

Safety-critical Partitioned Software Architecture: A Partitioned Software Architecture for Robotic Spacecraft

Greg Horvath * Seung H. Chung † Ferner Cilloniz-Bicchi ‡

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109, U.S.A.

The flight software on virtually every mission currently managed by JPL has several major flaws that make it vulnerable to potentially fatal software defects. Many of these problems can be addressed by recently developed partitioned operating systems (OS). JPL has avoided adopting a partitioned operating system on its flight missions, primarily because doing so would require significant changes in flight software design, and the risks associated with changes of that magnitude cannot be accepted by an active flight project. The choice of a partitioned OS can have a dramatic effect on the overall system and software architecture, allowing for realization of benefits far beyond the concerns typically associated with the choice of OS. Specifically, we believe that a partitioned operating system, when coupled with an appropriate architecture, can provide a strong infrastructure for developing systems for which reusability, modifiability, testability, and reliability are essential qualities. By adopting a partitioned OS, projects can gain benefits throughout the entire development lifecycle, from requirements and design, all the way to implementation, testing, and operations.

I. Introduction

For the past year, the Safety-critical Partitioned Software Architecture (SPSA) project at NASA JPL has been investigating the use of an ARINC 653-compliant partitioned operating system¹ as the basis for the flight software systems that will power future JPL robotic exploration missions. ARINC 653 is a mature industry standard with many examples of successful application. Partitioned operating systems (OS) that are compliant with the ARINC 653 standard for Integrated Modular Avionics (IMA) provide strict partitioning in both the time and space domain. A partition contains an application, consisting of one or more threads of execution. Each partition application is allocated processor time according to a fixed periodic schedule. Additionally, each partition application executes in its own memory space, essentially eliminating the chance that the entire flight software application will be affected by a memory usage error occurring in an unrelated and possibly non-critical portion of the software. A partitioned OS offers many appealing benefits: memory protection (space partitioning), execution guarantees (time partitioning) and improved fault containment capabilities.

Despite the fact that we are investigating a particular OS technology that, on the face, may seem to address only lower level software implementation concerns, we believe that by adopting a partitioned OS, projects can gain benefits throughout the entire development lifecycle from requirements and design, all the way to implementation, testing, and operations. The choice of a partitioned OS can have a dramatic effect on the overall system and software architecture, allowing for realization of benefits far beyond the concerns typically associated with the choice of OS. Specifically, we believe that a partitioned operating system, when coupled with an appropriate architecture, can provide a strong infrastructure for developing systems for which reusability, modifiability, testability, and reliability are essential qualities.

*Software Systems Engineer, Flight Software Applications Group, 4800 Oak Grove Dr., Pasadena, CA 91109, U.S.A.

†Software Systems Engineer, System Architectures & Behaviors Group, 4800 Oak Grove Dr., Pasadena, CA 91109, U.S.A., AIAA Senior Member

‡Software Engineer, System Architectures & Behaviors Group, 4800 Oak Grove Dr., Pasadena, CA 91109, U.S.A.

I.A. From Federated to Integrated Modular Avionics Architecture

At the early stage of today's aviation era, avionics were developed using federated architecture. Within a federated avionics architecture, a computing resource is dedicated to each functional module. One of the benefits of federated avionics is that the architecture minimizes the interactions among the functional modules, thus maximally separating various architectural concerns. Separation of concerns can enhance certain essential architectural qualities, such as verifiability, reliability and operability. On the other hand, as systems becomes more complex, the number of functional modules also increases. Accordingly, the number of dedicated computational resources also grows, leading to increase in mass, volume and required power. In practice, a federated avionics architecture is difficult to scalable.

One solution to the scalability concern is to share a common computing resource. Such an integrated avionics architecture, however, also mixes the concerns among modules, thus potentially degrading certain essential architectural qualities. A solution is an integrated modular avionics (IMA) architecture, in which a common computing resource is shared among functional modules, but the modularity among them is maintained. As a result, an IMA architecture has similar benefits of a federated architecture, but without the scalability problem. This IMA architecture has been in use since early 90's within both military and civil avionics. Boeing's 777 and 787, and Airbus's A380, are only some of examples of IMA architecture in use. In the case of Boeing's 787, use of IMA has reduced 2000 lbs.² More detailed comparison between federated avionics and IMA,³ and some of important considerations for the transitioning from federated avionics to IMA^{3,4} are available. Over the years of successful implementations of IMA, ARINC 653, an industry standard for IMA architecture-based OS has been developed.

