

Spacecraft Avionics Software development then and now: different but the same

Mark L. Mangieri¹
John (Jack) Garman
Jason Vice²

NASA – Johnson Space Center, Houston, Texas, 77058

[e-mail: mark.l.mangieri@nasa.gov](mailto:mark.l.mangieri@nasa.gov)

[email: jack@jackgarman.com](mailto:jack@jackgarman.com)

[e-mail: jvice@odysseysr.com](mailto:jvice@odysseysr.com)

NASA has always been in the business of balancing new technologies and techniques to achieve human space travel objectives.

NASA's historic Software Production Facility (SPF) was developed to serve complex avionics software solutions during an era dominated by mainframes, tape drives, and lower level programming languages. These systems have proven themselves resilient enough to serve the Shuttle Orbiter Avionics life cycle for decades. The SPF and its predecessor the Software Development Lab (SDL) at NASA's Johnson Space Center (JSC) hosted flight software (FSW) engineering, development, simulation, and test. It was active from the beginning of Shuttle Orbiter development in 1972 through the end of the shuttle program in the summer of 2011 – almost 40 years.

NASA's Kedalion engineering analysis lab is on the forefront of validating and using many contemporary avionics HW/SW development and integration techniques, which represent new paradigms to NASA's heritage culture in avionics software engineering. Kedalion has validated many of the Orion project's HW/SW engineering techniques borrowed from the adjacent commercial aircraft avionics environment, inserting new techniques and skills into the Multi-Purpose Crew Vehicle (MPCV) Orion program.

Using contemporary agile techniques, COTS products, early rapid prototyping, in-house expertise and tools, and customer collaboration, NASA has adopted a cost effective paradigm that is currently serving Orion effectively.

This paper will explore and contrast differences in technology employed over the years of NASA's space program, due largely to technological advances in hardware and software systems, while acknowledging that the basic software engineering and integration paradigms share many similarities.

Software Production Facility (SPF) – Retrospective

The Shuttle Orbiter Avionics Software Production Facility (SPF) existed functionally from 1972 through 2011 when it was decommissioned following the end of the shuttle program in June of that year. The almost 40-years of birth and evolution in avionics software engineering and test in that facility was fundamental to the successes of the shuttle program and mirrored the evolution of technology of those days through current times.

¹ *Kedalion Project Manager, Spacecraft Software Engineering Branch, Houston, TX 77058/MS ER6, AIAA Member.*

² *Kedalion Chief Engineer, Odyssey Space Research, LLC, Houston, TX (www.odysseyspace.com), AIAA Member.*

When the orbiter contract was awarded, in total, to Rockwell International in the summer of 1972, the then Director of the Johnson Space Center, Christopher C Kraft, directed that the flight software subcontract, which Rockwell had awarded to IBM's Federal Systems Division, be novated (moved in place) as a prime contract to the Center. A new organization, Spacecraft Software Division, was formed at JSC, consisting of members with primary experience on the Apollo flight computer software, the Mission Operations Directorate (MOD), and the Mission Planning and Analysis Division (MPAD) – for experience in avionics, software engineering, flight operations and systems management, and guidance and navigation.

The Real Time Computer Complex (RTCC) within the Mission Control Center (MCC) had recently gone through a major mainframe computer upgrade and a series of IBM 360's were made available to host shuttle flight software development and test. IBM set up operations in Nassau Bay across the street from JSC and were given access to these "left over" IBM mainframe computers. This system was initially called the "Software Development Laboratory" or "SDL" and evolved into the SPF over the following years with mainframe upgrades and a move in the administrative wing of the control center (Building 30) where it remained, albeit expanded, until 2010.

Under a research project initiated following Apollo, NASA had awarded a contract to Intermetrics Inc of Cambridge Mass for the development of a higher order language which eventually came to be called "HAL/S". The acronym "HAL" was never formally defined, and the "S" referred to a "shuttle" subset of the original language specification which was actually implemented in compiler-form for the development of the shuttle flight software – both primary and backup which is explained further below.

It is important to understand that the avionics experience coming into shuttle software engineering in 1972 was based on "assembly language" – higher order language compilers were considered inefficient in code generation, and given that flight computers used power-hungry (and weighty) core memory, the actual size or "lines of code" of software was considered a commodity to be managed carefully and kept as small as possible.

The Environment of the SPF and its Products

The Orbiter was equipped with five flight computers – called "GPCs" or "General Purpose Computers". They were based on the IBM AP-101 avionics computer which used essentially the same instruction set as the mainframe IBM 360, thus giving IBM a head-start on the various utilities and assemblers that could be used. An Input/output Processor (IOP – in essence another computer) was affixed to the AP-101s to form the GPCs and connect them to the bus system of the evolving orbiter design (See Figure 1). The AP-101 instruction set was implemented in microcode which could be and was modified to adapt it further to the Orbiter environment's needs. One other major component, the Display Electronics Unit (DEU) was a computer system attached to the GPCs for driving the CRT display system in the Orbiter's cockpit. Early in the design of these systems the decision was made to essentially "freeze" the software in the DEUs and the IOPs making them effectively peripherals to the AP-101s and concentrating all software development evolution and change to the AP-101s within the GPCs. This GPC flight software formed the basis of the products coming out of the SDL and SPF for all facilities requiring flight software in the orbiter program.

Initially the GPCs were assigned roles – three for guidance and navigation (redundant GNC), one for system management (SM) and one for payload management (PL). In fact there was actually a switch system in the cockpit for assigning those roles to the five computers.

The “fail operational/fail safe” requirement on the Orbiter actually required quad-redundancy (first failure leaving three and thus full fail operational capability). So the notion of a payload computer was dropped. Then, by 1976, it was concluded that backup flight computer software (BFS) would be a reliability requirement and the notion of a separate SM computer was dropped. This left four GPCs to flight with primary avionics software from the IBM FSD contract managed by SSD, and the fifth computer flying backup software built by Rockwell.

The BFS was not a “backup flight system” in any common meaning of that phrase. It used the same computer (and could be loaded in any of them), the same buses, and in fact the same requirements and HAL/S compiler – albeit ported to a different set of computers at Rockwell’s facility in Downey California. The primary difference was the operating system. The FCOS (“flight computer operating system”) in the four primary GPCs was based on an asynchronous priority driven structure which allowed for unanticipated errors and recovery paths – similar to (and in memory of) the alarm overload conditions that occurred in the Apollo Guidance Computer (AGC) in the Lunar Module (LM) during the last phase of descent on the first landing on the Moon by Apollo 11.

The BFS operating system was based on the traditional avionics fixed or synchronous structure, which allotted specific times slots for each major task or process. While difficult to get a BFS design (and minimum constraints on the primary) that could allow the BFS to “flight follow” the primary (until needed), it was successfully done. In fact setting up that flight following synchronization between the primary and backup was the root of a “bug” which held up the Shuttle’s first orbital flight for 48-hours in 1981.

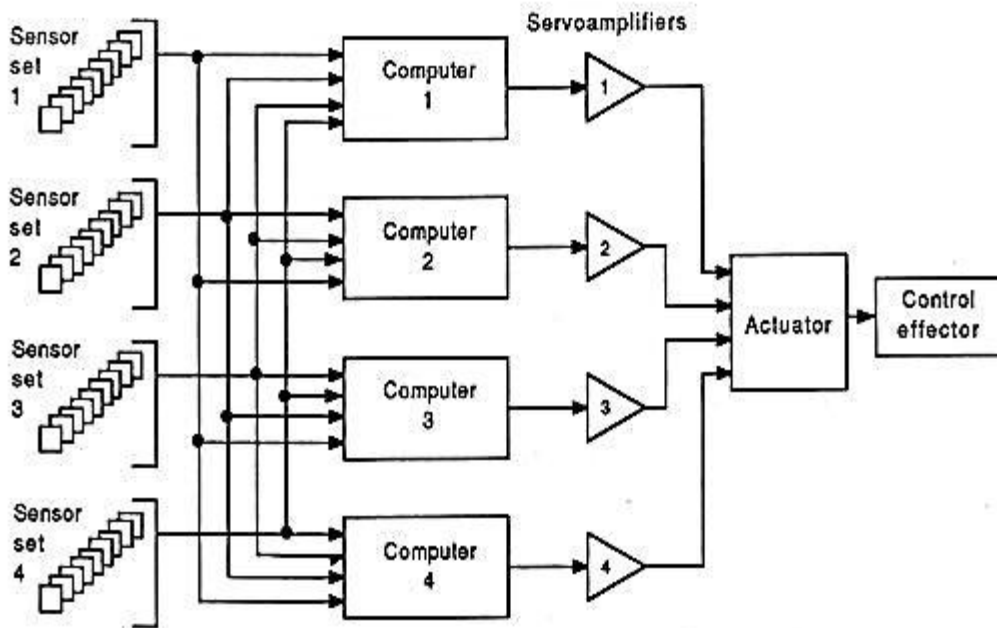


Figure 1 - Multiple “Voting” General Purpose Computer (GPC) Architecture

The notion of having a different operating system meant that the fundamental concern in primary avionics software reliability had some coverage. Specifically, if there were some way for a “bug” to survive in the four identical primary computers (i.e., the four GPCs loaded with primary software), they would all consider themselves operational in any voting scheme, but depending on the bug, could catastrophically fail in their mission. In the end, the diminishing number of software errors encountered in the changing primary software almost all related to asynchronous (unpredictable) timing and ordering or processes. The probability that a coincidence in timing in the primary with its operating system, would happen in the BFS with its alternate operating system, was diminishing small – hence the decision to have the BFS. There are a few other points worth mentioning before moving back in to the primary software and the SPF:

