

Designing safety-critical avionics software using open standards

By Tim King, LynuxWorks

Safety-critical avionics systems have continually grown more complex and software-intensive. Regulatory authorities and avionics manufacturers have responded with guidance such as DO-178B and RSC, and new technology such as p-RTOS. These tools help to ensure that software performs safely, with controlled development cost.



■ Safety-critical avionics software has been evolving for decades, providing increased functionality while becoming more complex. Developers of avionics software must demonstrate compliance with guidelines such as DO-178B and must adhere to sound design principles to avoid subtle design flaws and to assure the certifiability of their software. Open-standards-based, real-time operating systems enable reuse of legacy code, and facilitate the development of more complex systems without sacrificing certifiability or real-time performance. Over the past 30 years, several major trends have impacted commercial avionics.

Digital systems have displaced analog devices and have evolved to automate increasingly complex functions. These functions range from non-critical cabin entertainment features to safety-critical automated flight controls. These avionics functions have become increasingly software-intensive, while at the same time becoming increasingly safety-critical. For example, a fault in an aircraft's flight control software can lead to a catastrophic failure condition resulting in loss of the aircraft and the lives on-board. Due to many factors (e.g., size, weight, power and cost), avionics manufacturers have been driven to integrate disparate functions onto fewer numbers of CPUs/modules. In today's so-called "integrated modular architecture" (IMA) systems, multiple functions of

differing levels of criticality are often allocated to the same CPU. For example, non-critical cabin entertainment software may be hosted on the same CPU as the highly-critical flight control software. Due to the cost of developing avionics software, avionics manufacturers have been driven to leverage reusable software. Ideally, such software is developed once and then reused many times. For example, a math library could be reused in hundreds of applications on dozens of different aircraft. As a result of these trends, five key factors now play significant roles in the development of avionics software: DO-178B compliance, brick wall partitioning, sound software design principles, open standards, and reusable software components. In the remainder of this article, we discuss these factors in more detail and how they enable avionics developers to create safety-critical software efficiently.

In the early 1980s, civil aviation regulatory authorities (e.g. FAA, EASA) recognized that avionics equipment was becoming increasingly complex, software-intensive and safety-critical. For example, relatively simple flight control functions, once performed by analog devices, were being replaced by more sophisticated digital systems. In response, regulatory authorities and commercial avionics manufacturers co-authored guidelines for developing avionics software (i.e., RTCA/DO-178). The

current version of these guidelines is known as DO-178B. These guidelines were intended to ensure that avionics developers employed a degree of process rigor during software development and verification commensurate with the criticality of the function being performed. DO-178B defines five levels of failure conditions to which software might contribute (catastrophic, hazardous, major, minor and no effect) and five corresponding levels of software design assurance (levels A, B, C, D and E). For example, modern flight control software typically requires Level A design assurance, which imposes the most stringent process rigor, since a fault therein could be catastrophic (i.e., loss of life). Conversely, a cabin entertainment system would typically require Level E design assurance and the least stringent process rigor, since a fault therein would have no effect on the safe operation of the aircraft, just a plane full of unhappy passengers.

As noted, not all software in modern avionics has the same level of safety-criticality. Further, as the level of process rigor increases, the cost to develop and verify software increases. For example, software developed to Level A design assurance can easily cost 5 to 10 times more than software developed with a relatively low degree of process rigor, say Levels D or E. Consequently, manufacturers try to minimize the amount of software categorized at higher levels

of safety criticality. And, due to factors that drive increasing levels of integration, they often host numerous software functions - with varying levels of criticality - on a single CPU.

However, this integration creates a special challenge. Specifically, how does one prevent these different software functions from interfering with one another? Solving this challenge is especially important when mixing high- and low-criticality software on the same CPU. For example if the cabin entertainment system corrupts the flight control function aircraft attitude data (i.e., orientation in space), it could lead to a hard-over condition of the aircraft control surfaces. Similarly, the cabin entertainment system could get stuck in an infinite loop, thereby hijacking the CPU and denying the flight controls time to execute. In both cases, a catastrophic event could occur. In recent years, a new class of partitioned real time operating systems (p-RTOS), such as Lynx-Work LynxOS-178, has been developed that provide the ironclad guarantees of non-interference and deterministic behavior required by safety-critical systems. A p-RTOS allows the software developer to create brick wall partitions for each function. This partitioning prevents one function from interfering with another, in terms of time (i.e., CPU bandwidth), space (i.e., memory) and other resources (i.e., I/O and other physical devices). In a sense, each partition is like a separate CPU, even though all partitions reside on a single CPU.

