualified. For each feature you intend to use and for which the way you intend to use it would require tool qualification, you should check with the tool developer whether the feature is included in the qualification kit. All qualification kits should include all of the evidence needed from the tool developer. Some qualification kits may also include supporting material to help meet tool user objectives. It may pay to ask tool vendors what the scope of their qualification kits is and how they can help you qualify the tool.

6.12 The Verification milestone (SOI#3)

Similarly to SOI#2, SOI#3 should be conducted with the certification authority when there are examples of each of the verification artifacts available for review. SOI#3 focuses not only on test cases and test procedures, but also on test results and coverage analyses (see *Requirements-based functional testing* on page 33 *and Structural coverage analysis* on page 46).

You do not need to complete the verification of your software before SOI#3, but this milestone is usually reached after:

- Around half or more of the total expected test cases and procedures have been developed and reviewed
- The approach you'll take for verifying your verification process (e.g. structural coverage, data coupling and control coupling coverage) is demonstrable (ideally with at least sample data being available)

- The approach you'll take for verifying non-functional properties of your software, including resource usage, is demonstrable (ideally with at least sample data being available)

Similarly to SOI#2, the review will examine traceability information and also examine how changes have been managed throughout the life cycle of your project.

6.13 The final run for score

TIP

As you prepare for your final run for score, you should ensure that all of your requirements-based tests are passing and that you have 100% coverage. Doing so will make your final run for score go more smoothly.

Did you know?

For large projects, running a final "run for score" may take weeks or even months.

As you approach the end of the Verification stage of your project, you'll need to prepare for your final run for score, which will produce comprehensive verification results and compliance evidence for your final software baseline in the context of your final product. This means that the test environment should be the same as the final flight system, including all hardware and with all software components integrated. The final run for score is typically performed after the Verification milestone (SOI#3), but is necessary to produce the compliance artifacts you'll need for certification.

The most efficient strategy when preparing for the final run for score is to ensure that when you go ahead with it, you'll collect all of the results you need to. This includes results for all of your requirements-based tests and the evidence you need to show that you have achieved 100% data coupling and control coupling coverage and structural coverage of your final code (through either tests or manual analysis and justification) at the required coverage granularities for your DAL.

Your requirements-based test results should be collected from the final code as it will be written in your product. This means that, if you have instrumented your code to support coverage analysis, you will need to run your requirements-based tests on code that has not been instrumented and collect your coverage results separately.

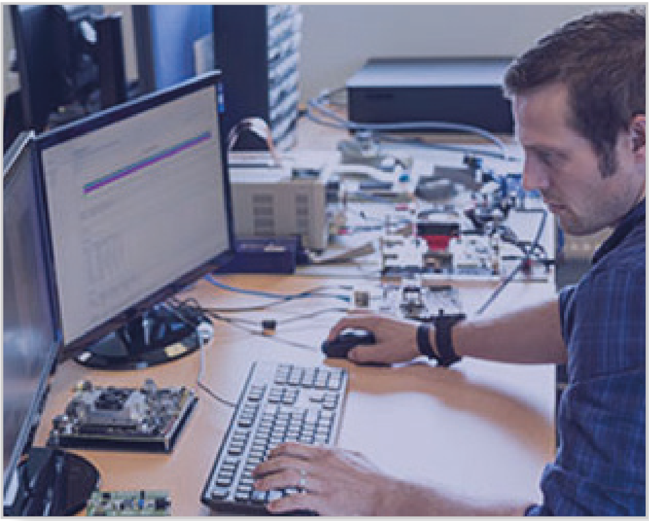
7. The Certification milestone (SOI#4)

The final SOI is the Certification review (SOI#4). This review typically follows the Quality Assurance Conformity Review and is focused on ensuring all compliance evidence is complete and correct.

A fundamental part of each of the SOI reviews is an examination of the configuration management processes and baselines and the evidence that quality assurance has been active during the design assurance process. SOI#4 looks back at each of the previous SOI reviews and ensures that all observations and findings have been resolved to the satisfaction of the certification authority. The review also ensures that the final design and deliverable documentation are complete and correct with respect to the design baseline (see *Baselines* on page 9).

SOI#4 is more than just a double-check of the release baseline – it is a detailed examination of all the development and verification artifacts to ensure that a complete and consistent design baseline is documented. This may involve the certification authority asking to see you building and loading a specific baseline, running tests against that baseline to produce the same results you reported previously, and performing desk reviews of documents and face to face audits. During the SOI, you may be asked to re-review any artifact or evidence produced during the whole life cycle of your project, so it's important to make sure that all of your life cycle data is kept up to date with the current baseline through all stages of development and verification.