I.B. ARINC 653 Overview

As previously described, an ARINC 653 compliant OS strictly partitions applications in both time and space. As illustrated in Fig. 1, ARINC 653 compliant OS consists of multiple partitions on top of a Core OS. The Core OS is responsible for managing shared resources, including processing time scheduling among partitions and shared services. Each partition contains an application, consisting of one or more threads of execution. The Partition OS provides the usual services of an OS to the application.

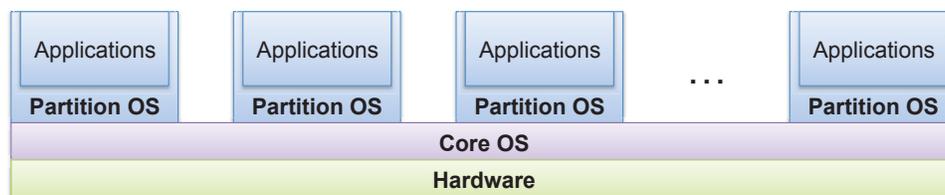


Figure 1. Structural patterns of ARINC 653 RTOS

Fig. 2 illustrates an example of the space and time partitioning among four partitions. Each partition application executes in its own memory space, essentially eliminating the chance that the entire flight software application will be affected by a memory usage error occurring in an unrelated, possibly non-critical, portion of the software. In Fig. 2, space partitioning is represented on the vertical axis, where the vertical length depicts the size and location of the memory used by each partition numbered from one to four. As depicted, each partition only uses the specified range of the memory space. The Core OS allows each partition application to execute according to a fixed periodic schedule using preemptive scheduling scheme. In Fig. 2, the time partitioning is represented on the horizontal axis. Green boxes represent the used processing time that is scheduled periodically to each partition. Through preemptive scheduling, the partitions are guaranteed of their fixed periodic processing time. White boxes represent the unused processing time. If desired, priority preemptive scheduling can be used to recoup the underutilized processing time. As illustrated with purple boxes, other partition applications could be allowed to use these underutilized processing times based on their priorities.

The benefits of using ARINC 653 compliant OS are summarized below:

Strong Memory Protection: Scope of any memory corruption is contained to a single partition

No CPU Hogging: All partitions are allocated CPU resources at design time

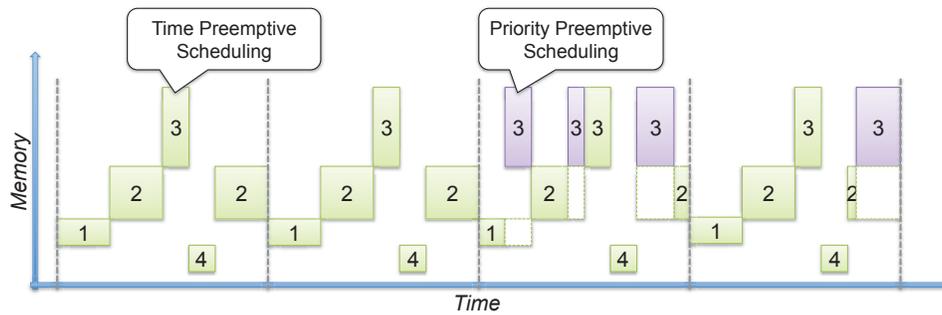


Figure 2. Time and space partition example of ARINC 653

Mixed Criticality Partitions: Each partition can contain code of different criticality or maturity

Easier Integration & Testing: Well-defined interfaces allow for easier segmentation of work

Despite the benefits, there are several issues that arise as a result of adopting a partitioned OS. The choice of how applications are allocated to partitions can have a significant impact on system performance as well as the degree of fault containment that can be guaranteed by design. Additionally, scheduling of partitions for execution can be a nontrivial task. These issues can be mitigated by software architecture, which allows for rapid modification of the system partitioning with minimal impacts to the code base.

