DDC-I

WHITE PAPER

How to Certify Your Code Once, And Use it in Multiple DO-178 Applications

Overview

It's no secret that developing DO-178 certifiable avionics software is an expensive and time consuming proposition. Compounding matters further, avionics companies have had to historically contend with three key factors that can make the difference between a successful or failing avionics program or product line:

1. Late software changes in a program's development
2. Ability to add new software capabilities (without impacting other certifiable software)
3. Reuse of software modules with their certification credits

Once a mission critical software system has been developed, verified, and fielded; developers, managers and customers typically enter a paranoid state where changes to the system are fiercely resisted. Why do may companies suffer from these issues? Often it's because in anything but a trivial software system, opening up software code to remove a defect or add a feature invariably exposes the entire software system to suffering a fresh round of new or yet to be discovered defects during the testing, verification, and certification processes. This same issue restricts software and certification reuse and stunts the ability for avionics products to be modified or adapted for new aircraft or feature enhancements.

A well proven means to break software change paralysis is to use technology that isolates software components. By using this technology a change to the defect prone or feature deficient area can be made, while leaving other software components identical to the day they were tested and accepted. Deos™, a time and space partitioned RTOS, certified to DO-178, level A since 1998, includes such technologies. In fact Deos certification reuse and its ability for users to minimize software impacts is a key differentiator Deos holds relative to other partitioned COTS RTOSs. These features allow mission and safety critical systems to evolve and innovate at a faster rate and lower cost. This paper describes some of the key Deos technologies that enable this capability.

Reusable By Design

Deos is designed to enable application software component portability, where binary (not just source code) reuse is the ultimate form of portability. Deos was originally developed for use in the aerospace industry where verification and certification costs are notoriously high. The ability to reuse executables and shared libraries without modification on new (compatible) target systems significantly reduces costs. Reuse of software is not a new concept but in the avionics market it is can provide even more value, especially if software modules (executables and shared libraries) do not change from product to product. Minimizing change is the key to driving the cost saving to a maximum.

With Deos this is accomplished with the use of the following key technologies:

- DO-178C, Level A, dynamic loader/linker
- Well controlled software component interfaces
- Component based documentation and XML configuration files
- No artificial constraints on resources
    - No magic numbers in code
    - API parameters in XML configuration files, not in the code

Binary Re-use and Dynamic Linking

A key differentiator is the Deos DAL-A dynamic loader/linker. This is a critical element in enabling binary modularity and the reuse of DO-178 certification credits. Deos software applications as well as the Deos operating system itself are made up of individual executables and run-time linkable libraries. Each is separately compiled and linked into stand-alone executable files. The Deos DAL-A dynamic loader/linker loads executables and dynamically links them to their required run-time libraries as part of application start-up (dynamic linking). This process of dynamically linking enables a software system to be decomposed into a collection of individual software components with the following characteristics:

- Each executable and run-time library has its own individual cyclic redundancy check (CRC).
- Dynamic linking the executables and libraries does not change the CRC of the executable or the run-time library.
- Software components are not dependent on an unqualified static linker which injects the library object code into the image, breaking its CRC, and necessitating re-verification. Or worse, a qualified linker that leaves you stuck with a specific version of tools.

Deos then executes each application (comprising or one of more of these executables) in its own separate memory protected virtual address space (space partitioning).

The picture below shows an example system with separate applications made up of the various software component binary images. Each (color) shaded block in Figure 1 represents a separate binary module, or executable.