- The primary flight software could never be made to “fit” in the GPCs. Different “loads” were built for ascent, orbit, and entry phases. And in fact, even this wasn’t enough as the ascent load was still almost twice the size of the GPC memory just a few years prior to the first orbital flight. Software “scrubs” were a watchword in those days, and required almost continuous rebuilding of the primary avionics software (PASS). Everyone concluded that this would not have been possible without the use of a higher order language (i.e., if software development had been purely assembly language). In fact, the BFS was later used in a quasi-primary function to absorb some of the crew display requirements away from the PASS.
- The different computers “loads” were kept on a mass memory unit (MMU) tape device. The ascent configuration supported an abort re-entry, and thus was the closest to a complete end-to-end flight scenario in content (and thus the driver in size and scrubs). On the other hand, there was such concern about loading the entry configuration into a GPC, that a spare GPC was actually carried on early flights pre-loaded with the entry software – “freeze dried” as it was called. It might be noted that while this concept was actually possible with core memory, it could not be done with today’s RAM technology without a continuous source of power – aka as “standby” mode in most modern computers.
- The notions of SM and PL software in separate computers may have disappeared, but the requirements for SM and PL were embedded along with the GNC requirements in the various “loads”.
- Switching to the BFS during critical flight (ascent or entry) was “one-way” – there was no switching back to the primary (it was never done, and the potential success of such a switch at all possible moments mitigated, to some extent, the overall reliability the BFS added to the orbiter GPCs.)
- The PASS software development effort was able to relax a number of significant requirements with the existence of the BFS. While considered extraordinarily lower in probability, the loss of power to all GPCs would require a switch over to the BFS if/when power was restored. The primary computers were programmed to “resign” from the redundant set in the face of power loss. Only the BFS was programmed to attempt recovery. Of course the most probable loss of power or any form of computer shutdown would be on just one computer, primary or backup, and resignation of a primary computer without disturbing the remaining computers was exactly the right approach to “recovery”.
- Both the PASS and BFS had to undergo “flight-to-flight” re-configuration for each and every shuttle flight. This was due to several factors:

- Each orbiter was different – not just parameters like weight and c.g. which drove the parameters associated with guidance and flight control, but also small differences (or even large differences following upgrades) in subsystems the software managed or was dependent upon.
- Each flight was different – for example, the parameters associated with trajectory and targeting. During a period of involvement with DoD payloads in the orbiter these were often classified information placing a huge burden on the SPF design and operation.
- Support to payloads was different – clearly different on each flight, the GPCs were loaded with different versions of flight software during orbital operations to support both shuttle orbit operations and what payload support was required of the GPCs.

A Challenge in Software Engineering

There is one more aspect to both the history of the SPF and the PASS critical to understanding the evolution of flight software over the years. In the early 70's when the Orbiter contract was awarded, all "real time" software was done in "assembly" language, and this largely precluded emerging software engineering practices. Early higher order languages such as COBOL and FORTRAN were in use and evolving, but almost entirely in "batch" (non-real time) software. Even with programming guidelines in place, the dependence that flight computers had on core memory and power-hungry processors meant that software actually "weighed" something in the very tight contest for space and power in aerospace vehicles. Hence efficiency, or very "tight code" was the norm, and "programmers could do anything" at the assembly language level.

Efficient code was generally the antithesis of good software engineering and software maintainability or "sustaining engineering" as it was called – the major driver to the life cycle cost of software.

Figure 2 depicts the "distance" from object code at which programmers design and creates software – the notion of "layers of abstraction". Programming at the object code level hasn't occurred since the very early days of computing during and following World War 2. However, programming at the alphanumeric or assembly language level persisted well into the '80s for critical control and real-time software.

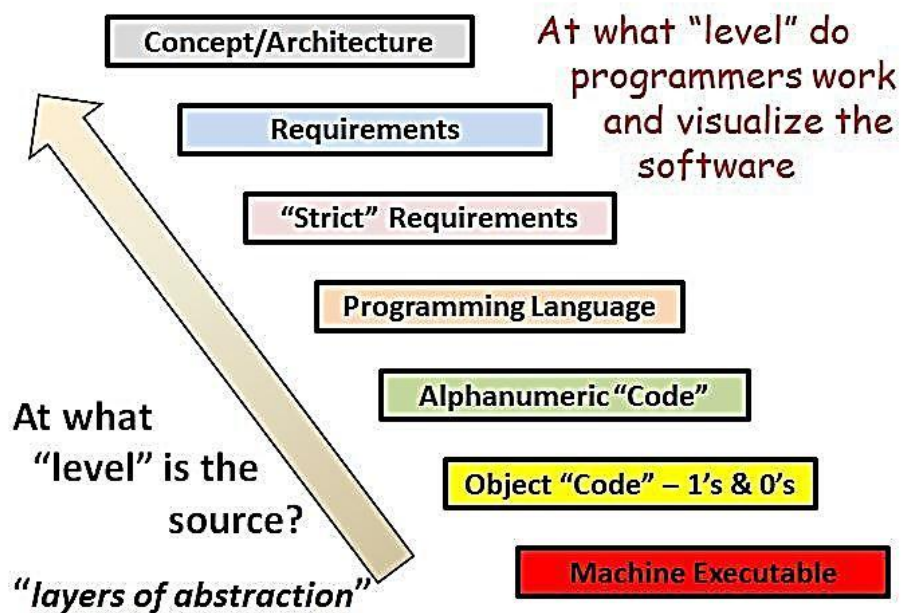


Figure 2 – Layers of Abstraction in Software Engineering

Figure 3 depicts the assembly language level of programming, and depicts the very large proportion of the overall software engineering and programming effort that is "manual" and thus by policy and written standards. Figure 4, by contrast, depicts the level that we urged for Orbiter flight software (PASS and BFS) with the HAL/S language and compiler effort as noted earlier.

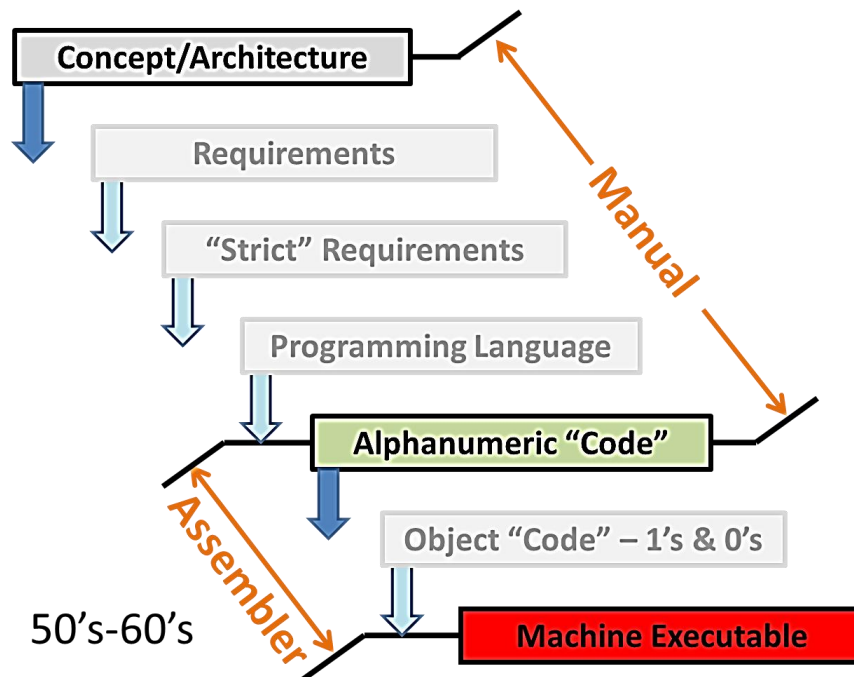


Figure 3 – Assembler Level Programming

However, the program insisted on some quantification of the “cost” (in increased size of flight software) for using a compiler vs. assembly language. The answer that was used in planning was 15%, and it was based, literally, on a “contest” that was performed early in the decision process.

Several of the very best HAL/S programmers (from Intermetrics) were teamed against some of the very best assembly language programmers available at the time. Each team was given a set of identical requirements designed to give a reasonably fair measure of flight software code sizing. The result of the contest was the 15% increase in lines of code for the HAL/S team over that by the assembler programming team.

But far more important than this “cost” was the gain in ease of change and maintainability afforded by the use of the compiler level of programming. A conservative estimate of PASS development is that it was programmed fully four times prior to STS-1 in order to make it “fit” in the available memory of the GPCs. In fact, it never did “fit”. The PASS had to be divided into ascent, orbit, and entry “pieces” kept on a “mass memory unit” (MMU) tape drive. That MMU was initially designed and required only to restore or initially load a single configuration of software in the GPCs.

All parties involved in the early development of the flight software later agreed that it would have been impossible to react and perform the amount of change done to the PASS during the final year or two preceding the first orbital flight (STS-1) had it been done in assembly language. By being “up” at the compiler “equation level” of programming, changes were much easier to design and implement - and much less error-prone.