I.C. Partitioned OS for Robotics Spacecraft

Like the aviation industries, both NASA and the European Space Agency (ESA) have also been studying the potential of ARINC 653 based IMA for the space domain. NASA has baselined an ARINC 653 compliant OS for Ares and Orion of the Constellation program.⁵ The challenges and the architectural concerns of the Constellation program are similar to what the aviation industries had faced over the years, challenges that have led the change from federated avionics architectures to an IMA architectures. Such change has been sparked mainly by a desire to minimize mass, volume and required power, but without sacrificing the safety properties required for humans on board.

The robotic spacecraft domain, however, differs from that of manned vehicle domain in that the avionics is already integrated, though not modular in a partitioned OS sense. While the safety of the system is also critical for robotic spacecraft, the effort and cost associated with human rated criticality cannot be required from robotic missions. Nonetheless, mission success is critical, and the growth of software complexity is nearing the limit of our ability to develop and manage using current approaches. Some of the concerns are integration and interoperation of mixed criticality level software and their reliability. While we can speculate the benefits of using ARINC 653 standard, due to the uncertainty associated with the cost and effort of adopting new OS, we must investigate the added benefits and the imposed consequences. We are concerned with additional costs and effort associated with new development and build process and methods, but more importantly we are concerned with the architectural impact on the lifecycle development and operations of software.

ESA has also studied the benefits of ARINC 653 for robotic spacecraft domain.⁶ Their study focuses on various benefits from OS feature perspective and the technology gaps that must be filled before fully adopting a partitioned OS. While we have been studying similar aspects, this paper adds a new architectural perspective that covers the entire lifecycle of robotic missions. More specifically, we identify relevant quality attributes of the resulting architecture, map quality attributes to lifecycle development activities, and identify key architectural features to support the desired quality attributes.

II. Flight Software Quality Attributes

Quality attributes, or QAs, are non-functional requirements on a system. QAs can capture run-time or static properties, business requirements and the like. Partitioned software architectures are uniquely able to support and enhance the following QAs: Reusability, Modifiability, Testability, and Reliability. In the

following section, we will discuss briefly what we mean by each of these QAs, as well as how a partitioned system based on the ARINC 653 specification can support each of these QAs.

II.A. Reusability

Throughout the aerospace industry, there is a continual push to do missions with significant science return for increasingly lower cost. As the budgets for exploration are squeezed even tighter, being able to leverage prior work for future efforts is nearly essential. With software taking up a larger share of the overall project budget, it is natural to look for ways to save in the software effort. Over the years, there have been many attempts to enable effective software reuse, from object-oriented analysis and design to product line architectures. Each has their own strengths and weaknesses, but ultimately a successful reuse strategy is more about what works for the organization. For this reason, we believe that the partitioned OS technology can support reusability due to its requirements for strong interface definition. In an ARINC 653 system, all inter-partition communication must take place over statically defined ports and channels. Ports can be configured for a queueing or sampling discipline; for sampling ports, an additional parameter indicating the update period must be specified as well. The standard practice of functional decomposition, which is used to break down a complex system into a set of manageable tasks, requires careful management of interfaces between the various functional elements. There is then a natural extension from interface management at the system level to specification of concrete interfaces between various functional partitions at the software implementation level which allows us to leverage existing analysis principles to the design of the functional software partitions as well.

As a result of the rigorous interface definitions required by the partitioned OS platforms, the task of analyzing the applicability of existing software for a new project is significantly enhanced. Message content and formats – and possibly timing specifications – for a partition application are explicitly stated, not implicit in the code. We can then determine the suitability, at a gross level, of a partition application for use in a new system by examining the data and timing requirements of its input and output ports. When combined with a suitable behavioral specification of the application, one can get close to the ideal of a ‘plug-n-play’ style component which can be dropped into a new application to fulfill a piece of the overall functionality of the system.

II.B. Modifiability

Often during development and testing, it is necessary to modify the composition or layout of the software to account for performance concerns, optimization, or a variety of other reasons. When developing software on a safety-critical partitioned OS platform, however, this task could be quite disruptive if not properly planned for. With a properly architected system and some minimal supporting framework code, it is easy to envision a partitioned system which can be easily modified to accommodate late-breaking changes.