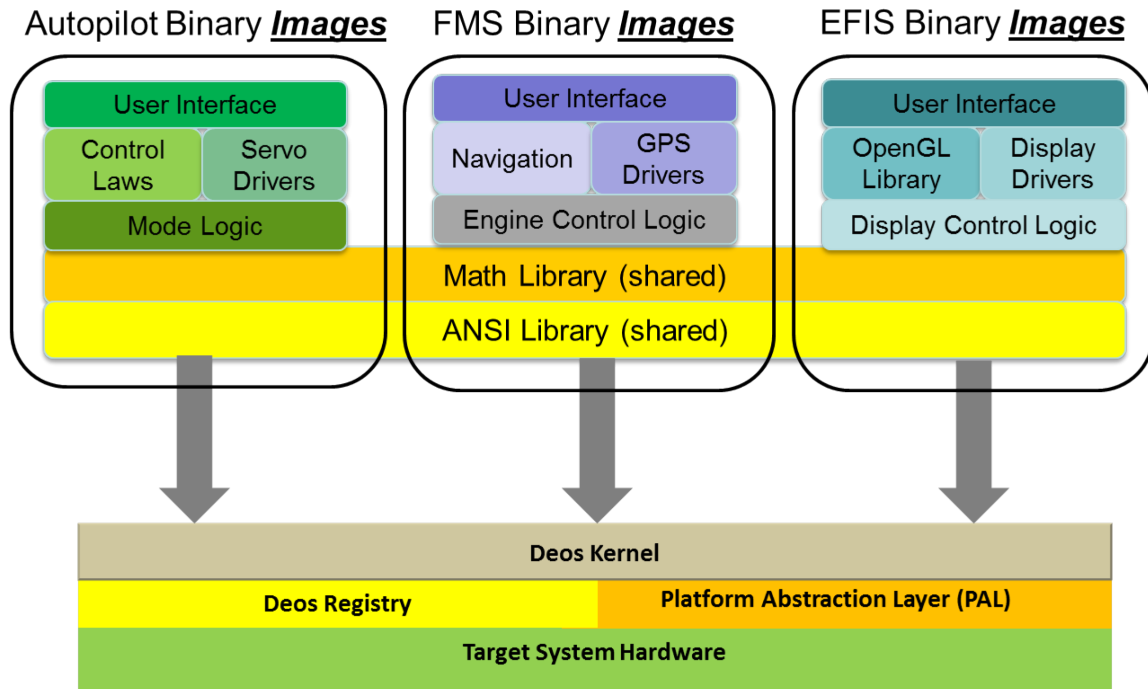

Figure1: System composed of separate executables with run-time linkable binaries.

By decomposing a software system into discrete components, one can limit component couplings to well defined interfaces. This provides the advantage of limiting interaction between software components, and thus minimizes the change impact to one when another is altered. A well-controlled inter-component coupling helps create a much smaller control and data coupling issue than if all the source code were in one large executable binary that executes in a single address space.

In addition, for software systems whose lifecycle extends beyond a particular target, decomposing the software into components that *are* and *are not* platform dependent is advantageous. Consider a network stack application where the *stack* portion of the logic is one component and the portion that accesses the ethernet hardware is another component. The *stack* portion could be reused on another target, possibly in binary form if that other target has a compatible CPU.

Dynamic Linking vs Static Linking

Even though Deos enables one to design, implement, and test within the paradigm of discrete software components, one can still share and re-use software implementations. As an example, a floating point math library is a classic example of software that may be shared amongst a suite of hosted software applications. There are two primary approaches to share software modules.

Static Linking

One way to share math library code is to link the library to the using software components at compile/link time. In other words, link all the component's code and data together with the math library code injected on the workstation and then load the resulting binary onto the target. With this scheme each component contains a copy of the math library.

One pitfall of static linking is the propagation of faults into all of the executable images that use it. In the aviation software domain, this pitfall is exacerbated by the fact that once an executable image is changed, even if that change is just a re-linking with an updated version of a static library, the executable image must be re-verified. This is generally driven by a distrust of workstation linkers that might not be performing as intended. As a result, static linking can be expensive and can contribute towards increased change impact.

Dynamic Linking

Another way to share the math library code is to use *dynamic* linking. The math library, being a separate binary executable, would then be independently re-verified and then could be re-loaded onto the target. The issue with the suspicious linker is gone, as Deos contains a DO178C Level A verified run-time linker that would then link the re-verified math library to all software components needing it.

In short, the operating system is a DO-178C Level A verified linker (in other words, it can be trusted). Rather than injecting code into a large binary whole, it connects together the interfaces leaving the binary partitions integrity and pedigree intact. It does this at run-time, instead of establishing the coupling at build time and performs this service independent of linker version, switches or language constructs used. All of this helps break change paralysis by easing software integration & field updates.

In the simplified picture below, we see that it is the Deos Operating System that performs the linking and manages the connections for each application. Each application has its own virtual address space (or partition) into which its binary images are mapped and linked together by the operating system. Note that multiple applications can share the same library (for instance, the math library), without duplication of code and while maintaining the required partition protection from the other application using this shared library. In other words, while they share the same read-only *view* of the code, they each have their own read-write *copy* of data.