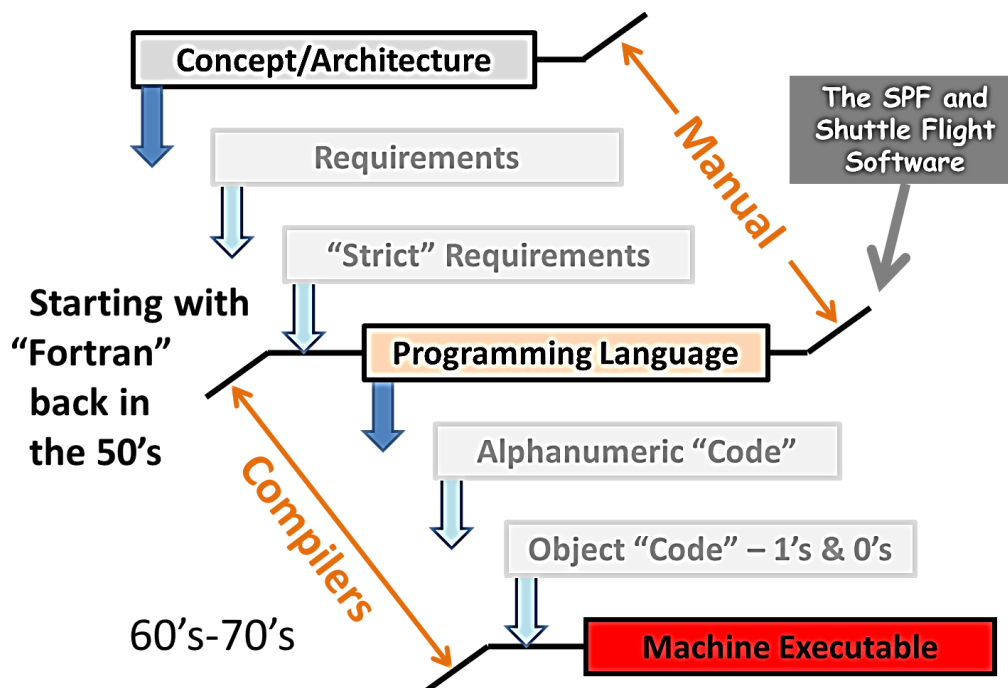


Figure 4 – The Compiler Level of Programming

Flight Software Testing

Using Actual Flight Software in Lab Environments

From early Apollo days the notion of flight software testing involved using actual flight computers in a laboratory environment, and “fooling” them into thinking they were actually flying the spacecraft. Host computers provided the simulation environment, and for shuttle, a special “box” called the Flight Equipment Interface Device (FEID) was designed and built. It contained actual GPCs but provided an interface to the host simulation environment (those IBM mainframes) and allowed them to stop/start the GPCs both to “keep up” with simulated real time, and to extract test information from the GPCs executing the flight software being tested.

Thus the SPF never ran in “real time” even though it utilized the actual flight software and hardware.

Crew trainers adopted the same approach, and a special version of the FEID was constructed for those facilities at JSC (motion base and fixed base). These simulators had to run in real time, and therefore lacked the ability to extract much test information while running.

On the other hand, the Shuttle Avionics Integration Laboratory (SAIL) at JSC was designed to test and integrate the various subsystems of the overall avionics world, and thus required minimum alteration of GPC data buses and traffic, and generally also had to run in real time. So yet another “FEID” device was designed and built for that facility.

Two more major facilities rounded out the “customers” for PASS (and later BFS) software – the orbiter manufacturing facility in Palmdale California, and the Kennedy Space Center (KSC) launch and launch preparation facilities in Florida, both of which involved “real Orbiters”. In summary, there were five “customer locations” for the PASS:

- The SPF itself – source of the software
- Palmdale – the earliest customer outside the SDL/SPF (for vehicle manufacture and test)
- SAIL – for avionics subsystems integration and test
- Crew Trainers
- KSC

One of the more interesting “accidents” in the planning of Orbiter Avionics software development was the absolute need of “actual” flight software at Palmdale. The very nature of Orbiter design dictated the need for GPCs and software to perform virtually any form of vehicle test – even during early fabrication and development. This need was not fully anticipated early in the program, and even “test” software, while absent the rigor of most safety critical requirements, is simply applications running in the environment of a flight computer – the GPCs. The creation of such applications is a layer of software on top of that required for software engineering and production (e.g. compilers), an operating system, a user interface, and more. These elements simply didn’t exist early in the program. In fact, when the AP-101s were acquired to be the core component of the GPCs, there was no operating system available.

A Simplified View of the Real Time Applications Software Environment

A reasonable classic (today) view of applications software (“apps” in today’s consumer environment) is shown in Figure 5 (“A View of Apps in a “real time” Computer”). The notion of “real time” computing didn’t really exist in the early days of computing. This didn’t happen until applications emerged that allowed users to interact in “real time” – adjusting the flow or paths of applications. Clearly any form of

“flight software” is “real time” – as are virtually all applications in today’s personal computing devices.

This is a far cry from the notion of submitting some form of input like a card deck to a mainframe computer and receiving the results of the application program following a “batch processing” of user’s inputs in a sequence vs. real time ordering.

To illustrate, Figure 5 shows the basic components surrounding applications in a real time computing system:

- The UI or user interface that receives user input and send back user information
- The OS or operating system that stands between all applications (including UI) and the computing hardware environment.
- The “Source Code” which flows through a set of software engineering tools depicted here simply with the label “Programming Language”

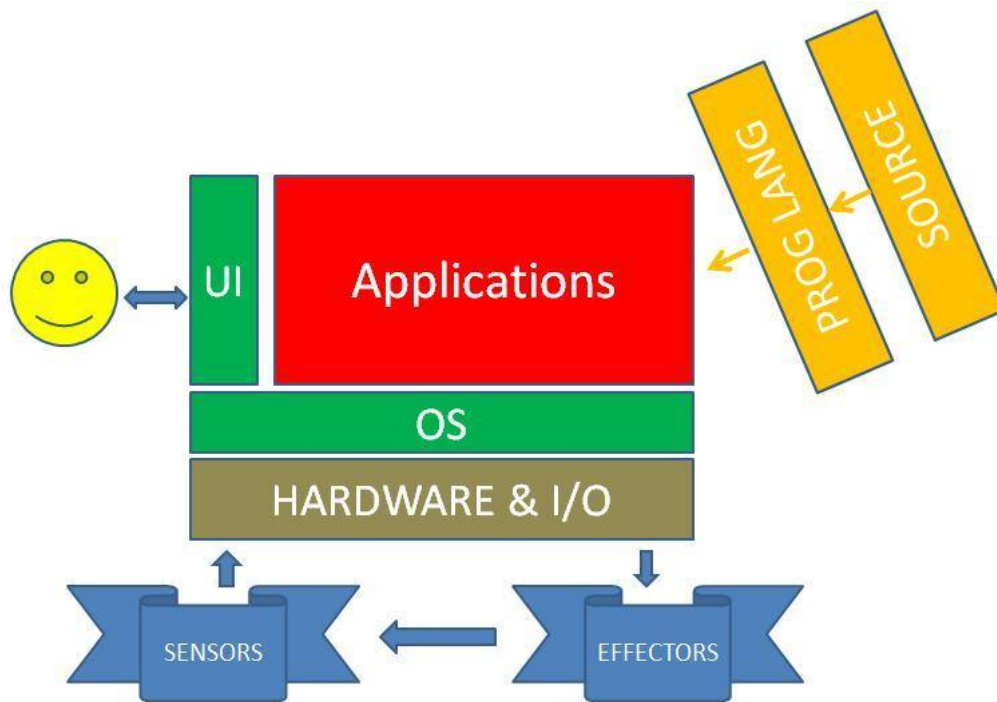


Figure 5 – A View of Apps in a “real time” Computer

Finally, the computer hardware and input/output structure that “hears” the real world via sensors and feeds back output through one form or another of “effectors”. In this simplified view, UI uses the same sensor/effector loop (e.g. a computer mouse moving across a computer screen). But in the context of any kind of control applications, the focus here is on sensor input to the application’s prime function and similarly the output to effectors required for the software to perform its function.

- Even a home programmable thermostat follows this overall structure:
 - The UI being input of desired temperature and time, with output being display of the user’s settings
 - Whereas the control loop is the sensor input of temperature, and the effector output is control of the heating or air condition unit.
- A flight control (or factory control, or any process control system) follows an identical “albeit more complicated) “structure”.

- The “user” interface to manage the overall “mission” (e.g. pilot control), with sensor inputs such as acceleration and position and effector output such as engine throttle, pointing, and other vehicle control components.

Figure 6 takes this simplistic view of applications software and puts it in the context of the Orbiter avionics. The applications are those related to GN&C (guidance, navigation, and control), systems management, and payloads. The software engineering tools are embodied in the SPF with HAL/S being the dominant programming language and compiler.