The perceived need for modification when developing on a partitioned OS platform is primarily related to the partitioning of the application itself. While there are guidelines for how to partition the software system, the choice is still highly project specific. Therefore, it is conceivable that a partitioning scheme that seems perfect in early development becomes overly constraining as the implementation progresses. In order to account for concerns with timing or data throughput, the system integrator may choose to combine two or more partition applications into a single partition, or split a single partition application into two or more new partitions. In order to accommodate such a change, the following modifications would have to be made:

- Update the system configuration to reflect the new layout
- Update the run-time schedule to reflect the new layout
- Convert calls to inter-partition communication routines to intra-partition communication routines, or vice versa

The first two tasks can be performed by a graphical system design and layout tool which generates the required system configuration files. The last task is a bit more involved, and has the potential to greatly alter the existing code. However, a lightweight communication framework which abstracts the details about where a particular task is located can effectively address this concern, and minimize the amount of actual code that would need to change as a result of any repartitioning of an existing system. A detailed discussion of such a framework is outside the scope of this paper, but would be a fruitful direction for future work.

II.C. Testability

If software development is consistently one of the hardest to manage portions of project development, testing is not far behind. “Big bang” integrations and difficult to debug emergent behaviors all make the job of testing an integrated software-intensive system a laborious, expensive job. To some extent the difficulties in testing are a function of how we build our systems. By building all of the functionality required of a flight software system into a single executable object, we are therefore committing ourselves to testing each function of that system to the same standard. Additionally, in order to comply with at least the spirit of the “Test as You Fly, Fly as You Test” mantra, it is imperative not to pollute the flight object code with any test-only logic.

Partitioned OS platforms allow us to effectively address the problems outlined above. First – and perhaps most importantly – by partitioning the entire flight software application into smaller functional pieces, we are no longer forcing ourselves to test each function of the software to the same level of rigor. It is obvious that not every function performed by the flight software is of the same criticality – real-time control code is clearly of critical importance, but downlink of science data is most definitely not. Partitioning allows us to test the critical functions more thoroughly than the non-critical functions without altering the software object itself.

Second, partitioning allows us to reduce the pain of integration testing by enabling testing of partitions in isolation. Since partition application only care that the input data is correct and that something exists to service the output data, they can effectively be tested in isolation. A new component that is ready to be deployed can be built into a test image, with all or some of the rest of the system replaced with stub or test components specifically designed to feed appropriate data to the component under test. In this way, debugging and testing efforts can be focused on the component under test, and not on secondary issues arising from interface incompatibilities and the like. Additionally, the explicit specification of inter-partition interfaces allows us to find data type mismatches at compile time instead of run-time.

Lastly, all of the above testing and verification activities are executed on the delivered objects themselves. The system can be built incrementally with a mix of delivered partition application and test-only partitions, without requiring a complete re-test once all final versions of each partition application are in place. By enabling a finer granularity of testing and an easier path to complete integration, partitioned OS platforms promote the development of testable software and support streamlined verification throughout the development lifecycle.

II.D. Reliability

Producing reliable software is hard, given that delivering a bug-free system of even moderate size is nearly impossible. Partitioned OS platforms aim to increase the reliability of critical software systems by attacking two of the more common problems – poor memory management and CPU utilization issues. By segmenting the application into brick-wall memory partitions, partitioned OS platforms allow for finer grained fault containment for any residual memory management errors in the delivered system. By enforcing a statically-defined cyclic schedule, each partition is guaranteed to receive its allocated amount of CPU cycles.

In order to accomplish the increasingly ambitious missions that are being proposed, the industry as a whole must find a way to qualify and gain confidence in new algorithms and techniques; the question, however, is how to deliver these potentially less-mature technologies while still ensuring the overall reliability of the final system. By isolating application code to its own memory partitions, partitioned OS platforms allow us to develop robust applications which strike a balance between well-tested, mature code and newer algorithms which promise significant savings – perhaps in terms of lowered operations costs – but which may be less mature in pedigree. By ensuring that each piece of the application receives some allotment of CPU cycles, we are eliminating the possibility that a rogue task will lock out a critical function, which could have disastrous results. Lastly, we note that the memory partitioning afforded by a partitioned OS platform, when combined with a well-thought partitioning strategy, can reduce the overall susceptibility of the flight software to radiation-induced memory corruption events. This feature, when taken in concert with a well designed avionics suite, can enable missions which must endure extreme radiation environments, such the proposed Europa/Jupiter System Mission.