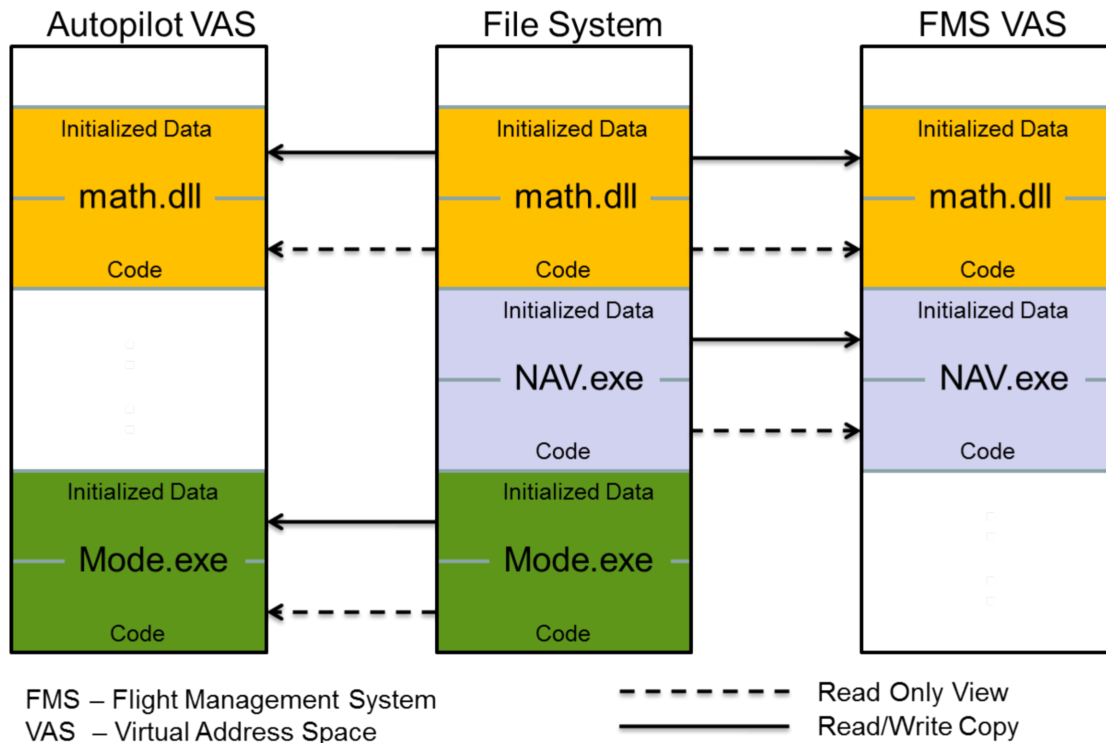

Figure2: Partitions protected from multiple binary users

Component based Documentation

Mission critical software, such as that used in aviation, is accompanied by a plethora of documentation. This documentation includes software requirements and design information, software test documentation, and software development planning documents. It is often the cynical claim that high design assurance aircraft software isn't done until the paperwork equals the weight of the airplane.

Because Deos allows you to break software systems down into a collection of separate binary components, it also becomes possible to separate the software system's documentation along similar lines. Thus changes limited to a particular software component only affect that component's documentation. In short, Deos allows you to pursue a reusable software component philosophy where everything about the software can be reused, including the documentation, reviews, tests, etc.

Figure 3, continuing with our example autopilot application, shows how a set of component binaries and associated documentation make up a package. All these packages go together to provide the complete system. Note that the operating system follows this same design and is a collection of packages.

