Chapter 7

Software and the Challenge of Flight Control[§]

Nancy G. Leveson
Aeronautics and Astronautics
MIT

Introduction

A mythology has arisen about the Shuttle software with claims being made about it being "perfect software" and "bug-free" or having "zero-defects,"[1,2] all of which are untrue. But the overblown claims should not take away from the remarkable achievement by those at NASA and its major contractors (Rockwell, IBM, Rocketdyne, Lockheed Martin, and Draper Labs) and smaller companies such as Intermetrics (later Ares), who put their hearts into a feat that required overcoming what were tremendous technical challenges at that time. They did it using discipline, professionalism, and top-flight management and engineering skills and practices. Many lessons can be learned that are still applicable to the task of engineering complex software systems today, both in aerospace and in other fields where software is used to implement critical functions.

Much has already been written about the detailed software design. This essay will instead take a historical focus and highlight the challenges and how they were successfully overcome as the Shuttle development and operations progressed. The ultimate goal is to identify lessons that can be learned and used on projects today. The lessons and conclusions gleaned are necessarily influenced by the experience of the author; others might draw additional or different conclusions from the same historical events.

To appreciate the achievements, it is necessary first to understand the state of software engineering at the time and the previous attempts to use software and computers in spaceflight.

Learning from Earlier Manned Spacecraft

Before the Shuttle, NASA had managed four manned spacecraft programs: Mercury, Gemini, Apollo, and Skylab that involved the use of computers. Mercury did not need an on-board computer. Reentry was calculated by a computer on the ground and the retrofire times and firing attitude were transmitted to the spacecraft while in flight.[3]

Gemini was the first U.S. manned spacecraft to have a computer onboard. Successful rendezvous required accurate and fast computation, but the ground tracking network did not cover all parts of the Gemini orbital paths. The type of continuous updates needed for some critical maneuvers therefore could not be provided by a ground computer. The Gemini designers also wanted to add accuracy to reentry and to automate some of the preflight checkout functions. Gemini's computer did its own self-checks under software control during prelaunch, allowing a reduction in the number of discrete test lines connected to launch vehicles and spacecraft. During ascent, the Gemini computer received inputs about the velocity and course of the Titan booster so that it would be ready to take over from the Titan's computers if they failed, providing some protection against a booster computer failure. Switchover could be either automatic or manual, after which the Gemini computer could issue steering and booster cutoff commands for the Titan. Other functions were also automated and, in the end, the computer operated during six mission phases: prelaunch, ascent backup, orbit insertion, catch-up, rendezvous, and reentry.[4]

[§] This chapter will appear in a forthcoming book (2013) *Space Shuttle Legacy: How We Did It/What We Learned* to be published by the AIAA and edited by Roger Launius, James Craig, and John Krige.

The computer for Gemini was provided by IBM, the leading computer maker at that time. Both NASA and IBM learned a great deal from the Gemini experience as it required more sophistication than any computer that had flown on unmanned spacecraft to that date. Engineers focused on the problems that needed to be solved for the computer hardware: reliability and making the hardware impervious to radiation and to vibration and mechanical stress, especially during launch.

At the time, computer programming was considered an almost incidental activity. Experts wrote the software in low-level, machine-specific assembly languages. Fortran had only been available for a few years and was considered too inefficient for use on real-time applications, both in terms of speed and machine resources such as memory. The instructions for the Gemini computer were carefully crafted by hand using the limited instruction set available. Tomayko suggests that whereas conventional engineering principles were used for the design and construction of the computer hardware, the software development was largely haphazard, undocumented, and highly idiosyncratic. "Many managers considered software programmers to be a different breed and best left alone."[3]

The requirements for the Gemini software provided a learning environment for NASA. The programmers had originally envisioned that all the software needed for the flight would be preloaded into memory, with new programs to be developed for each mission.[3] But the programs developed for Gemini quickly exceeded the available memory and parts had to be stored on an auxiliary tape drive until needed. Computers had very little memory at the time and squeezing the desired functions into the available memory became a difficult exercise and placed limits on what could be accomplished. In addition, the programmers discovered that parts of the software were unchanged from mission to mission, such as the ascent guidance backup function. To deal with these challenges, the designers introduced *modularization* of the code, which involved carefully breaking the code into pieces and reusing some of the pieces for different functions and phases of flight, a common practice today but not at that time.

Another lesson was learned about the need for software specifications. McDonnell-Douglas created a Specification Control Document, which was forwarded to the IBM Space Guidance Center in Oswego to validate the guidance equations. Simulation programs, written in Fortran, were used for this validation.

Despite the challenges and the low level of software technology at the time, the Gemini software proved to be highly reliable and useful. NASA used the lessons learned about modularity, specification, verification, and simulation in producing the more complex Apollo software. In turn, many of the lessons learned from Apollo were the basis for the successful procedures used on the Shuttle. Slowly and carefully NASA was learning how to develop more and more complex software for spacecraft.

After the success with Gemini, the software ambitions for Apollo stretched the existing technology even further. Although computer hardware was improving, it still created extreme limitations and challenges for the software designers. During Mercury, the ground-based computer complex supporting the flights performed one million instructions per minute. Apollo did almost 50 times that many—approaching one million instructions per second.[3] Today, the fastest computer chips can execute 50 billion instructions per second, or 50,000 times the speed of the entire Apollo computer complex. George Mueller, head of the NASA Office of Manned Space Flight, envisioned that computers would be "one of the basic elements upon which our whole space program is constructed."[5] But first, the problems of how to generate reliable and safe computer software had to be solved. The handcrafted, low-level machine code of Gemini would not scale to the complexity of later spacecraft.

In 1961, NASA contracted with Draper Labs (then called the MIT Instrumentation Lab) for the design, development, and construction of the Apollo guidance and navigation system, including software. The plans for using software in Apollo were ambitious and caused delays.