III. Mapping Quality Attributes to Flight Software Lifecycle

The previous section presented a discussion of the relevant quality attributes that are well addressed by the adoption of a partitioned OS. In this section, we discuss the salient features of a system which embodies these attributes, and look at how the benefits of these quality attributes are realized throughout the software development lifecycle to develop and field a robust software system.

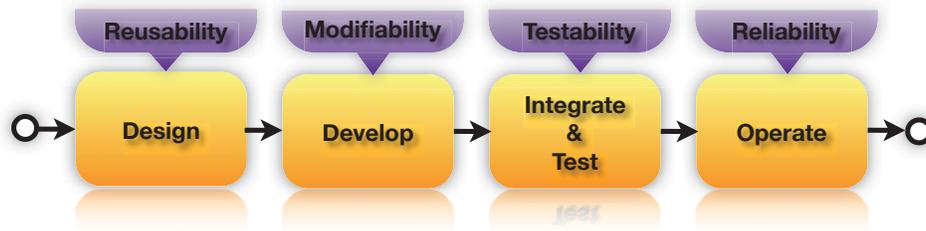


Figure 3. Architectural qualities of flight software lifecycle

III.A. Reusability during Design

In an ARINC 653-compliant partitioned OS, all communication takes place over ports and channels that must be statically defined. This requirement forces the application software components to have clearly defined interfaces with other portions of the system. Not only does this make the job of defining how the component should interact with other components much more transparent, it also provides an opportunity to make a direct mapping from system requirements to the implementation. Another feature of this requirement for strong interface definition is that it promotes both a clear method to approach code reuse in the early phases of the project, as well as an efficient way to design reusable, plug-n-play-type components.

To capitalize on this obvious benefit, we propose a software architecture that supports the type of reuse amenable to systems built on a partitioned OS. The type of reuse we envision is different in character from many recent attempts at developing a reusable code base. The reusable software systems that have been proposed in the past were either too closely coupled to the underlying avionics – potentially rendering the entire system obsolete by the time it reaches maturity – or required the acceptance of too many architectural commitments. In this case, the architecture would provide the required infrastructure – the ‘plumbing’ – but defer many of the application-specific details to the adaptation.

Common Problems: Design decision to create new or reuse software components.

- Create reusable software components: Minimal or no consideration is made in creating a reusable software components.
- Reuse existing software components: The difficulty of reusing existing software components is largely underestimated.

Concerns: Difficulty of creating reusable software and uncertainties associated with reusing software.

- Interoperability uncertainty: Software components are typically designed with their own specific interface and behavior specifications that are generally poorly documented. Thus, the interoperability of the existing software components with the new components is uncertain.
- Cost & Schedule uncertainty: Due to the uncertainty of the interface and behavior specifications of the existing software components, the cost and the schedule associated with interfacing with the existing software component is difficult to predict.

For Partitioned OS:

- Create reusable software components: New software component is built in a partition, with specific ports and timing requirements.

- Reuse existing software components: This software component can be reused as was designed as long as the port and the timing requirements are satisfied.

Benefits provided by Partitioned OS:

- Mitigate interoperability uncertainty: Partition interoperability is well defined by the port and timing specifications. Inter-partition communication occurs over statically-defined channels. Strong interface specification minimizes the possibility of runtime errors introduced by improper communication among various tasks within the system. Statically defined communication channels and data types allow for data flow and schedulability analysis before application is ever executed.
- Mitigate cost & schedule uncertainty: Well defined interface and timing specifications lead to more predictable reuse costing and scheduling.
- Required time allocation: During the design time, the time allocations can be scheduled statically.
- Required I/O ports: Design the port and channel configuration and memory allocations statically.

III.B. Modifiability during Development

A partitioned OS – when coupled with an appropriate architecture – can help facilitate the eventual and inevitable need to make certain types of changes to the system configuration based on test results. Specifically, we envision a software architecture that makes possible the movement of applications from one partition to another, or collapsing multiple partitions into a single partition, with minimal changes to the code base. This capability will afford software architects and system integrators the flexibility to make trades among the driving architectural qualities without posing a significant threat to the correctness of the affected applications.