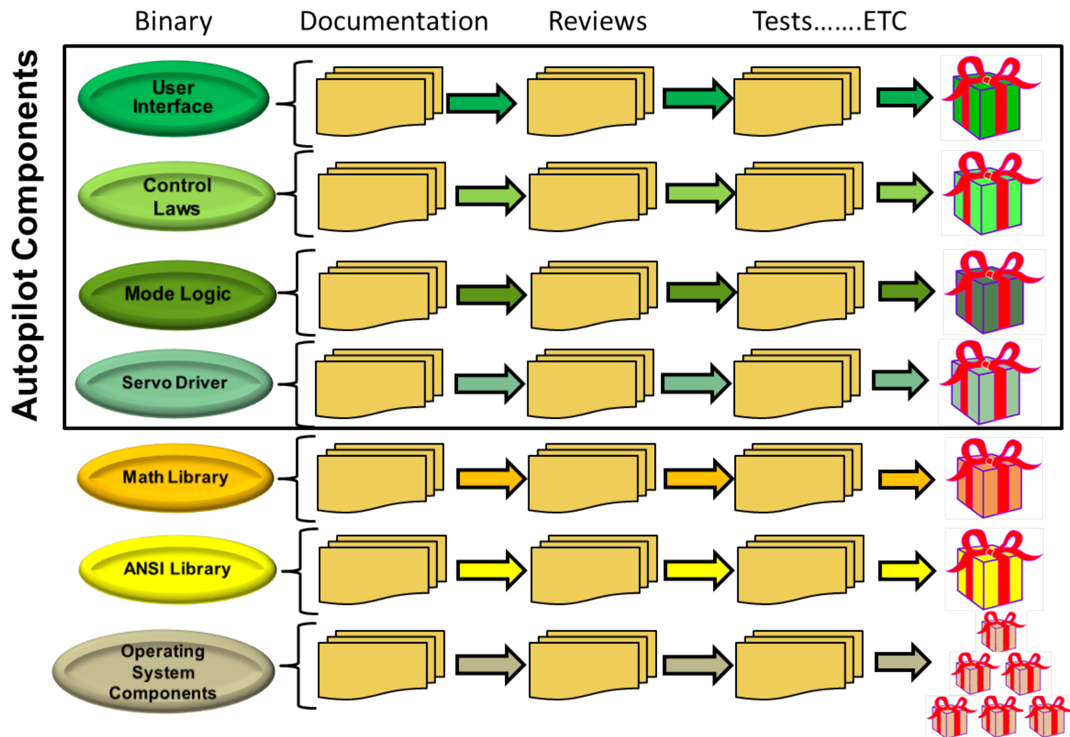

Figure3: Autopilot components and associated documentation

XML Configuration Files (No Magic Numbers in Source Code)

When developing software for reuse, common sense dictates caution when directly or inadvertently injecting artificial constraints into the logic. If a software developer reasons that "no one will ever use more than 16 foobars," that belief could permeate the design, and inevitably, the constraint will become another reason against re-use. Deos has been designed with a "no magic number" philosophy. The constraints that Deos has (e.g., the number of semaphores to support) are user configurable by the system integrator via an XML based platform configuration file that ultimately is included in the *Deos registry.*

The *Deos registry* is a single binary file that defines the personality of the entire system running the Deos kernel. It defines how the operating system should behave on a specific target as well as all the included drivers, applications and libraries. The behaviors defined include a list of processes to automatically start on target power-up, access control lists for hardware devices, I/O routing, time and space partitioning boundaries, and a variety of other Deos and application software configuration data. The registry is the place where *magic numbers* are deposited. In this location, they are separate from any executables and easily changed with provided workstation tooling.

A quick glance at the Deos Kernel User Guide reveals that the Deos API is intentionally devoid of parameters that are directly or indirectly platform specific. For example, when creating a thread, the API will not contain information about a CPU budget, the execution rate, or how much stack space to associate with that thread. Instead, the API accepts an

ASCII string, which is used to reference the desired set of parameters configured in the application's configuration file and stored in the Deos registry.

Capturing component magic numbers within the registry has significant benefit for portability and reuse. If a software component is ported from one target system to another, it is unlikely the two platforms will have identical performance characteristics. Porting the software component could be as simple as assigning new thread execution time budgets in the configuration file to reflect the new target's execution performance.

Porting is not the only reason to need Deos registry modification. Consider an executable that uses a run-time linkable library. In the course of development, a new version of the run-time linkable library is released and in doing so, the library's data segment is larger and the worst case stack usage for an exported function is larger. The software component that uses the run-time linkable library may break due to insufficient RAM quota for the new data segment size, and it may break with a stack overflow fault. These problems can be easily solved by adjusting the component's RAM and stack quotas in the application's configuration file and stored in the Deos registry.

Summary

In mission critical software domains, Deos helps accelerate innovation and lower software change costs. It does this by providing the ability to decompose the software system into discrete, separately verifiable, binary reusable software components and the DO-178C Level A runtime dynamic linking capability of these components. This capability along with the individual XML configuration files (Deos registry) enables the componentization of the software and certification artifacts which maximizes reuse.

Software and certification artifact reuse is a huge advantage for customers that have multiple product lines with substantial overlap in features and functions. Deos' customers can cross-leverage applications and verification evidence from system-to-system and across product lines spanning multiple aircraft while minimizing the required reverification and subsequent aircraft certification activities to integration and testing work.

Another significant challenge that avionics system designers face is how to manage the volatility of I/O. This paper has not addressed this particular challenge. Please see the DDC-I white paper "Maximizing Reuse in Safety-Critical Software using IOI" to see how DDC-I's IOI product provides a solution to meet these challenges.

In short, Deos provides tools and technologies to enable avionics to better tolerate changes late in a program's development, addition of new capabilities, and reuse of not only software source code, but the certification artifacts that support the software modules.

For Additional Information

©2015 – For details about DDC-I, Inc. or DDC-I product offerings, contact DDC-I at 4600 E. Shea Blvd., Suite #102, Phoenix, AZ 85028; phone 602-275-7172, fax 602-252-6054, email sales@ddci.com or visit http://www.ddci.com.