Much of the computing for Apollo was done on the ground. The final Apollo spacecraft was autonomous only in the sense it could return safely to Earth without help from ground control. Both the Apollo Command Module (CM) and the Lunar Exploration Module (LEM) included guidance and control functions, however. The CM's computer handled translunar and transearth navigation while the LEM's provided for autonomous landing, ascent, and rendezvous guidance. The LEM had an additional computer as part of the Abort Guidance System (AGS) to satisfy the NASA requirement that a first failure should not jeopardize the crew. Ground systems backed up the CM computer and its guidance system such that if

the CM system failed, the spacecraft could be guided manually based on data transmitted from the ground. If contact with the ground was lost, the CM system had autonomous return capability.

Real-time software development on this scale was new to both NASA and MIT—indeed to the world at that time—and was considered to be more of an art than a science. Both MIT and NASA originally treated software as a secondary concern, but the difficulty soon became obvious. When testifying before the Senate in 1969, Mueller described problems with the Apollo guidance software, calling Apollo's computer systems "the area where the agency pressed hardest against the state of the art," and warning that it could become the critical path item delaying the lunar landing.[5] He personally led a software review task force that, over a nine month period, developed procedures to resolve software issues. In the end, the software worked because of the extraordinary dedication of everyone on the Apollo program.

In response to the problems and delays, NASA created some important management functions for Apollo including a set of control boards—the Apollo Spacecraft Configuration Control Board, the Procedures Change Control Board, and the Software Configuration Control Board—to monitor and evaluate changes. The Software Configuration Control board, chaired for a long period by Chris Kraft, controlled all onboard software changes.[3]

NASA also developed a set of reviews for specific points in the development process. For example, the Critical Design Review (CDR) followed the preparation of the requirements definition, guidance equations, and engineering simulation of the equations and placed the specifications and requirements for a given mission under configuration control. The next review, a First Article Configuration Inspection (FACI) followed the coding and testing of the software and the production of a validation plan and placed the software under configuration control. The Customer Acceptance Readiness Review (CARR) certified the validation process after testing was completed. The Flight Readiness Review was the final step before clearing the software for flight. This review and acceptance process provided for consistent evaluation of the software and controlled changes, which helped to ensure high reliability and inserted much needed discipline into the software development process. The control board and review structure became much more extensive for the Shuttle.

Like Gemini, computer memory limitations caused problems, resulting in the abandonment of some features and functions and resulting in the use of tricky programming techniques to save others. The complexity of the resulting code led to difficulty in debugging and verification and therefore to delays. MIT's distance from Houston also created communication problems and compounded the difficulty in developing correct software.

When it appeared that the software would be late, MIT added more people to the software development process, which simply slowed down the project even more. This basic principle that adding more people to a late project makes it even later is well-known now, but it was part of the learning process at that time. Configuration control software was also late, leading to delays in supporting discrepancy[6] reporting. Another mistake made was to take shortcuts in testing when the project started to fall behind schedule.

The 1967 launch pad fire gave everyone time to catch up and fix the software, as later the Challenger and Columbia accidents would for the Shuttle software. The time delay allowed for significant improvements to be made. Howard "Bill" Tindall, who was NASA's watchdog for the Apollo software, observed at the time that NASA was entering a new era with respect to spacecraft software: no longer would software be declared complete in order to meet schedules, requiring the users to work around errors. Instead quality would be the primary consideration.[3] The Mueller-led task force came to similar conclusions and recommended increased attention to software in future manned space programs. The dynamic nature of requirements for spacecraft should not be used as an excuse for poor quality, it suggested. Adequate resources and personnel were to be assigned early to this "vital and underestimated area."[5] The Mueller task force also recommended ways to improve communication and make coding easier.

Despite all the difficulties encountered in creating the software, the Apollo software was a success. Mueller attributed this success to the "extensive ground testing to which every subsystem was subjected,

the simulation exercises which provided the crews with high fidelity training for every phase of the flights, and the critical design review procedures … fundamental to the testing and simulation programs."[5]

In the process of constructing and delivering the Apollo software, both NASA and the MIT Instrumentation Lab learned a lot about the principles of software engineering for real-time systems and gained important experience in managing a large real-time software project. These lessons were applied to the Shuttle. One of the most important lessons was that software is more difficult to develop than hardware. As a result,

- software documentation is crucial,
- verification must proceed through a sequence of steps without skipping any to try to save time,
- requirements must be clearly defined and carefully managed,
- good development plans should be created and followed, and
- adding more programmers does not lead to faster development.

NASA also learned to assign experienced personnel to a project early, rather than using the start of a project for training inexperienced personnel.

Skylab was the final large software effort prior to the Shuttle. Skylab had a dual computer system for attitude control of the laboratory and pointing of the solar telescope. The software contributed greatly to saving the mission during the two weeks after its troubled launch and later helped control Skylab during the last year before reentry.[3] The system operated without failure for over 600 days of operation. It was also the first onboard computer system to have redundancy management software. Learning from their previous experience, the software development followed strict engineering principles, which were starting to be created at that time in order to change software development from a craft to an engineering discipline.

There were some unique factors in Skylab that made the problem somewhat easier than Apollo. The software was quite small, just 16K words, and the group of programmers assigned to write it was correspondingly small. There were never more than 75 people involved, not all of whom were programmers, which made the problems of communication and configuration control, which were common in larger projects, less important. Also, IBM assigned specialists in programming to the software in contrast to Draper Labs, which used spacecraft engineering experts. Draper and IBM had learned that the assumption that it was easier to teach an engineer to program than to teach a programmer about spacecraft was wrong.[3]

Limitations in memory were again a problem. The software resulting from the baselined requirements documents ranged from 9,000 words to 20,000 in a machine with only 16,000 words of memory. Engineers made difficult choices about where to cut. Memory limitations became the prime consideration in allowing requirements changes, which ironically may have actually contributed to the success of the software given the difficult problems that ensue from requirements changes.