Common Problems: Changes in the design requirements, schedule, budget, etc.

- Addition or Removal of features
- Redistribution of features

Concerns: Uncertainties due to design and development changes.

- Integrity uncertainty: Adding or removing features can affect other parts of the software and determining the extent of the effects can be challenging.
- Cost & Schedule uncertainty: With the uncertainty in the effect of the changes, the development cost and schedule also becomes uncertain.

For Partitioned OS:

- Addition or Removal of features: Add and Remove features by adding or removing partitions or processes.
- Redistribution of features: Redistribute features by splitting or combining partitions.

Benefits provided by Partitioned OS:

- Mitigate integrity uncertainty: Making changes to the relevant partitions in isolation does not affect other partitions.
- Mitigate cost and schedule uncertainty: Isolating the changes and knowing how the interfaces must change accordingly leads to predictable cost and schedule.
- Number of new partitions to be added: Change memory and time allocation. For removed partitions, simply remove the allocated memory and time.
- Number of ports and channels to change: Change the port and channel configuration and memory allocations.

III.C. Testability during I&T

Another major benefit of a partitioned OS is the flexibility that it provides during the integration and test phase. Partitions allow for code to be developed in isolation once the interfaces are defined; additionally, code modules can be integrated one at a time as well. This allows for early integration scenarios where the actual production version of one module can be run, even if other partition applications are not yet complete. Simulation or test-only components may be installed in unused partitions to ease testing of a single partition application, without negating the validity of the testing. Another essential benefit of partitioning with respect to integration is the ability to certify partition applications individually, which allows for safety-critical application code to be certified to the highest level, without forcing the same stringent certification requirements on non-critical application code.

Common Problems: Rolling up many features into one large delivery.

- “Big bang” integration: Many problems are caught during this integration step, and such problems may lead to late feature changes.
- Testing fully integrated deliveries: Debugging a fully integrated software can be exponentially more difficult. Though very difficult, such integrated test is required since one part of the code can inadvertently affect other parts of the code.

Concerns: Uncertainties due to the integration process and integrated software complexity.

- Testing completeness uncertainty: For fully integrated software, identifying all important test cases is impossible.
- Cost and Schedule uncertainty: Much of the cost and schedule overrun occur during integration and testing phase.

For Partitioned OS:

- Big bang integration: Integrate partitions as they become available.
- Testing fully integrated deliveries: Test partially integrated partitions based on their interface specification and other stub partitions.

Benefits provided by Partitioned OS:

- Mitigate testing completeness uncertainty: Identifying the important test cases becomes much easier. Test cases can be defined for each independent partitions. Test cases can be defined based on the coupling of the partitions, i.e. ports and channels are the only potential coupling among partitions.
- Mitigate cost and schedule uncertainty: The complexity of partition coupling can help predict the cost and schedule associated with integration and testing.
- Number of independent groups of partitions: Test each independent groups of partitions in isolation.
- Number of connections among interdependent partitions: The number of required test cases increase at least exponentially with the number of connections.

III.D. Reliability during Operation

Lastly, a partitioned OS provides many potential benefits to increase the reliability and operability of our fielded systems. Strong and intelligent partitioning increases the ability to isolate faults to within the involved application, therefore prevent a bug in a non-critical application from affecting the critical tasks. Such a system is therefore more robust to SEUs occurring in non-critical tasks.

In addition to the increased reliability, a partitioned OS can help to ease the task of integrating advanced algorithms that may not be as mature as other portions of the system. Lack of maturity is often cited as a reason for not integrating advanced algorithms into the software system, despite strong evidence that such algorithms can have a dramatic effect on science return and operations cost. The Autonomous Sciencecraft Experiment and EO-1 missions are prime examples of what can be achieved when advanced software is

integrated on-board. A partitioned OS provides an environment in which such algorithms can be integrated while offering protections against latent bugs that would jeopardize mission success by monopolizing CPU resource or corrupting memory of a critical task.

Common Problems: Faults occurring during operations.

- Fault isolation: A fault in one software component may corrupt the memory region of other software components and steal processing times from others.
- Fault recovery: Fault must be handled in a timely manner to assure mission success.