To enforce time partitioning, each software function is allocated a strict budget of CPU time. Using these CPU budgets, a system timer and interrupt and a schedule p-RTOS (operating in the CPU privileged mode) control the order in which functions execute on the CPU, and the amount of CPU time each is granted. If a function attempts to overrun its CPU budget, the timer interrupt fires, the p-RTOS takes control of the CPU, preempts the offending function and allows the next function in the schedule to run. To enforce space partitioning, each software function is allocated a strict quota of its own memory (e.g. RAM and stack space).

The p-RTOS uses the CPU memory management unit to enforce this partitioning by mapping virtual to physical addresses appropriately. Resource partitioning is achieved in a similar manner, wherein each function is granted explicit access (i.e., read/write, write-only, read-only, none) to resources (e.g., interrupts, I/O devices) and the p-RTOS enforces these access controls. With proper use of these capabilities, developers have a high degree of confidence that the software they develop will be free from defects wherein a fault in one function could interfere with the intended, correct and safe operation of another.

Even with a partitioned RTOS, software developers must adhere to sound design principles in order to avoid introducing subtle design flaws that could compromise partitioning and lead to unwanted interactions between partitions. Further, sound design principles help ensure the developer's ability to certify the software. Regarding design flaws, partitioning provides the software developer with a means of erecting barriers that prevent software in one partition from interfering with software in another partition. However, in many cases, software applications in different partitions need a means of communicating or interacting. RTOS features such as Posix pipes support such

interaction. But, if done incorrectly, the developer can create holes in the partitions that may lead to problems. In particular, DO-178B calls out control and data coupling as cause for concern and scrutiny. A control coupling exists when software application X can cause application Y to perform an action on the behalf of X. Similarly, a data coupling exists when X can send data to Y, presumably so that Y will read and use the X data. As long as X works properly, Y likely will too. However, if X experiences a fault, which causes it to make an erroneous request of Y or to send Y erroneous data, then the fault in X propagates to Y. These scenarios are especially problematic if X has a low-level

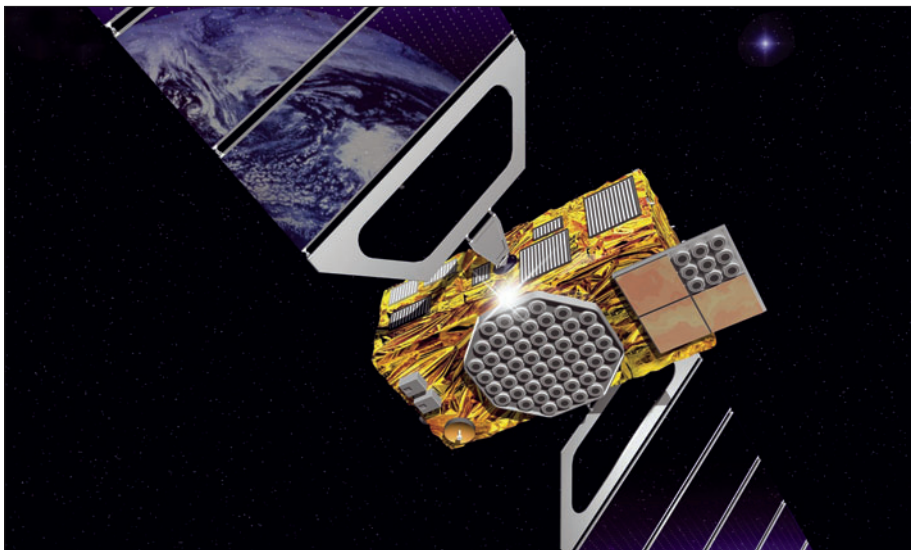


Figure 1. LynuxWorks software is deployed in space applications like Galileo.