Tomayko concludes that the Skylab program demonstrated that careful management of software development, including strict control of changes, extensive and preplanned verification activities, and the use of adequate development tools, results in high quality software with high reliability.[3] However, the small size of the software and the development teams was also an important factor.

Challenges to Success for the Shuttle

The Space Shuttle design used computers in much more ambitious ways than previous spacecraft. The Shuttle onboard computers took the lead in all checkout, guidance, navigation, systems management, payload, and powered flight functions. These goals pushed the state of the art at the time so NASA was required to create new solutions to problems that had not previously been faced. At the same time, they realized that conservatism was important to success. The result was often a compromise between what was desirable and what could be done with confidence.

The full scope of what needed to be accomplished was not recognized at first. NASA engineers estimated the size of the flight software to be smaller than that of Apollo. The cost was also vastly underestimated: Originally, NASA thought the cost for development of the Shuttle software would be

$20,000,000 but the agency ended up spending $200,000,000 just in the original development and $324 million by 1991.[7] In 1992, NASA estimated that $100,000,000 a year was being spent on maintaining the onboard software.[8] Note that the onboard software was only a small part of the overall software that needed to be developed. There was four times as much ground support software used in construction, testing, simulation, and configuration control.

NASA had learned from Apollo and the earlier spacecraft just how important software development was in spacecraft projects and that quality had to be a goal from the beginning. Software could not just be declared complete in order to meet schedules, requiring users to work around errors[3], although this goal was not completely realized as witnessed by the number of software-related waivers and user notes [9] during Shuttle operations. By STS-7 in June 1983, over 200 pages of such exceptions and their descriptions existed.[33] Some were never fixed, but the majority were addressed after the Challenger accident in January 1986 when flights were temporarily suspended.

Many difficult challenges had to be overcome in creating and maintaining the Shuttle software, including continuing hardware and memory limitations, continually evolving software requirements, project communication and management, software quality, and computer hardware reliability. How NASA and its contractors met those challenges is described later. But first, to understand their solutions, it is necessary to have some basic information about the Shuttle software architecture.

Onboard Software Architecture

There are basically three computer systems on board the Shuttle: a computer that controls the Space Shuttle Main Engines (SSME), the primary avionics computer, and a computer that backs up the primary computer.

The Main Engine Control (MEC) software, built by Rocketdyne and managed by Marshall Space Flight Center, was a "first" in space technology. The Shuttles three main liquid-propellant engines were the most complex and "hottest" rockets ever built.[3] The Shuttle engines can adjust flow levels, can sense how close to exploding they are, and can respond in such a way as to maintain maximum performance at all times. This design would have been impossible without the use of a digital computer to control the engines. But it also provided huge challenges to the software designers.

After studying the problem, Rocketdyne and Marshall decided to use a distributed approach. By placing controllers at the engines themselves, complex interfaces between the engine and vehicle could be avoided. Also the high data rates needed for active control were best handled by a dedicated computer. In addition, they decided to use a digital computer controller rather than an analog controller because software would allow for more flexibility. As the control concepts for the engines evolved over time, digital systems could be developed faster, and failure detection would be simpler.[10]

The MEC software operated as a real-time system with a fixed cyclic execution cycle. The requirement to control a rapidly changing engine environment led to the need for a high-frequency cycle. Each major cycle started and ended with a self test. It then executed engine control tasks, read sensor readings, performed engine limit monitoring tasks, provided outputs, then read another round of input sensor data, checked internal voltage, and finally performed a second self test. Some free time was built into the cycle to avoid overruns into the next cycle. Direct memory access of engine component data by the primary avionics software was allowed to ensure that the main engine controller did not waste time handling data requests. NASA managers adopted a strict software engineering approach to developing and maintaining the MEC software as they did for all the Shuttle software.

The second main onboard computer, the Primary Avionics Software System (PASS), provided functions used in every phase of a mission except for docking, which was a manual operation provided by the crew.[7] PASS is also sometimes referred to as the on-board data processing system (DPS). The two main parts of the software are (1) an operating system and code that provided essential services for the computer (called the FCOS or Flight Computer Operating System) and (2) the application software that ran the Shuttle. The application software provided guidance navigation, and flight control; vehicle systems management (including payload interfaces); and vehicle checkout.

Because of memory limitations, the PASS was divided into major functions, called OPS (Operational Sequences) reflecting the mission phases (preflight, ascent, on-orbit, and descent). The OPS were divided into modes. Transition between major modes was automatic, but transition between OPS was normally initiated by the crew and required that the OPS be loaded from the magnetic tape-based MMU (Mass Memory Unit). Common data used by more than one OPS was kept in the computer memory continuously and was not overlaid.[11]

Within each OPS, there were special functions (SPECs) and display functions (DISPs), which were supplemental functions available to the crew in addition to the functions being performed by the current OPS. Because SPECs and DISPs had lower priority than the regular OPS functions, they were kept on tape and rolled into memory when requested if a large OPS was in memory.

Originally the FCOS was to be a 40-millisecond time-slice operating system, but a decision was made early to convert it into a priority-driven system. If the processes in a time-slice system get bogged down by excessive input/output operations, they tend to slow down the total process operation. In contrast, priority systems degrade gracefully when overloaded. The actual FCOS created had some features of both types, with cycles similar to Skylab, but which could be interrupted for higher-priority tasks.

The PASS needed to be reconfigured from flight to flight. A large amount of flight-specific data was validated and installed into the flight software using automated processes to create a flight load. The flight load was tested and used in post-release simulations.[12] About 10 days before flight, a small number of low-risk updates were allowed (after retesting). In addition, an automated process was executed on the day of launch to update a small set of environmental factors (e.g., winds) to adapt the software to the conditions for that particular day.