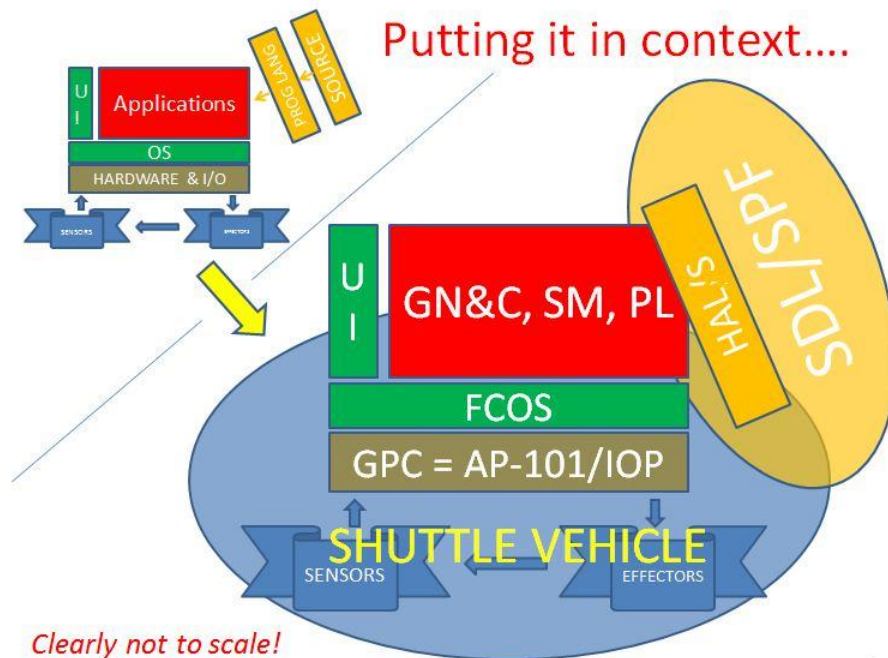


Figure 6 – Putting it in the Context of Orbiter Avionics

And the UI is all the functions associated with displays and crew input devices (hand controllers, switches, keyboards, etc. as inputs: CRT” displays, lights and other indicators as output. Similarly, the sensor/effector interface is the orbiter vehicle with its many sensors and effectors (from accelerometers to commands fire and throttle “rocket” engines).

While this “model” of applications environment leads to a clearer discussion of the SPF and its functions, it also demonstrates the difficulty in providing “test software” to support early Orbiter development and fabrication.

In fact, the UI, OS, HAL/S compiler, and SPF tools were all in development at the same time, each following a carefully constructed set of written interfaces to hopefully bring everything together. If you consider today’s world of software engineering and programming, this was almost literally “Mission Impossible”.

However, this unanticipated need for Test Software became both a boon and bane to Orbiter Avionics software development. On one hand, it provided a real world environment in which to test the platform of early versions of software engineering tools, UI, and OS, upon which the relatively simple test software applications rested – thus aiding their parallel development. On the other hand, all these components had to work! (... and very early in their development cycle)

A similar need for early flight software existed for other “laboratory” systems – not as early, but the applications had to be closer to the behavior of the real Orbiter avionics software applications.

The development of the crew trainers and the Shuttle Avionics Integration Laboratory (SAIL) are the two primary examples. And likewise, these facilities provided the same boon and bane to the ongoing development of the flight software for use in the first flights.

Requirements on the SPF

Flight software Testing

The SPF had to comply with a set of “simple to describe but hard to do” requirements in testing flight software. In summary, these “test” requirements were the ability to:

- “Archive” every test
 - *The bit-for-bit version of FSW, compiler, and simulation used in the simulation test run*
 - *This allowed exact repeatability for further test evaluation)*
- Reach in to any “layer of abstraction” to retrieve information from a simulation/test
 - *Requires stopping the GPCs as needed to reach into the RAM*
 - *All data buses and such were simulated (unlike SAIL, Crew Trainer, Orbiter)*
- Extract information based on “names” at the HAL/S level
 - *Translated to binary value in a specific memory location*
 - *Extracted by “freezing the GPCs*
 - *Post run creation of English value with proper units*
- Establish “branches” off “reset” points for variants in testing
 - *This permitted the creation of large varieties of test profiles off common roots without having to “start from scratch” for each.*

To appreciate the complexity here, it’s important to remember that the tests referred to here are “simulation” runs on actual GPCs inside FEIDs, running the actual object code of the flight software being tested.

Additional SPF Requirements

Moreover, these test requirements were in addition to the equally complex requirements necessary to support programmers in the actual development or production of the flight software. The HAL/S compiler is of note simply because it was developed by yet another contractor (Intermetrics, Inc of Cambridge, MA) and provided to IBM as “GFE” (government furnished equipment”). Because compilers not only create the actual object code resident in the flight computers, they also make use of “libraries” (collections of subroutines) called upon as necessary in response to programming inputs. These libraries were in fact additional “flight software” components created by Intermetrics and not IBM. This notion is innate to software development tools today, but was a reasonably radical concept “back then”.

Beyond the compiler were the tools needed to create and store libraries of HAL/S source code, test setup’s, and much more. As it turns out, virtually all the Orbiter avionics software development was done with IBM 80-column card decks submitted in “batch” runs to the SDL/SPF mainframes. As noted earlier, the IBM facility was outside the gates of the Johnson Space Center, and courier vehicles went

back and forth between the SPF and the IBM facility continuously all day long carrying card decks and printed paper output from both code creation and test simulation “runs”. It wasn’t until very close to the first orbital flight that 80-column CRT screen and keyboard input was migrated into the SPF environment (these “dumb terminals” went by IBM’s extensive and arcane numeric device naming: 3270, 3278, and more).

Finally, in order to run any test, all the components shown in Figure 6 were required, including, and not yet discussed, a complete simulation of both the orbital vehicle and its real world environment. This simulation had to be real and accurate enough to provide an equivalent “view of the world” to the GPCs inside the FEIDs as the actual flight computers would and did see.

Peering in to the SPF

Figures 7, 8, and 9 depict a build-up of SPF components. Along with the GPCs and FEIDs, Figure 7 depicts the flight software generation tools. Figure 8 adds the vehicle and environment simulation. Finally Figure 9 adds the flight software itself (as in Figure 6) depicting the entire SPF structure.

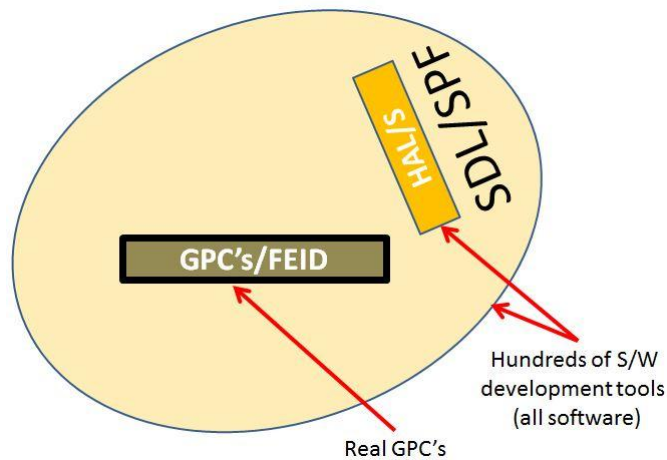


Figure 7

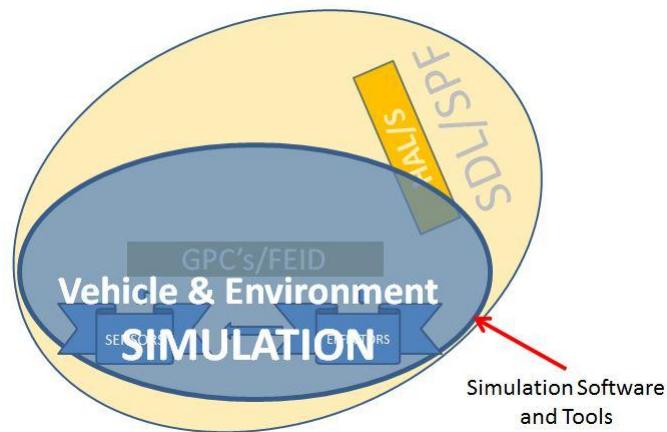


Figure 8

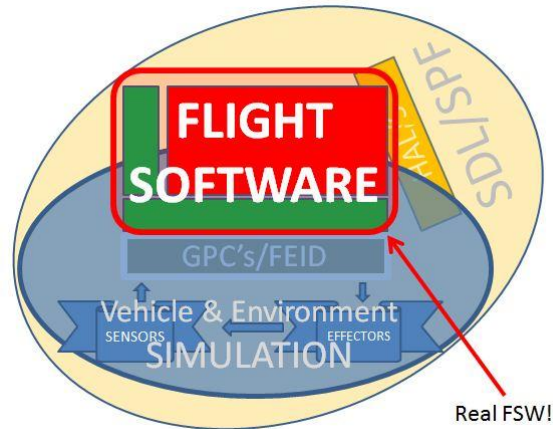


Figure 9

Aside from the FEIDs, the hardware in the SPF was no more than the collection of IBM mainframe components typical of any data center. As noted above, the SPF was initially known as the “Software Development Laboratory” (SDL) and existed in the Houston Mission Control Center running on “hand me down” IBM mainframes from that facility.

From SDL to SPF

The Argument for Evolution

The evolution to becoming the SPF was mainly in the acquisition of upgraded mainframe components and movement into the core area of the administrative wing of the building housing the MCC. Mainframe acquisitions were terribly difficult to perform in the 70’s due to Congressional rules and oversight. The argument for SDL-to-SPF evolution was mainly based on projected flight rate.