Concerns: Lack of robust fault isolation and recover may risk the safety and reduce productivity.

- Fault propagation: Fault must be isolated and prevent it from causing a chain reaction of faults.
- Lack of robustness: Fault recover software may also be affected by other faulty software components.
- Reduced productivity and functionality: Less mature, but more advance algorithms are not allowed on board due to potential risk on critical components.

For Partitioned OS:

- Fault isolation: Only way for faults to propagate from one partition to another is through the ports and channels. The memory and processing time of a partition is not affected by another faulty partition.
- Fault recovery: Built-in health monitoring enables more robust software health management techniques.

Benefits provided by Partitioned OS:

- Mitigate fault propagation: Only way for faults to propagate from one partition to another is through the ports and channels. This decreases susceptibility to problems arising in non-critical tasks (e.g. single even upsets). The core functionality is protected against latent errors in less mature application code.
- Improve robustness: Eases implementation of the simplified backup strategy for achieving redundancy within the application.
- Improve productivity and functionality: Integration of advanced on-board algorithms with reduced concern for the safety of the system.

IV. Demonstrations

During the first year of this task, we developed a standalone demonstration application to showcase the benefits of the technology (see top of Fig. 4). This application runs on a commercial single board processor, similar to that used on many flight projects, and is intended to form the basis of a more flight-like demonstration application running on a flight-like hardware testbed, to be developed in year two. The demonstration simulates a 2-D inverted pendulum. We have implemented an inverted pendulum controller on top of ARINC 653 compliant OS in a single partition. This controller represents a safety and time critical application. Additionally, we have implemented a non-critical application in a separate partition for which we can arbitrarily increase memory usage and processing time. In doing so, we are able to demonstrate that safety and time critical applications can perform its required tasks without being affected by other non-critical, potentially buggy, applications. Throughout this demonstration design, implementation and operation processes, we have been able to better assess how partitioned OS can be leveraged to improve the quality attributes of the software lifecycle as mentioned earlier.

In the second year, we are architecting a subset of a flight software (FSW) on top of ARINC 653 compliant OS that is running on and along with flight-like hardware. This demonstration will help us realistically assess the benefits of FSW architectures based on ARINC 653. Through this demonstration, we expect to gain confidence in our ability to design, develop, integrate, test and operate FSW that are based on ARINC 653. We also hope to better assess performance and resource overhead related concerns.