A third computer, the Backup Flight System (BFS), was used to backup the PASS for a few critical functions during ascent and reentry and for maintaining vehicle control in orbit. This software was synchronized with the primary software so it could monitor the PASS. The BFS used a time-slice operating system,[13] which led to challenges in synchronizing the PASS and BFS and ultimately led to a delay of the first launch, described later in Section 3.3. If the primary computers failed, the mission commander could push a button to engage the backup software. One of the features of this system was that the mission commander had to make a decision very quickly about switching to the BFS—the BFS could only remain in a "ready" stage for a short time after failure of the PASS. It was not possible to switch back from BFS to PASS later. In addition, the BFS had to "stop listening" whenever it thought the PASS might be compromising the data being fetched so that it would not also be polluted.

Originally the BFS was intended to be used only during pre-operations testing but was extended to STS-4 and later for the life of the Shuttle. It was never actually engaged in flight.

Support for developing the PASS software (including testing and crew training) was provided in a set of ground facilities. The Software Production Facility (SPF) provided a simulation test bed that simulated the flight computer bus interface devices, provided dynamic access and control of the flight computer memory, and supported digital simulation of the hardware and software environments.[14] SPF requirements were defined early by NASA and all new capabilities required NASA approval. IBM maintained this facility while Rocketdyne was responsible for the MEC testing

After development of the software for a mission was completed, testing and simulation continued at the Shuttle Avionics Integration Laboratory (SAIL), which was designed to test the actual Shuttle hardware and flight software in a simulated flight environment and with a full cockpit for human-in-the-loop testing and integration testing with other flight and ground systems. SAIL was used to verify that the flight software loads were compatible with hardware interfaces, the flight software performed as designed, and the flight software was compatible with mission requirements. Major contractors involved in the SAIL included Rockwell-Downey, Lockheed Martin, and Draper Labs.

This architecture was designed to overcome some of the limitations and challenges for real-time software at the time, as discussed in the next subsections.

Hardware and Memory Limitations

In Gemini and Apollo, important functions had to be left out of the software due to lack of adequate computer memory. NASA and its contractors struggled to overcome these limitations. The Apollo software, for example, could not minimize fuel expenditures or provide the close guidance tolerance that would have been possible if more memory had been available.[3] Much development time was spent simply deciding which functions could be eliminated and how to fit the remainder in memory.

While computer memory had increased by the early 1970's from the tiny computers of previous spacecraft, there were still severe limits in how many instructions could be in memory at the same time. The techniques developed to share memory created additional system complexity. Like the earlier spacecraft, much effort in the Shuttle software development was expended in reducing the size of the software that had to be resident in the computer at any given time.

The earliest onboard computers had only a 4K to 16K word (8-bit) memory. In comparison, the main memory in the PASS computers used for the first Shuttle flights contained 106K 32-bit words. The onboard code required 400K words of memory, however, including both instructions and data.[15] The operating system and displays occupied 35K of memory at all times. With other functions that had to be resident, about 60K of the 106K was left for application programs. The solution was to delete functions, to reduce execution rates, and to break the code into overlays with only the code necessary to support a particular phase of the flight loaded into computer memory (from tape drives) at any time. When the next phase started, the code for that phase was swapped in.

The majority of effort went into the ascent and descent software. By 1978, IBM reduced the size to 116K, but NASA headquarters demanded that it be reduced to 80K. It never got down to that size, but it was reduced to below 100K by moving functions that could wait until later operational sequences. Later, the size increased again to nearly the size of the memory as changes were made.

Besides the effort expended in trying to reduce the size of the software, there were other important implications imposed by the memory restrictions. For example, requests for extra functions usually had to be turned down. The limited functionality in turn impacted the crew.

Shuttle crew interfaces are complex due to the small amount of memory available for the graphics displays and other utilities that would make the system more useful and simpler for the users. As a result, some astronauts have been very critical of the Shuttle software. John Young, the Chief Astronaut in the early 1980s, complained "What we have in the Shuttle is a disaster. We are not making computers do what we want."[3] Flight trainer Frank Hughes also complained, saying that "the PASS doesn't do anything for us"[3] when noting that important functions were missing. Both said "We end up working for the computer rather than the computer working for us."[3]

Some of the astronaut interface problems stemmed from the fact that steps usually preprogrammed and performed by computers must be done manually in the Shuttle. For example, the reconfiguration of PASS from ascent mode to on-orbit mode has to be done by the crew, a process that takes several minutes and needs to be reversed before descent. The response of John Aaron, one of NASA's designers of the PASS interface, was that management "would not buy" simple automatic reconfiguration schemes and, even if they had, there was no computer memory to store them.[3]

The limited memory also required many functions to be displayed concurrently on a screen due to the large amount of memory such displays require. As a result, many screens were so crowded that reading them quickly was difficult. That difficulty was compounded by the primitive nature of graphics available at the time. These interface limitations along with others added to the potential for human error.

Some improvements suggested by the astronauts were included when the Shuttle computers were upgraded in the late 1980s to computers with 256K memory. A further upgrade, the Cockpit Automation Upgrade (CAU) was started in 1999 but was never finished because of the decision to retire the Shuttle.

The astronauts took matters into their own hands by using small portable computers to augment the onboard software. The first ones were basically programmable calculators, but beginning with STS-9 in December, 1983, portable microcomputers with graphics capabilities were used in flight to display ground stations and to provide functions that were impractical to add to the primary computers due to lack of memory.

The backup computer (BFS) software required 90,000 words of memory so memory limitations were never a serious problem. But memory limitations did create problems for the Main Engine Control (MEC) computers and its software. The memory of the MEC computers was only 16K, which was not enough to hold all the software originally designed for it. A set of preflight checkout functions were stored in an auxiliary storage unit and loaded during the countdown. Then, at T–30 hours, the engines were activated and the flight software was read from auxiliary memory. Even with this storage saving scheme, less than 500 words of the 16K were unused.