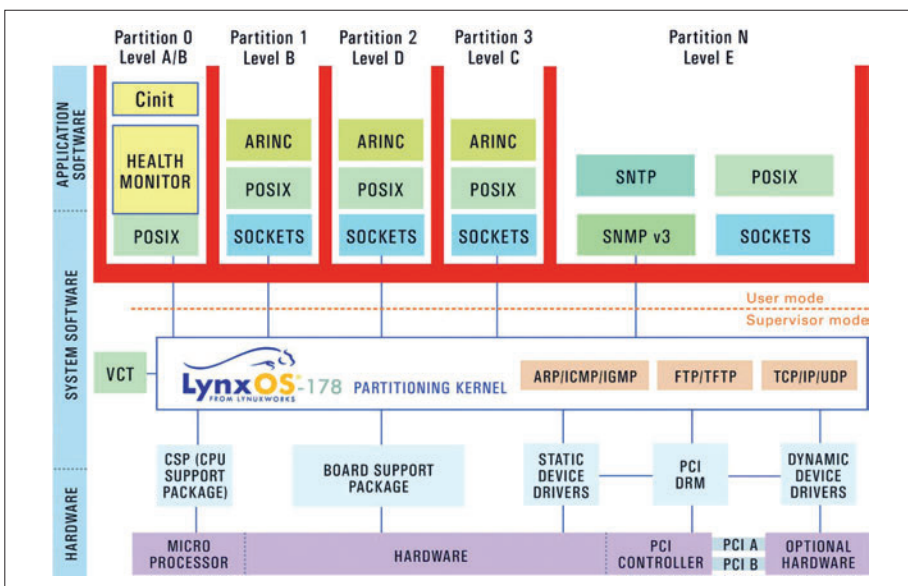


Figure 2. The architectural diagram of LynxOS-178

design assurance (say Level E) and Y has high-level design assurance (say Level A). As a result, DO-178B requires developers to inspect such couplings and evaluate potential failure conditions in X and their impacts on Y. At a minimum, the designer of Y should employ reasonableness checks and other protective measures to guard any commands/inputs received from X.

Regarding certifiability, testing plays a large role in DO-178B certification (along with reviews and analyses). In general, DO-178B encourages requirements-based tests (or black box), wherein the majority of tests in a test suite are written against the formally documented requirements of the software. In limited cases, unit level tests may be needed to drive hard to reach paths in a given piece of code (or white box). In the end, all requirements must be fully tested and a given level of structural coverage must be

attained, commensurate with the software's level of design assurance. For example, Level C design assurance requires statement coverage, such that every statement has been executed at least once. An outcome of this requirement is that every path through the code must be exercised by at least one test.

Levels A and B require more complex coverage patterns which thoroughly stress each condition/decision in the software. Essentially, these coverage requirements take a more sophisticated view of the paths that exist in the conditions and decisions of the code. A decision is a Boolean expression composed of one or more conditions and zero or more Boolean operators. For example, $((X > 0) \text{ AND } (Y < 0))$ is a decision with two conditions (one involving the value of X and one involving the value of Y) and a single Boolean operator (AND). For Level

B coverage, testing must drive this decision to take all possible outcomes (or paths) at least once (for this decision, there are two outcomes: TRUE and FALSE). For Level A coverage, testing would have to drive this decision to take all possible outcomes (as for Level B), but in addition, Level A coverage requires that each condition take all possible outcomes (or paths) at least once (for both these conditions, there are two outcomes: TRUE and FALSE). Further, each condition must be shown to independently impact the decision outcome, while holding the other condition(s) fixed (in a sense, yet more paths). Consequently, one must take care when creating paths in the software. If the code is too complex, it becomes difficult to understand and errors may not be caught by either review or test. Further, achieving the appropriate level of coverage for complex code can become very difficult, time-consuming and costly. Note that Level D imposes no structural coverage requirement. Level E imposes no testing requirement.

Open standards, such as Posix and ARINC-653 have evolved for real-time, safety-critical RTOSes. For example, LynxOS-178 is conformant with both. Such standards bring two distinct advantages. First, they embody the collected wisdom of many RTOS experts (e.g., in the case of Posix, several thousand years of expertise). This experience base ensures that the design of a standards-based RTOS rests on very solid ground. When designing safety-critical software, the more solid the ground underneath you (i.e. the RTOS), the better. Second, an open-standards-based RTOS facilitates reuse of proven legacy code, often from external sources, and enables the development of more complex software without sacrificing certifiability or real-time performance.

Driven by a desire to reduce cost and schedule, avionics manufacturers have begun to design and deploy reusable/portable hardware platforms and application software. This approach is especially attractive in markets such as commercial avionics, which require costly certification activities. For example, if the manufacturer can incur the cost of developing a software component once, along with associated certification evidence, and it can then reuse that software and evidence at a drastically reduced cost, it gains an advantage over its competitors who lack that capability. Consequently, the FAA has developed guidelines for so-called Reusable Software Components (RSCs as defined in AC20-148). This guidance requires some activities beyond those of DO-178B, but the advantage is that all certification evidence created under the RSC during one certification can be reused, as is, on subsequent certifications. Currently, LynxOS-178 is the only COTS RTOS to have been accepted as an RSC. ■