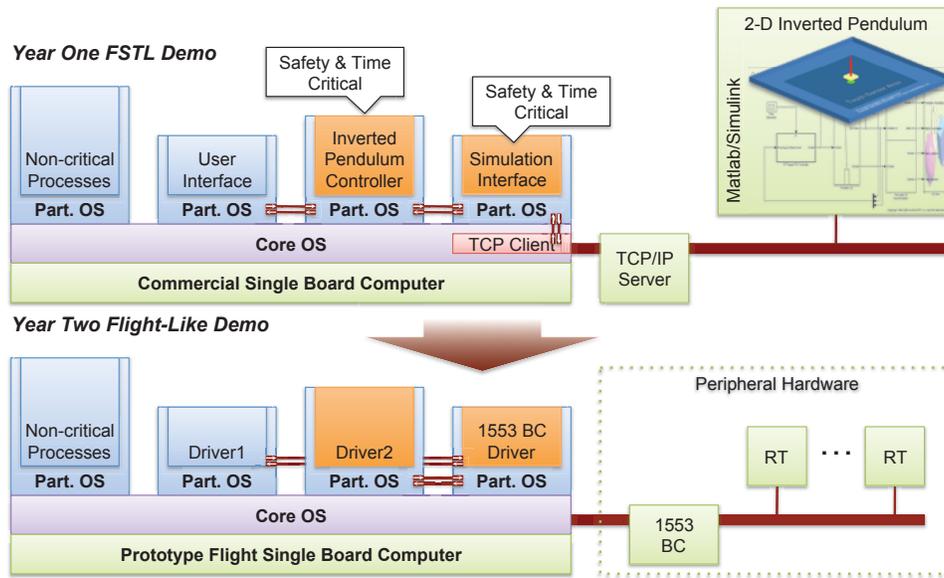


Figure 4. ARINC 653 Year 1 and Year 2 Demonstrations

V. Conclusion

In year one of the task, we undertook a detailed analysis of the lower-level software components of an existing system to both understand the driving architectural concerns, as well as to determine how such a system might be structured on a partitioned OS. This analysis led to the realization that many of the decisions about exactly how a system should be partitioned are too sensitive to the needs of a specific mission to make any sort of general prescription. Rather, it appears that providing a suitable architecture, along with a set of guidelines for how to approach the required analyses, is the more appropriate path. In the preceding discussions, we presented a case for how the adoption of a partitioned OS can allow projects to leverage benefits throughout the *entire* lifecycle, a perspective that is not readily apparent – or typically considered – when considering adoption of a new OS, which is traditionally thought of as only affecting low-level software and hardware interactions. To provide a more concrete idea of the benefits that can be achieved through the adoption of a partitioned OS, we provided a demonstration showcasing the benefits that have the greatest potential to improve the overall reliability of our flight software systems.

The remaining work in the task is focused on the implementation of a detailed demonstration on a hardware testbed with flight-like avionics to both prove out the proposed architecture, as well as to generate realistic performance data to use as a basis for comparison against existing software systems. Through this effort, we hope to provide partitioned OS-based architectural guidelines that can provide a strong infrastructure for developing systems that provides benefits throughout the entire development lifecycle, from requirements and design, all the way to implementation, testing, and operations.

Looking ahead, we see several categories of possible future work to be considered beyond this task. First, we believe that the development and refinement of existing development processes is necessary for successful adoption of a partitioned OS. Development of a software system based on a partitioned OS platform requires definition of new roles, as well as re-scoping of existing roles. Additionally, in order to have guidelines for how to treat software of different criticality levels, we must first assign criticality ratings to each function within a typical flight software system. Second, we note the need for a robust suite of CASE tools to support partitioned OS development. Many tools are already in existence, but some will need to be developed in-house; this effort would be tasked with baselining a development toolchain. Third, partitioned OS platforms require a unique set of modeling and analysis tools not previously required; for instance, it will be necessary to have a way to generate and verify the feasibility of a proposed partition execution schedule. Investigation of appropriate modeling abstractions will also prove beneficial to overall development process. Lastly, we propose the development of a lightweight software framework to abstract some of the communication within a partitioned system to allow for greater modifiability throughout the development lifecycle.

Acknowledgments

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration and funded through the Research and Technology Development program. The authors would like to thank Dan Dvorak and David Hecox for their technical insight and guidance over the course of the task, and Dave Wagner for his insight on the discussion of architectural quality attributes for flight software systems.

References

- ¹Airlines Electronic Engineering Committee, "Avionics Application Software Standard Interface Part 1 - Required Services," ARINC Document ARINC Specification 653P1-2, Aeronautical Radio, Inc., Annapolis, Maryland, March 7 2006.
- ²Ramsey, J. W., "Integrated Modular Avionics: Less is More Approaches to IMA will save weight, improve reliability of A380 and B787 avionics," *Avionics*, February 1 2007.
- ³Watkins, C. B. and Walter, R., "Transitioning from Federated Avionics Architectures to Integrated Modular Avionics," *Proceedings of the 26th Digital Avionics Systems Conference (DASC '07)*, Dallas, Texas, October 21–25 2007, pp. 2.A.1–1–2.A.1–10.
- ⁴Garside, R. and F. Joe Pighetti, J., "Integrating Modular Avionics: A New Role Emerges," *Proceedings of the 26th Digital Avionics Systems Conference (DASC '07)*, Dallas, Texas, October 21–25 2007, pp. 2.A.2–1–2.A.2–5.
- ⁵Black, R. and Fletcher, M., "Next generation space avionics: a highly reliable layered system implementation," *Proceedings of the 23rd Digital Avionics Systems Conference (DASC '04)*, Salt Lake City, Utah, October 24–28 2004, pp. 13.E.4–1–13.E.4.–15.
- ⁶Windsor, J. and Hjortnaes, K., "Time and Space Partitioning in Spacecraft Avionics," *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT '09)*, Pasadena, California, July 19–23 2009.