The Challenge of Changing Requirements

A second challenge involved requirements. The software requirements for the Shuttle were continually evolving and changing, even after the system became operational and throughout its 30-year operational lifetime. Although new hardware components were added to the Shuttle during its lifetime, such as GPS, the Shuttle hardware was basically fixed and most of the changes over time went into the computers and their software. NASA and its contractors made over 2,000 requirements changes between 1975 and the first flight in 1981. Even after first flight, requirements changes continued. The number of changes proposed and implemented required a strict process to be used or chaos would have resulted.

Tomayko suggests that NASA lessened the difficulties by making several early decisions that were crucial for the program's success: NASA chose a high-level programming language, separated the software contract from the hardware contract and closely managed the contractors and their development methods, and maintained the conceptual integrity of the software.[3]

Using a High-Level Language: Given all the problems in writing and maintaining machine language code in previous spacecraft projects, NASA was ready to use a high-level language. At the time, however, there was no appropriate real-time programming language available so a new one was created for the space shuttle program. HAL/S (High-order Assembly Language/Shuttle)[16,17,18] was commissioned by NASA in the late 1960s and designed by Intermetrics.

HAL/S had statements similar to Fortran and PL/1 (the most prominent programming languages used for science and engineering problems at the time) such as conditions (IF) and looping (FOR or WHILE) statements. In addition, specific real-time language features were included such as the ability to schedule and coordinate processes (WAIT, SCHEDULE, PRIORITY, and TERMINATE). To make the language more readable by engineers, HAL/S retained some traditional scientific notation such as the ability to put subscripts and superscripts in their normal lowered or raised position rather than forcing them onto a single line.

In addition to new types of real-time statements, HAL/S provided two new types of program blocks: COMPOOL and TASK. Compools are declared blocks of data that are kept in a common data area and are dynamically sharable among processes. While processes had to be swapped in and out because of the memory limitations, compools allowed the processes to share common data that stayed in memory.

Task blocks are programs nested within larger programs that execute as real-time processes using the HAL/S SCHEDULE statement. The SCHEDULE statement simplified the scheduling of the execution of specific tasks by allowing the specification of the task name, start time, priority, and frequency of execution.

HAL/S was originally expected to have widespread use in NASA, including a proposed ground-based version named HAL/G ("G" for ground), but external events overtook the language when the Department of Defense commissioned the Ada programming language. Ada includes the real-time constructs pioneered by HAL/S such as task blocks, scheduling, and common data. Ada was adopted by NASA rather than continuing to use HAL/S because commercial compilers were available and because the Department of Defense's insistence on its use seemed to imply it would be around for a long time.

The use of a high-level programming language, which allowed top-down structured programming, along with improved development techniques and tools have been credited with doubling Shuttle software productivity over the comparable Apollo development processes.

Separating the Hardware and Software Contracts with NASA Closely Managing the Software Development: IBM and Rockwell had worked together during the competition period for the orbiter contract. Rockwell bid on the entire spacecraft, intending to subcontract the computer hardware and software to IBM. To Rockwell's displeasure, NASA decided to separate the software contract from the orbiter contract. As a result, Rockwell still subcontracted with IBM for the computer hardware, but IBM had a separate software contract managed closely by Johnson Space Center.

Tomayko suggests several reasons for why NASA made this unusual division.[3] First, software was, in some ways, the most critical component of the Shuttle. It tied the other components together and, because it did not weigh anything in and of itself, it was often used to overcome hardware problems that would require extra mechanical systems and components. NASA had learned from the problems in the Apollo program about the importance of managing software development. Chris Kraft at Johnson Space Center and George Low at NASA Headquarters, who were both very influential in the manned space program at the time, felt that Johnson had the software management expertise (acquired during the previous manned spacecraft projects) to handle the software directly. By making a separate contract for the software, NASA could ensure that the lessons learned from previous projects would continue to be followed and accumulated and that the software contractors would be directly accountable to NASA management.

In addition, after operations began, the hardware remained basically fixed while the software was continually changing. As time passed, Rockwell's responsibilities as prime hardware contractor were phased out and the Shuttles were turned over to an operations group. In late 1983, Lockheed Corporation and not Rockwell won the competition for the operations contract. By keeping the software contract separate, NASA was able to continue to develop the software without the extreme difficulty that would have ensued by attempting to change the software developers while it was still being developed.

The concept of developing a facility (the SPF) at NASA for the production of the onboard software originated in a Rand Corporation memo in early 1970 which summarized a study of software requirements for Air Force space missions during the decade of the 1970s.[3] One reason for a government-owned and operated software "factory' was that it would be easier to establish and maintain security for Department of Defense payloads, which could require special software interfaces and control. More important, it would be easier to change contractors, if necessary, if the software library and development computers were government owned and on government property. Finally, having close control by NASA over existing software and new development would eliminate some of the problems in communication, verification, and maintenance encountered in earlier manned spacecraft programs. The NASA-owned but IBM run SPF had terminals connected directly to Draper Laboratory, Goddard Space Flight Center, Marshall Space Flight Center, Kennedy Space Flight Center, and Rockwell International. The SAIL played a similar role for prelaunch, ascent, and abort simulations as did the Flight Simulation Lab and the SMS for other simulations and crew training.

Maintaining Conceptual Integrity: The on-going vehicle development work did not allow an ideal software development process to be employed, where requirements are completely defined prior to design, implementation, and verification.

The baseline requirements were established in parallel with the Shuttle test facility development. Originally, it was assumed that these tests would only require only minor changes in the software. This assumption turned out to be untrue; the avionic integration and certification activities going on in Houston, at the Kennedy Space Center, and in California at Downey and Palmdale, such as the ALT tests,[19] resulted in significant changes in the software requirements. In many cases, NASA and its contractors found that the real hardware interfaces differed from those in the requirements, operational procedures were not fully supported, and additional or modified functions were required to support the crew.