Initial flight planning for the Shuttle assumed 52 flights per year – yes, one flight per week! Looking back, this is hard to believe, but nonetheless it was the plan prior to STS-1. All the preparation for a given flight, and in particular that required on the flight software which became known as “flight-to-flight reconfiguration”, took as long as a year. Taking 52 flights and laying out a year’s effort for each created a view of parallel efforts similar to that depicted in Figure 10 below. As noted in the figure, a vertical line drawn through such a diagram was “shocking” in terms of the efforts that would be necessary.

Virtually all focus was on STS-1 until, as the April 1981 date of that flight approached. That made sense of course – basic development of the orbiter, all its subsystems, and all the processes, tools, and operational support required for just that flight were prerequisite to the second and any further flights. However, even the most conservative estimate of SPF computer resources required to support 52-flights per year wildly exceed the computer capacity of the SDL used for all initial flight software development.

On one hand, the Shuttle program and the SPF were fortunate that nothing approaching those flight rates was achieved. But on the other, it provided justification for the upgrade and evolution of hardware (and software) that became the Software Production Facility or SPF! While program management initially asked “Aren’t we done with flight software development?” there was a profound change and realization in thinking about program operations for flights after STS-1.

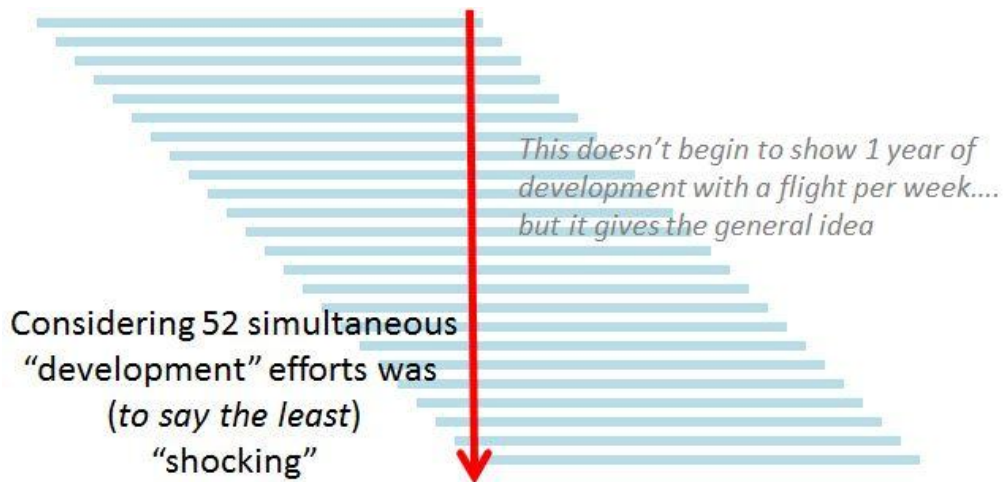


Figure 10 – Flight-to-flight Reconfiguration

To our credit, the flight software development community did recognize the aforementioned need for flight-to-flight reconfiguration required in the software to support a specific vehicle, payload, and mission profile. But we did not anticipate the magnitude of the required ongoing flight software development effected by evolving requirements (often termed "sustaining engineering"). Even reducing the flight frequency and flight-to-flight reconfiguration effort by a factor of 10 (to 5 flights per year), still required a facility much larger than the SDL. Moreover, this facility had to be fully operational prior to STS-1.

The challenges of any complex facility upgrade are enormous, and the SDL-to-SPF transition was no exception. The major challenges included

- ✓ Mainframe Acquisition and the "Brooks Act" (legal impediments and overhead to mainframe acquisitions)
- ✓ The building modifications required to house the SPF as a major "1970's data center"
- ✓ Physically moving all the software, FEIDs, and support systems from the RTCC to the new SPF
- ✓ Getting it "done" in time

That effort was successful, and is truly "another story".

Kedalion – NASA’s Orion insight and SW analysis lab

Since Orion’s inception, NASA has conducted a risk mitigation strategy by implementing an in-house lab resource for targeted analysis and experimentation of Orion concepts. Largely used for analysis and validation of flight test loads, and GN&C Multi Organization Development Environment (MODE) team support, Kedalion developed into a significant program resource recognized by Orion program management as an integral part of Orion’s overall success.

Integration Synch Points (ISPs) are developed as interim demonstrations of Orion developmental technology at strategic points mutually agreed to by NASA and Lockheed-Martin. These ISPs accomplish both milestones of progress and confidence building, especially in the area of inherently risky program technologies. ISP schedules and capabilities are coordinated with the Kedalion lab for purposes of technology sharing between NASA and LM, as well as validation of ISP content as it is stood up in the Kedalion lab. Orion FSW loads will be delivered to the Kedalion lab for validation runs, much like PA-1, to further mitigate risk as the FSW loads are prepared for formal testing on the programs scarce VTB test platforms.

Lab heritage (JAEL, GITF, etc.)

NASA’s Kedalion lab is the latest implementation in a rich history of analysis, prototyping and integration facilities. Understanding the complex interactions between spacecraft avionics has always required hands-on engineering analysis and testing. Experience gained on previous NASA programs has shown that the use of an agile analysis and integration facility can greatly benefit a program by allowing for risk mitigation very early in the program development as well as throughout the life cycle of the program.

NASA facilities such as the Shuttle Program’s JAEL (JSC Avionics Engineering Lab) and the ISS Program’s GITF (GN&C Integration Test Facility) have brought many years of value to those programs by providing more efficient and productive facilities for early integration testing, risk mitigation, avionics burn-in, problem resolution and other critical program tasks best done in an agile process oriented facility. While not always initially implemented as an in-line testing facility, these types of NASA facilities have typically become early in-line testing facilities that allow for pre-formal testing in advance of the more expensive formal V&V facilities.

Kedalion was born from this heritage of less formal integration and test facilities. In fact, the Kedalion facility is housed in the same room as GITF was for many years and even uses some of the same facility assets as were used by GITF.

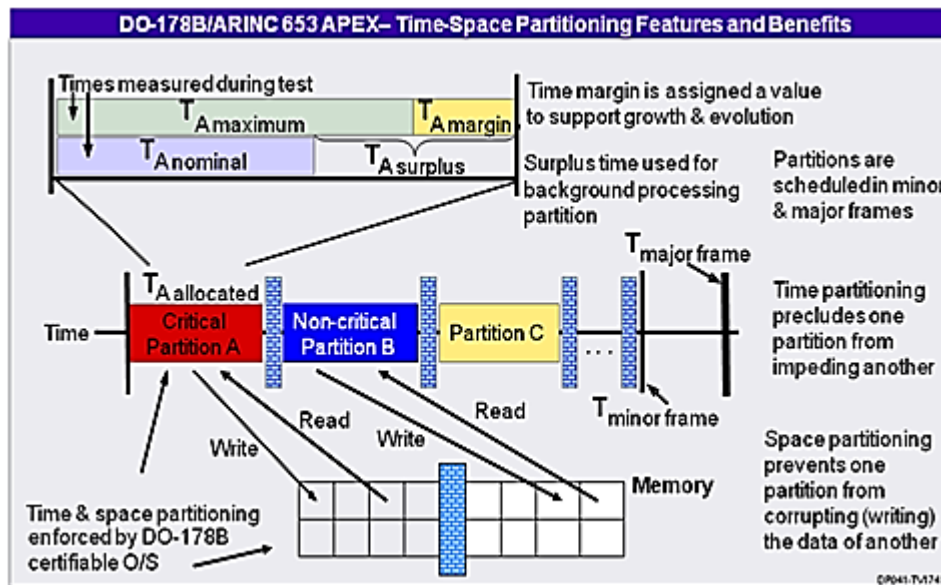
Kedalion’s purpose/mission

The Kedalion lab’s fundamental mission is to provide a facility for risk mitigation throughout the life cycle of a space vehicle’s development and operation. To achieve success the lab is oriented toward hands-on engineering analysis that allows NASA engineers to achieve a deeper understanding of the avionics software and hardware than could be achieved by only reviewing documentation. It is not, however, a formal oversight or verification facility but rather a flexible facility that welcomes collaboration with the vehicle contractors as well as personnel from other facilities to establish the best possible solutions to design issues, operational constraints and problem resolution as early in the life cycle of a vehicle as possible. It is this approach that allows NASA to blend years of experience on past programs with new

ideas from industry and from within NASA to produce solutions that can guide a vehicle's development toward ultimate success.

Orion Avionics Instantiation: Integrated Modular Avionics (IMA)

IMA represents a real-time computer network airborne system. This network consists of a number of computing modules capable of supporting numerous applications of differing criticality levels, isolated from one another in partitions supported by the underlying operating system. IMA provides for an alternative Hardware/Software (HW/SW) development and testing paradigm.