8. The Authors



Rapita Systems provides on-target software verification tools and services to the embedded aerospace and automotive electronics industries. Its solutions help to increase software quality, deliver evidence to meet safety and certification objectives (including DO-178C) and reduce project costs. With offices in the UK and US, it serves its solutions globally.

Daniel Wright

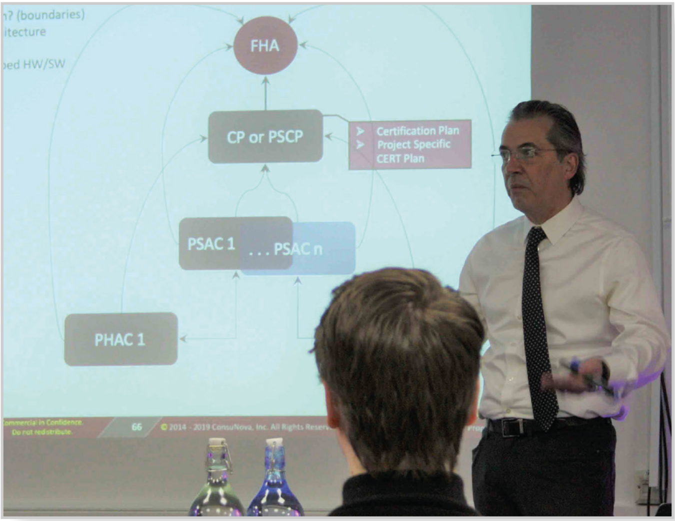
Technical Marketing Executive

Daniel's roles at Rapita Systems include creating and curating technical marketing content, including collateral, blogs and videos, and capturing and maintaining data that is used to further develop Rapita's verification solutions to best meet the needs of the global aerospace market. Daniel received a PhD in Structural Biology from the University of York in 2016.

Zoë Stephenson

Head of Software Quality

Zoë has over a decade of experience in developing and delivering DO-330 tool qualification evidence for Rapita's flagship verification solution, the Rapita **Verification Suite** (R**VS**), as well as experience managing verification projects including compiler verification. Zoë received a PhD in Computer Science from the University of York in 2002 and has worked on academic research projects with a number of industrial partners including Esterel, BAE Systems, Rolls-Royce, Goodrich Control Systems and Aero Engine Controls.



RAPITA
CONSUNOVA
2.5-Day Avionics Training
DO-178C, DO-330 & Multi-core Compliance
York, UK – Feb 2020

ConsuNova is a leading global provider of certification, compliance engineering services and solutions for safety-critical systems to the aerospace and defense industries. It provides fast, optimized and cost-effective solutions for ARP 4761, ARP 4754A, DO-200B, DO-254 and DO-178C compliance, including gap analysis, process optimizations and DER or CVE services, compliance liaison services, compliance training courses and template documents and checklists for certification.

Martin Beeby

Head of Advanced Avionics Systems

Martin has been working in Software, Hardware, and Systems development for over 30 years. He has worked on a wide range of avionics applications including Engine Controllers, Display Systems, Communication Systems and Cabin Systems. He also participates in industry working groups developing new guidance and was part of the working group that defined DO-178C.

Get in touch with Rapita Systems

rapitasystems.com

info@rapitasystems.com

+1 248-957-9801 (US)
+44 (0)1904 413945 (int.)

Get in touch with ConsuNova

consunova.com

team@consunova.com

+1 858-444-6762

About Rapita

Rapita Systems provides on-target software verification tools and services globally to the embedded aerospace and automotive electronics industries.

Our solutions help to increase software quality, deliver evidence to meet safety and certification objectives and reduce costs.

Find out more

A range of free high-quality materials are available at:
rapitasystems.com/downloads

SUPPORTING CUSTOMERS WITH:

Tools	Services	Multicore verification
Rapita **Verification Suite**:	V&V Services	CAST-32A Compliance
Rapi**Test**	Integration Services	Multicore Timing Solution
Rapi**Cover**	Qualification	
Rapi**Time**	SW/HW Engineering	
Rapi**Task**	Compiler Verification	

Contact

Rapita Systems Ltd.
Atlas House
York, UK
YO10 3JB

+44 (0)1904 413945

Rapita Systems, Inc.
41131 Vincenti Ct.
Novi, Mi, 48375
USA

+1 248-957-9801

rapitasystems.com

linkedin.com/company/rapita-systems

info@rapitasystems.com