The strategy used to meet the challenge of changing requirements had several components: rigorously maintained requirements documents, using a small group to create the software architecture and interfaces and then ensuring that their ideas and theirs alone are implemented (called "maintaining conceptual integrity"[20]), and establishing a requirements analysis group to provide a systems engineering interface between the requirements definition and software implementation worlds. The latter identified

requirements and design tradeoffs and communicated the implications of the trades to both worlds.[21] This strategy was effective in accommodating changing requirements without significant cost or schedule impacts.

Three levels of requirements documents were created. Levels A and B were written by Johnson Space Center Engineers and Level C, which was more of a design document, was the responsibility of Rockwell International. John Garman created the level A document, which described the operating system, application programs, keyboards, displays, other components of the software, and the interfaces to the other parts of the vehicle. William Sullivan wrote the Level B guidance, navigation, and control requirements, while John Aaron wrote the system management and payload specifications for the Level B document. They were assisted by James Broadfoot and Robert Ernull. Level B specifications differed from Level A in that they were more detailed in terms of what functions were executed when and what parameters were needed. Level B also defined what information was to be kept in the Hal/S COMPOOLS for use by different tasks. Level C, developed for the contractors to use in development, was completely traceable to the Level B requirements.

The very small number of people involved in the requirements development contributed greatly to their conceptual integrity and therefore the success of the Shuttle's software development effort.[3]

Early in the program, Draper Labs was retained as a consultant to NASA on requirements development because they had learned the hard way on Apollo and had become leaders in software engineering. Draper provided a document on how to write requirements and develop test plans, including how to develop highly modular software. Draper also wrote some of the early Level C requirements as a model for Rockwell. Rockwell, however, added a lot of implementation detail to the Draper requirements and delivered basically detailed design documents rather than requirements. These documents were an irritation for IBM, which claimed that they told IBM too much about how to do things rather than just what to do. Tomayko interviewed some IBM and NASA managers who suspected that Rockwell, miffed when the software contract was taken away from them, delivered incredibly detailed requirements because they thought that if they did not design the software, it would not be done right.[3] In response, IBM coded the requirements to the letter, which resulted in exceeding the available memory by over two times and demonstrating that the requirements were excessive.

Rockwell also argued for two years about the design of the operating system, calling for a strict time-sliced system with synchronization points at the end of each cycle. IBM, at NASA's urging, argued for a priority-interrupt driven design similar to the one used on Apollo. Rockwell, more experienced with time-sliced operating systems, fought this proposal from 1973 to 1975, convinced it would never work.[3] Eventually, Rockwell used a time-sliced system for the BFS while IBM used a priority-driven system for the PASS. The difference between the two designs caused complications in the synchronization process. In the end, because the backup must listen in on PASS operation in order to be ready to take over if necessary, PASS had to be modified to make it more synchronous.

The number of changes in the software requirements was a continuing problem but at least one advantage of having detailed requirements (actually design specifications) very early was that it allowed the use of some software during the early Shuttle hardware development process. Due to the size, the complexity, the still evolving nature of the requirements, and the need for software to help develop and test the Shuttle hardware, NASA and IBM created the software using incremental releases. Each release contained a basic set of capabilities and provided the structure for adding additional functions in later releases. Seventeen interim releases were developed for the first Shuttle flight, starting in 1977. The full software capability was provided after the ninth release in 1978, but eight more releases were necessary to respond to requirements changes and to identified errors.

NASA had planned that the PASS would involve a continuing effort even after first flight. The original PASS was developed to provide the basic capability for space flight. After the first flight, the requirements evolved to incorporate increased operational capability and changing payload and mission requirements. For example, over 50% of the PASS modules changed during the first 12 flights in response to requested enhancements.[3] Among the Shuttle enhancements that changed the flight control requirements were changes in payload manifest capability, MEC design, crew enhancements, addition of

an experimental autopilot for orbiting, system improvements, abort enhancements (especially after the Challenger accident), provisions for extended landing sites, and hardware platform changes (including the integration of GPS). The Challenger accident was not related to software, but it required changes in the software to support new safety features.

After STS-5, the maintenance and evolution process was organized by targeting requested changes to specific software Operational Increments (OIs). Software changes were generally made to correct deficiencies, to enhance the software's capabilities, or to tailor it to specific mission requirements. OI's were scheduled updates of the primary and backup software. Each OI was designed to support a specific number of planned missions. The OIs included required additions, deletions, and changes to thousands of lines of code. OIs were scheduled approximately yearly but they could take up to 20 months to complete so usually multiple OIs were being worked on at the same time.

All requested changes were submitted to the NASA Shuttle Avionics Software Control Board (SASCB). The SASCB ranked the changes based on program benefits including safety upgrades, performance enhancements, and cost savings. A subset of potential changes were approved for requirements development and placed on the candidate change list. Candidates on the list were evaluated to identify any major issues, risks, and impacts[22] and then detailed size and effort estimates were created. Approved changes that fit within the available resources were assigned to specific OIs.

Once the change was approved and baselined, implementation was controlled through the configuration management system, which identified (a) the approval status of the change, (b) the affected requirements functions, (c) the code modules to be changed, and (d) the builds (e.g., operational increment and flight) for which the changed code was scheduled. Changes were made to the design documentation and the code as well as to other maintenance documentation used to aid traceability.[23]

Communication Challenges

A third challenge involved project communication. As spacecraft grew in complexity as well as the number of engineers and companies involved, communication problems increased. The large number of computer functions in the Shuttle meant that no one company could do it all, which increased the difficulty in managing the various contractors and fostering the required communication. One of the lessons learned from Apollo was that having the software developed by Draper Labs at a remote site reduced informal exchanges of ideas and created delays. To avoid the same problems, the developers of the Shuttle software were located in Houston.