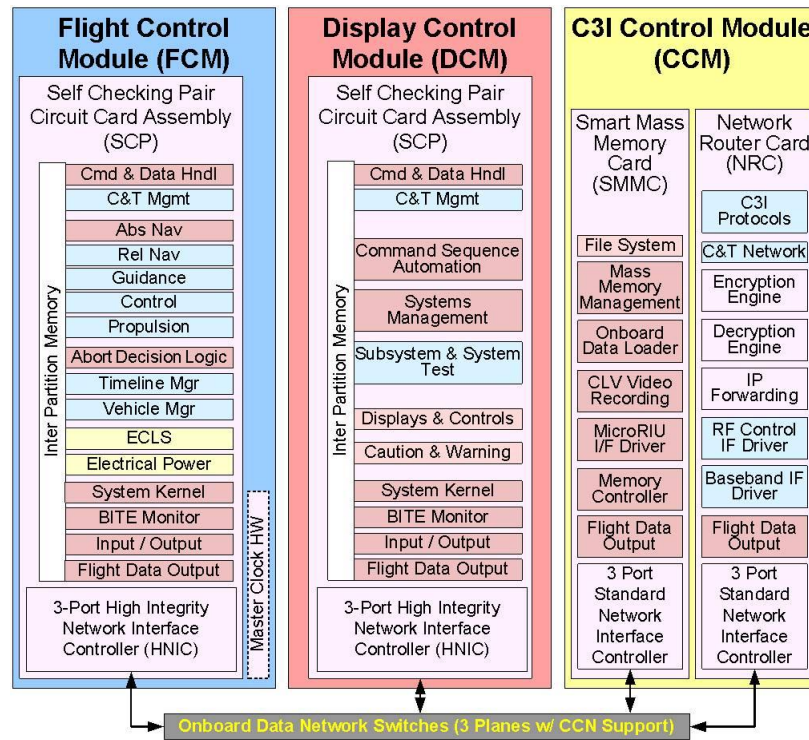
IMA Time & Space Partitioning

Orion's prime contractor, Lockheed Martin, has borrowed from Honeywell's vast experiences in its aircraft avionics heritage. Honeywell's paradigm is an ARINC compliant adaptation of IMA, which Honeywell had successfully deployed on several Boeing and McDonnell Douglas aircraft designs (B737/777/787, MD 10/90). These techniques are mature enough to have been certified as DO-178b compliant by the FAA, and also have the support of major commercial Real-time Operation System (RTOS) vendors with mature products.

While there are significant similarities between aircraft avionics system requirements and space vehicle avionics system design, an augmentation would be required to achieve the Orion avionics system design. Aircraft don't have flight dynamic phases as intense as a space vehicles ascent and decent phases, which require instantaneous reaction with multiple failover scenarios to avert catastrophic failure. Additionally, aircraft don't generally sustain missions as long as typical spacecraft, which could last weeks or months.

Orion's FAA Heritage Avionics Hardware & Software Design

The FAA has successfully used IMA techniques, primarily via Honeywell technology, to independently develop and test partitioned modules of flight SW across vendors at dispersed locations. This paradigm underwent significant analysis by NASA, as it was an unusual form of conducting a SW development program since NASA's earlier Shuttle experience base and culture.



Notional Orion IMA Architecture

Backup vs. Redundancy: Self Checking Pair (SCP) Hardware Architecture

Modern advances in both computer hardware and software engineering have allowed NASA to approach the problem of designing a safe architecture for human space flight, and specifically Orion, that differs from past designs like the redundant GPCs used in the Shuttle Program. The Orion Program utilizes an approach borrowed from the commercial airline industry and relies upon an Integrated Modular Avionics (IMA) architecture to insure safe operations. Rather than using multiple computers to implement the requirements of individual subsystems Orion's IMA architecture moves most of the computing requirements for various systems onto a single processor that is divided into separate and protected time and memory space partitions. The structure and reliability of this partitioning scheme is defined and guaranteed by the ARINC 653 architecture that is the basis of the Orion computing architecture.

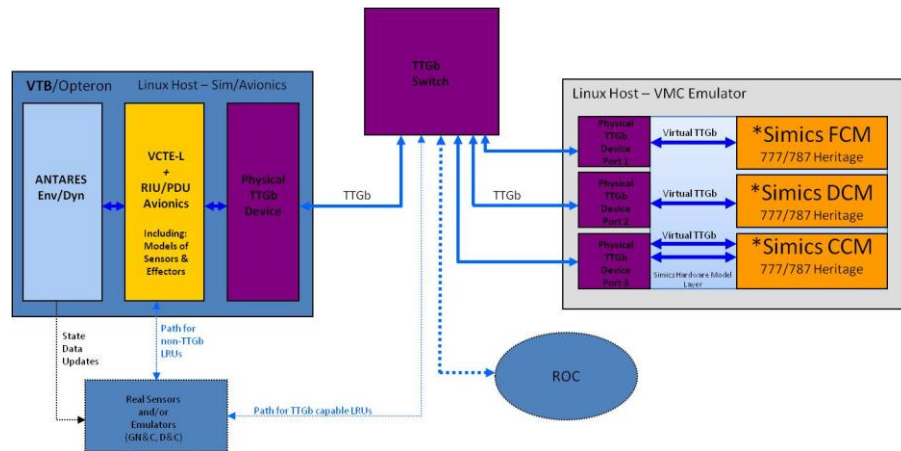
This approach allows major sets of functionality to be concentrated on a single avionics box rather than across multiple avionics boxes. While the Shuttle relied on the GPCs to compare outputs across multiple computers, Orion relies on a dual processor that executes instructions in lockstep across the two redundant cores and compares computations internal to the computer. If a failure is detected, for example due to a single event upset or a hardware failure, the dual lockstep processor shuts itself down and allows a second copy of the avionics box to take over. This dual lockstep processor is known as a Self-Checking Pair (SCP) as is a basis for satisfying the redundancy and reliability requirements needed to support human space flight.

While a second redundant computer is implemented for critical functionality (example: GN&C, life support) the two computers do not vote as the Shuttle GPCs did. Instead the concept of a primary and a secondary computer is implemented. If the primary computer detects a fault in it processing it will release control and allow the secondary computer to take control. Both computers receive the same inputs and compute the same outputs but only the current primary computer is allowed to command and control effectors and sensors.

The Orion Program does not rely on the same type of backup computer as implemented in the Shuttle program. While the Shuttle used the same hardware platform for the primary and backup computers, the Orion backup computer, if implemented, would use different avionics hardware with severely limited capability designed only for insuring the safety of the crew and returning them to the Earth. Ideally the backup computer would allow enough functionality to safe the vehicle until the primary systems could be brought back online.

Multiple fidelity test platforms – Adapting Early to Unavailable Hardware to Progress FSW Development

The first platform that a SW module is subject to test on the Orion program is the desktop SW development platform that every programmer has access to. This platform includes all of the tools to design and develop SW algorithms, and a suite of simulation tools (Simics™) capable of executing low fidelity executions of newly developed algorithms. Simics™ is an approach to bring target HW simulation to the developer’s desktop SW development platform, based on contemporary advances in simulation technology.



Early Kedalion Test Bed Concept with Simulated Vehicle Management Computer (VMC)

Commercially known as the Honeywell (HI) Valfac Test Bench, the VTB platform was adapted first to successfully support the integration and testing of major SW subsystems on the Orion flight test program. The Orion instantiation of the VTB is a modification of HI’s commercial VTB concept to include support for Orion unique interfaces and target HW, including Time Triggered Gigabit Ethernet (TTGbE) and VMC flight computers to support medium fidelity testing and integration.

The VTB platform is conceived as a highly adaptable and reconfigurable rig to support the many variations of tests required by the Orion SW project. The ultimate variation is the highest fidelity SW test

platform known as the Super VTB. This test platform includes a dual string set of bus and target processors for testing redundancy and failover scenario's, as well as robust interface capability to access sensors/actuators allowing for initial subsystem vertical integration and test.

Commercial Tools vs. Custom Tools

Historically, NASA Human Spaceflight has relied on custom software tools (programming languages, operating systems, etc) to implement reliable systems. While necessary for the time, programs such as the Shuttle showed that custom software tools can be extremely expensive over the lifespan of a program. Modern NASA budgets and the availability of highly reliable commercial software tools has allowed NASA to "rethink" its approach to implementing reliable and safe software systems.

The Orion Program relies extensively on commercially available software tools, operating systems and programming languages to implement its software architecture. Just as the IMA approach was adopted from the commercial aircraft industry so was the selection of a real-time operating system which is used to implement that architecture. In addition, standard modern programming languages (primarily C++) are used instead of custom languages like HAL/S. The selection of commercial tools brings advantages over custom solutions. Modern development and test tools and techniques can be used and there is a much wider availability of software engineers in the workforce who are experienced with commercially available tools than would be available with custom solutions. Also, the maintenance of these commercially available software tools can be purchased rather than needing to be maintained internally to a program at enormous cost.

Code Generators: High Order Language (HOL) vs. Very High Order Language (VHOL)

In addition to adopting new system architectural designs that take advantage of modern hardware, the Orion Program (and the Kedalion lab) has adopted the use of modern advanced software development tools and design philosophy. With the use of modern design and development tools NASA hopes to reduce the cost of software system development and maintenance. Advances in software engineering have allowed for the incorporation of advanced design concepts into space vehicle design. Just as the use of a high order language like HAL/S on the Shuttle program brought advantages over previous use of assembly language, likewise, the use of modeling tools like UML and graphical model based development tools (such as Mathworks Matlab/Simulink) allows for an increase in the efficiency of software development and a reduction in the overall cost.