In response to the communication and coordination problems during Apollo development, NASA had created a control board structure. A more extensive control board structure was created for the Shuttle. The results, actions, and recommendations of the independent boards were coordinated through a project baselines control board, which in turn interfaced with the spacecraft software configuration control board and the orbiter avionics software control board. Membership on the review boards included representatives from all affected project areas, which enhanced communication among functional organizations and provided a mechanism to achieve strict configuration control. Changes to approved configuration baselines, which resulted from design changes, requirements change requests, and discrepancy reports, were coordinated through the appropriate boards and ultimately approved by NASA. Audits to verify consistency between approved baselines and reported baselines were performed weekly by the project office.

Finally, the review checkpoints, occurring at critical times in development, that had been created for Apollo were again used and expanded.

Quality and Reliability Challenges

The Shuttle was inherently unstable, which means it could not be flown manually even for short periods of time during either ascent or reentry without full-time flight control augmentation.[23] There were also vehicle sequencing requirements for Space Shuttle Main Engine and Solid Rocket Booster ignition, launch pad release and liftoff operations, and Solid Rocket Booster and External Tank separation, which must occur within milliseconds of the correct time. To meet these requirements, the Shuttle was one of

the first spacecraft (and vehicles in general) to use a fly-by-wire flight control system[23]: In such systems, there are no mechanical or hydraulic linkages connecting the pilot's control devices to the control surfaces or reaction control system thrusters. Because sensors and actuators had to be positioned all over the vehicle, the weight of all the wire became a significant concern and multiplexed digital data buses were used.

The critical functions provided by digital software and hardware led to a need for high confidence in both. NASA used a fail-operational/fail-safe concept which meant that after a single failure of any subsystem, the Shuttle must be able to continue the mission. After two failures of the same subsystem, it must still be able to land safely.

Essentially there are two means for the "failure" of digital systems. The first is the potential for the hardware on which the software executes to fail in the same way that analog hardware does. The protection designed to avoid or handle these types of digital hardware failures is similar, and often involves incorporating redundancy.

In addition to the computer hardware failing, however, the software (which embodies the system functional design) can be incorrect or include behaviors that are unsafe in the encompassing system. Software is pure design without any physical realization and therefore "fails" only by containing systematic design defects. In fact, software can be thought of as *design abstracted away from its physical representation*, that is, software (when separated from the hardware on which it is executed) is pure design without any physical realization of that design. While this abstraction reduces many physical limits in design and thus allows exciting new features and functions to be incorporated into spacecraft that could not be achieved using hardware alone, it also greatly increases potential complexity and changes the types of failure modes. With respect to fault tolerance, potentially unsafe software behavior always stems from pure design defects so redundancy—which simply duplicates the design errors—is not effective. While computer hardware reliability can depend on redundancy, dealing with software errors must be accomplished in other ways.

Computer Hardware Redundancy on the Shuttle

To ensure fail-operational/fail-safe behavior in the MEC (main engine controllers), redundant computers were used for each engine. If one engine failed, the other would take over. Failure of the second computer led to a graceful shutdown of the affected engine. Loss of an engine did not create any immediate danger to the crew as demonstrated in a 1985 mission where an engine was shut down but the Shuttle still achieved orbit.[3] Early in a flight, the orbiter could return to a runway near the launch pad, Later in the flight, it could land elsewhere. If the engine failed near orbit, it could still be possible to achieve an orbit and modify it using the orbital maneuvering system engines.

The redundant MEC computers were not synchronized. Marshall Space Flight Center considered synchronizing them, but decided the additional hardware and software overhead was too expensive.[3] Instead, they employed a design similar to that used in Skylab, which was still operating at the time the decision was made. Two watchdog timers were used to detect computer hardware failures, one watchdog incremented by a real-time clock and the other by a clock in the output electronics. Both were reset by the software. If the timers ran out, a failure was assumed to have occurred and the redundant computer took over. The time out was set for less than the time of a major cycle (18 milliseconds).

The MEC had independent power, central processors and interfaces but the I/O devices were cross strapped such that if Channel A's output electronics failed, then Channel B's could be used by Channel A's computer.

Packaging is important for engine controllers as they are physically attached to an operating rocket engine. Rocketdyne bolted early versions of the controller directly to the engine, which resulted in vibration levels of up to 22g and computer failures. Later, a rubber gasket was used to reduce the levels to about 3–4g. The circuit cards within the computer were held in place by foam wedges to reduce vibration problems further.[3]

When the original MEC hardware was replaced with a new computer with more memory, the new semiconductor memory had additional advantages in terms of speed and power consumption, but did not

have the ability to retain data when power was shut off. To protect against this type of loss, the 64K memory was duplicated and each loaded with identical software. Failure of one memory chip caused a switchover to the other. Three layers of power also provided protection from losing memory. The first layer was the standard power supply. If that failed, a pair of 28-volt backup supplies, one for each channel, was available from other system components. A third layer of protection was provided by a battery backup that could preserve memory but not run the processor. The upgraded PASS computers also used semiconductor memory, with its size, power, and weight advantages, and solutions had to be created to protect against the stored programs disappearing if power was lost, although a different solution was devised for the PASS computer memory.

Like the MEC, PASS used redundancy to protect against computer hardware failures, but the designers used an elaborate synchronization mechanism to implement the redundancy. Again, the objective was fail-operational/fail-safe. To reach this goal, critical PASS software was executed in four computers that operated in lockstep while checking each other. If one computer failed, the three functioning computers would vote it out of the system. If a second computer failed, the two functioning computers took over and so on. A minimum of three computers is needed to identify a failed computer and continue processing. A fourth computer was added to accommodate a second failure.