HAL/S was used on the Shuttle Program and allowed complex code to be developed and maintained much easier than if it had been written in a lower level language. But the HAL/S code did not stand alone. It was accompanied by volumes of design documents and requirement documents that detailed what the software was supposed to do and how it was implemented. With the use of modern model based development tools the software can be, in large part, generated directly from the requirements and model based designs.

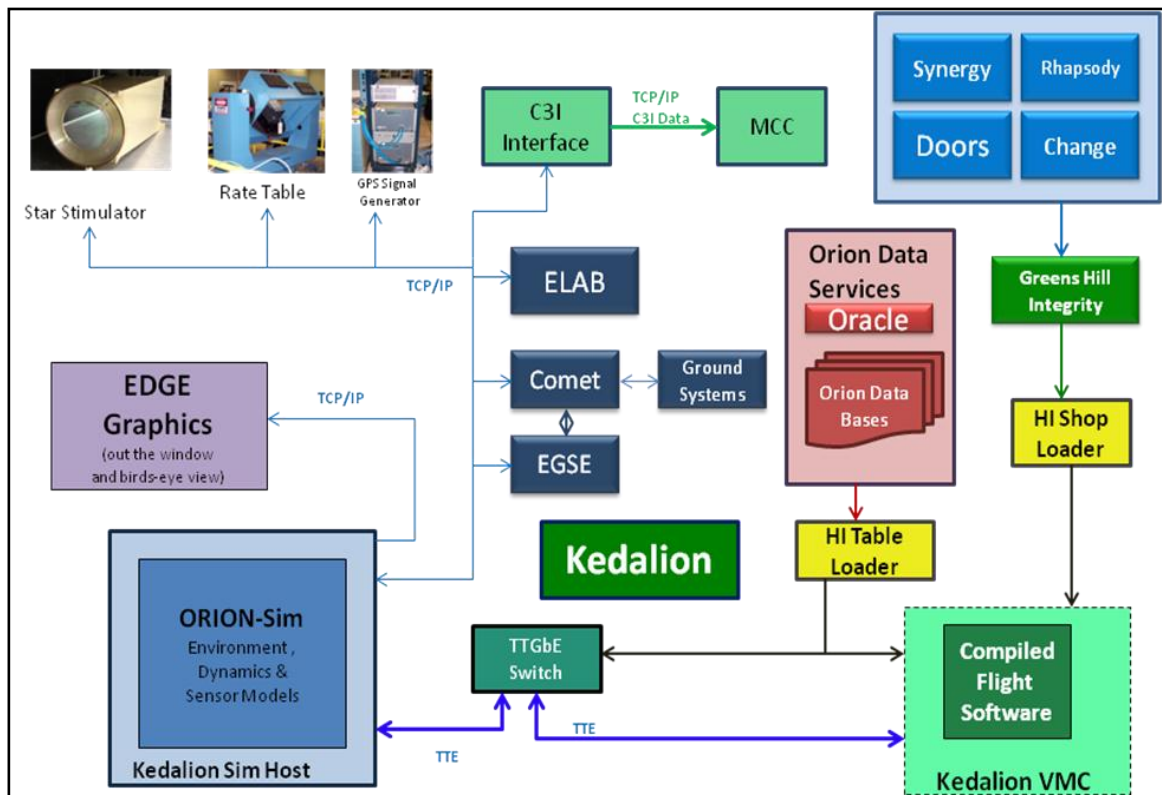
Lab architecture/components

The architecture of the Kedalion lab emphasizes flexibility, modularity and reconfiguration. The lab is populated with a variety of commercial hardware and software as well as some custom items and legacy assets from previous programs.

To establish a robust prototyping and development environment the lab was populated with a rich mix of development workstations that are powered by a variety of commercial computing operating systems including Microsoft Windows, Linux, IRIX and Solaris. By employing modern and legacy operating systems in the development environment engineers are able to maximize the use of modern tools as well as legacy software proven on prior programs.

More capable computing platforms are used to host simulation environments and emulation platforms. Embedded systems running real-time operating systems such as Green Hills Integrity and Windriver's VxWorks are used to establish more robust real-time computing platforms. Non-real-time Linux platforms are also used for simulation and emulation but through careful software design these systems can be used in integrated configurations with real-time hardware.

A variety of data bus configurations are also used to allow integration of a wide range of devices. These data bus types include MIL-STD-1553, commercial Ethernet, Time Triggered Ethernet, various serial interfaces and others as required. The concept of "bus adapters" is also used to allow legacy equipment to be mixed into configurations with more modern data buses.




Current Kedalion Schematic

The overall emphasis of this mixed bag of equipment is to allow all of these diverse lab configuration pieces to be integrated together into any as needed architecture to support prototyping and testing. The same equipment can then be easily and quickly reconfigured to implement some other configuration as required. Computing platforms can be repurposed and data busses can be reconfigured to produce the necessary configurations to support the engineer's needs.

Software Tool Chain (integrated SW development tool set)

Critical to understanding, designing and analyzing the avionics systems of a space vehicle is an understanding of the software tools used to build the vehicle's software products. For this purpose, the Kedalion lab has instantiated a modern and fully capable software development environment. While not limited to these tools the core of this development environment was based on the software tool chain selected by the prime contractor for the Orion program. For this reason, the IBM Rational Tool Suite was implemented as the foundation of the tool chain for the Kedalion lab. In addition, tools such as Mathworks' Matlab/Simulink, National Instruments Labview, Green Hills Multi, Microsoft Visual Studio, DiSTI's GL Studio as well as a variety of commercial software testing tools were added to the development environment to enhance the lab's tool chain.

- **Heavy use of COTS software tools**
 - IBM Rational Tool Suite
 - Mathworks ModelBased Development – Simulink
 - NI Labview
 - Windriver Simics & VxWorks RTOS
 - Green Hills Software – Multi, Integrity RTOS
- **Reuse of common NASA developed tools**
 - Trick Simulation Framework
 - EDGE – Graphics visualization package
- **Multipurpose workstations and servers**
 - Multiple OS environment
 - Linux, Windows, IRIX, Sun
- **Diverse hardware platforms**
 - Orion flight computers
 - Vehicle test bench
 - Multiple I/O interfaces available
 - Emulators and simulators used in place of unavailable hardware
 - Sensor integration platforms
 - Rate Table for inertial sensors
 - Star Field Stimulator for star trackers
 - GPS signal generator for GPS
 - Cockpit prototyping hardware
 - Engineering quality Display Units – low cost
 - Touch screen to virtualize display units
 - Hand controllers
 - “Out the window” views



Diverse Breadth of Hardware and Software Tools

Simulators/Emulators/Hardware/Software

Another key feature of the Kedalion lab is the use of high fidelity simulations to add realism to test configurations. Vehicle avionics components are integrated with complex high fidelity simulations to produce a realistic configuration that can produce the same environmental and dynamic data as would be experienced in actual flight. Interactions with the natural environment and full vehicle dynamics as well as subsystem behaviors are produced through these complex high fidelity simulations.

Many of the models used in these simulations are reused common models and legacy models developed in other programs. This model reuse is enabled by the use of a NASA developed simulation framework known as Trick. The Trick framework is used extensively in the Kedalion lab as the basis of most simulations and emulators. Trick provides an extensive suite of simulation development, execution, monitoring and post execution analysis tools. In addition, it provides a full-featured executive framework that can be configured and built into a variety of simulation architectures. It provides auto-coding services that allow model developers to focus on their domain expertise rather than on the development of a simulation structure. This combined with the easy reuse and reconfiguration of models allows for very rapid development of complex simulations. Trick also provides simple options for integration with hardware interfaces and, therefore, provides an ideal framework for implementing emulators. If actual hardware elements are too expensive or simply not available an emulation of the device can quickly be developed through the use of Trick.

Orion VMC Test Benches (VTB's) – Sim Host I/O Pump (SHIOP)

The VTB provides integration capabilities that allow flight software to be executed on actual flight processors while integrated with a full-featured simulation. This allows scenario based testing to be performed where the flight software is executed in a “test like you fly” approach. Since the flight computers for Orion are based on IMA architecture, this type of test configuration is able to produce the fidelity necessary to reproduce actual flight conditions, as the flight computer would perceive them. The high fidelity simulation combined with a flight data bus capable Input/Output (I/O) pump makes this type of test configuration possible. The simulation feeds the I/O pump with all of the inputs necessary to fully populate the flight data bus. This allows all of the elements of the IMA architecture on the flight computer to interact with the same interfaces as they would in actual flight.

Orion test bed architecture – adding Hardware as it matures

Early versions of Orion hardware are now becoming available to the project. Brass board flight computers have been added to the Kedalion configuration. Flight software has been moved off the simulated or emulated platforms and onto the brass board flight computers. Since the changes to the configuration have happened at the hardware interface points the disruption to the operation of the test configuration was minimal. As additional hardware becomes available it will be added to the configuration in a similar way.

The Kedalion test configuration will continue to be modified and upgraded as the Orion program matures. Hardware and software will be integrated with the other elements as it becomes available in an effort to keep the Kedalion configuration for continued support of program needs.



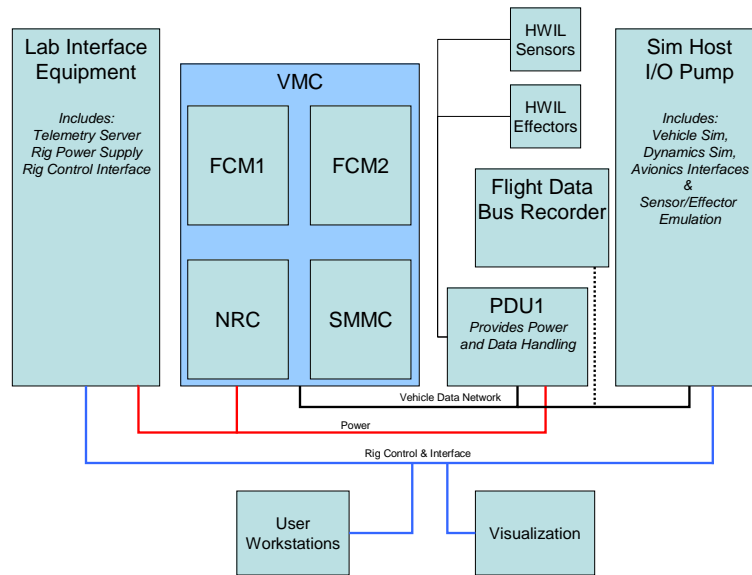
***Kedalion built Orion Ground Support Equipment
Before moving to Lockheed Test Facility***



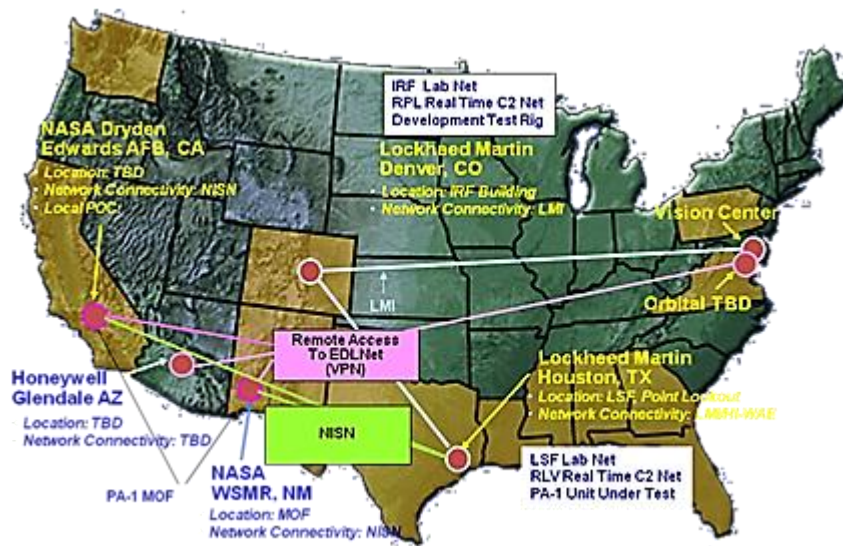
GSU Supporting an Integration Synch Point (ISP)

Integrated Test Lab's (ITL's)

An Integrated Test Lab (ITL) rig configuration is the highest fidelity test platform that Orion flight hardware and software would experience prior to actual testing regimens on the assembled vehicle. In this configuration, many actual vehicle avionics components are connected and loaded with associated flight software (FSW). Appropriate vehicle simulations, environmental/dynamics simulations, actual and/or simulated sensor/actuators, and stimulators provide as close to a “test like you fly” environment as can be assembled within a lab setting.



Simplified Diagram showing the configuration used in the Integrated Test Lab



Geographically Distributed Development and Test Environment

Extending the Kedalion beyond the Lab - HOTH

The Kedalion lab has continued to mature and expand over the last several years. The successful integration of hardware and software with steadily increasing levels of fidelity and maturity has demonstrated that the approach of the Kedalion lab can add great value to a project. In the constrained budgetary atmosphere of the aerospace industry the Orion program leadership at both NASA and Lockheed Martin looked for ways to expand the effectiveness of decreasing budgets and limited resources. The Kedalion lab was identified as an ideal means of “filling holes” in the program and augmenting areas that needed additional hardware and resources. For this purpose the mission of the Kedalion lab was altered to expand the scope of the lab to beyond its physical walls. In a way, the Kedalion lab was converted from just a lab to a type of approach or mindset.



***Kedalion built Test Bed with Honeywell Orion Flight Computer Brass Boards
Located in Lockheed Test Facility***

To fulfill the new mission, resources from the Kedalion lab have been diverted to other locations where they have been combined with other program resources and assets to better insure the success of the program. The combining of assets allowed for a more cost effective and timely way of implementing necessary integration and test facilities needed to meet the milestones of the aggressive development, integration and test schedule.

Assets, however, were not the only things diverted to other locations. Personnel from the Kedalion lab were also asked to work in field locations that could include any facility identified appropriate and necessary for reaching program goals. The Kedalion personnel were also asked to collaborate with the prime contractor team performing the Integration Sync Point (ISP) development and integration work. This meant that the Kedalion personnel would not be performing a program oversight role but would be working to “build the vehicle” in a “badge less” environment allowing for enhanced collaboration and cooperation between NASA and contractor personnel.

This new way of fulfilling the Kedalion mission, as a combined government and contractor team, has allowed the Kedalion personnel to make significant contributions to meeting program objectives and has allowed the flexible, dynamic approach of the Kedalion lab to be broadened beyond the walls of the lab.

Same Objective – Different Tools and Techniques

Less oversight and more insight

Traditionally, the government's roles for acquisition oversight placed individuals into highly segmented roles in relationship to the acquisition activity. The person responsible for system procurement was assumed to remain detached from the day-to-day design, build, and test activities. While accountabilities remain similar, a new level of teaming and integration has successfully emerged within the NASA contracting environment that is characterized by a common understanding that everyone has a responsibility as a "producer" and the general culture feels more embedded or "inline" than segmented and distinct.

New criteria for success

Part of the Government role is to act as an approver with oversight into the prime contractor activity. The prime contractor would traditionally be responsible for subcontracting and procurement of subsystems that integrate into the prime contract. Under the inline teaming arrangements, a slightly different approach is being taken.

- Strategically placed subcontracting clauses that allow the government to purchase system elements direct from a subcontractor, and GFE of these system elements into the prime contract portfolio
- Co-habitation (flatter working environments) that substantially mute the organizational hierarchy between NASA and the contracting community

Transitional challenges

In the early days of this transition, there was less trust between contractor and government personnel. If the government staff were more closely imbedded with the contractor staff in program execution, would the contractors be at greater risk in communicating openly about risk and problems, and difficulties? Time has shown that if the government is playing a more active role in acquisition on selective subsystems, then it has more insight into true program status – and the transition to operating in a more "native" model has paid off.

Space Flight Systems – Achieving Required Quality

The introduction of COTS components into modern space systems has been necessitated by the realities of current budgetary thresholds, multiple mission support, and competing models introduced by NASA commercial partners. There have been exponential advances in electronics and aircraft avionics systems in the years since the conception of STS systems. Coincidentally, the heritage of "borrowing" design elements from the adjacent aircraft industry was pioneered by STS designers. Orion is following this tradition with designs significantly influenced by aircraft avionics designs that have matured in the commercial marketplace.

Traditional methods to achieve high levels of quality standards on STS systems required extreme levels of inspection of hardware and software systems. These methods are disproportionately costly relative to the actual production of the systems under inspection. Additionally, some of the commercial components selected to implement current subsystems, are not conducive to traditional inspection methods. A paradigm shift is underway to combine a practical introduction of heritage based system component shelf-life, introducing legacy avionic system metrics, to augment traditional inspection methods to achieve a balanced yet affordable and safe space vehicle. This approach will have to cross boundaries within NASA culture that meets heritage quality standards, yet allows for a vibrant and affordable vehicle that meets today's space flight requirements.

Acknowledgments

The authors thank the Orion project management team including Mike Brieden and David Petri for their guidance, collaboration, and support. The authors further thank the Spacecraft Software Engineering Branch management lead Steven Fredrickson for encouragement and technical support in producing this paper. For support in sharing this information at the AIAA Space 2012 conference, the authors thank Rob Ambrose, as well as the overall Orion project management team.

A special thanks to John “Jack” Garman, who graciously collaborated with his vast knowledge of historical systems that reach back to early Apollo and STS systems, especially FSW avionics development and test systems.

References

"Modular Verification: Testing a Subset of Integrated Modular Avionics in Isolation", Christopher B. Watkins, 25th Digital Avionics Systems Conference (DASC), Portland, Oregon, October 2006.

Feiler, Peter H. "Challenges in Validating Safety-Critical Embedded Systems." *Proceedings of SAE International AeroTech Congress*. Seattle, WA (USA), November 2009.

<https://www.sae.org/technical/papers/2009-01-3284>

"ARINC 653 - An Avionics Standard for Safe, Partitioned Systems". WindRiver Systems/IEEE Seminar. August 2008. http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf. Retrieved 2009-05-30.

RTCA, Incorporated, document RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1, 1992.

"How the IBM Rational Team Unifying Platform integrates with IBM Development Tools", IBM Corporation, Document G507-0962-00, July 2005

"Orion Flight Software V&V and Kedalion Engineering Lab Oversight", AIAA Space 2010, Document AIAA 2010-8761

"Kedalion: NASA's Adaptable and Agile Hardware/Software Integration and Test Lab", AIAA Space 2011, Document AIAA-2011-7176