The failure protection did not occur in all flight phases. PASS was typically run in all four redundant computers during ascent and reentry and a few other critical operations. During most orbital operations, the guidance, navigation, and control software was run on one computer while the system management software was run on a second computer. The remaining three computers (including the one running the BFS) were powered down for efficiency.[23]

Even when all four computers were executing, depending on the configuration, each of the computers was given the ability to issue a subset of the commands. This partitioning might be as simple as each controlling a separate piece of hardware (e.g, the reaction control jets) or more complex. This redundancy scheme complicated the design as reallocation of some functions had to occur if one or more of the computers failed. Input data also had to be controlled so that all the computers received identical information from redundant sensors even in the face of hardware failures.[23]

Synchronization of the redundant computers occurred approximately 400 times a second. The operating system would execute a synchronization routine during which the computers would compare states using three cross-strapped synchronization lines. All of the computers had to stop and wait for the others to arrive at the synchronization point. If one or more did not arrive in a reasonable amount of time, they were voted out of the set. Once the voting was complete, they all left the synchronization point together and continued until the next synchronization point. While the failed computer was automatically voted out of the set if its results did not match, it had to be manually halted by the astronauts to prevent it from issuing erroneous instructions. The capability to communicate with the hardware the failed computer was commanding was lost unless the DPS was reconfigured to pick up the busses lost by the failed computer.[23]

The BFS ran on only one computer and therefore was not itself fault tolerant with respect to hardware failures. The only exception was that a copy of its software was stored in the mass memory unit so that another computer could take over the functions of the backup computer in case of a BFS computer failure.

The same software was run on the four independent computers so the hardware redundancy scheme used could not detect or correct software errors. In the 1970s (when the Shuttle software was created), many people believed that “diversity” or providing multiple independently developed versions of the software and voting on the results would lead to very high reliability. Theoretically, the BFS was supposed to provide fault tolerance for the PASS software because it was developed separately (by Rockwell) from the PASS. In addition, a separate NASA engineering directorate, not the on-board software division, managed the Rockwell BFS contract.

In reality, using different software developed by a different group probably did not provide much protection. Knight and Leveson, in the mid-1980s showed that multiple versions of software are likely to contain common failure modes even if different algorithms and development environments are used.[24] Others tried to demonstrate that the Knight and Leveson experiments were wrong, but instead confirmed

them.[25] People make mistakes on the hard cases in the input space; they do not make mistakes in a random fashion.

In addition, almost all the software-related spacecraft losses in the past few decades (and, indeed most serious accidents related to erroneous software behavior) involved specification or requirements flaws and not coding errors [26,27]. In these accidents, the software requirements had missing cases or incorrect assumptions about the behavior of the system in which the software was operating. Often there was a misunderstanding by the engineers of the requirements for safe behavior, such as an omission of what to do in particular circumstances or special cases that were not anticipated or considered. The software may be "correct" in the sense that it successfully implements its requirements, but the requirements may be unsafe in terms of the specified behavior in the surrounding system, the requirements may be incomplete, or the software may exhibit unintended (and unsafe) behavior beyond what is specified in the requirements. Redundancy or even multiple versions that implement the same requirements do not help in these cases. If independently developed requirements were used for the different versions, there would be no way that they could vote on the results because they would be doing different things.

Although the BFS was never engaged to take over the functions of a failed PASS during a Shuttle flight, the difficulty in synchronizing the four primary computers with the BFS did lead to what has been called "The Bug Heard Round the World"[28] when the first launch was delayed due to a failed attempt to synchronize the PASS computers and the BFS computer. The BFS "listened" to all the inputs and some outputs to and from the PASS computers so it will be ready to take over if switched in by the astronauts. Before the launch of STS-1, the BFS refused to "sync" up with (start listening to) some of the PASS data traffic. The problem was that a few processes in the PASS were occurring one cycle early with respect to the others. The BFS was programmed to ignore all data on any buses for which it heard unanticipated PASS data fetches in order to avoid being polluted by PASS failures. As a result, the BFS stopped listening.

The Approach to Software Quality on the Shuttle

Because redundancy is not effective for requirements and design errors (the only type of error that software has), the emphasis was on avoiding software errors through use of a rigorous process to prevent introducing them and extensive testing to find them if they were introduced.

Using the management and technical experience gained in previous spacecraft software projects and in ALT, NASA and its contractors developed a disciplined and structured development process. Increased emphasis was placed on the front end of development, including requirements definition, system design, standards definition, top-down development, and creation of development tools. Similarly, during verification, emphasis was added on design and code reviews and testing. Some aspects of this process would be considered sophisticated even today, 30 years later. NASA and its contractors should be justly proud of the process they created over time. Several aspects of this process appear to be very important in achieving the high quality of the Shuttle software.

Extensive planning before starting to code: NASA controlled the requirements, and NASA and its contractors agreed in great detail on exactly what the code must do, how it should do it, and under what conditions. That commitment was recorded. Using those requirements documents, extremely detailed design documents were produced before a line of code was written. Nothing was changed in the specifications (requirements or design) without agreement and understanding by everyone involved. One change to allow the use of GPS on the Shuttle, for example, which involved changing about 6300 lines of code, had a specification of 2500 pages. Specifications for the entire onboard software fill 30 volumes and 40,000 pages.[23]

When coding finally did begin, top-down development was the norm, using stubs[29] and frequent builds to ensure that interfaces were correctly defined and implemented first, rather than finding interface problems late in development during system testing. No programmer changed the code without changing the specification so the specifications and code always matched.

Those planning and specification practices made maintaining software for over 30 years possible without introducing errors when changes were necessary. The common experience in industry, where such extensive planning and specification practices are rare, is that fixing an error in operational software is very like to introduce one or more additional errors.

Continuous improvement: One of the guiding principles of the shuttle software development was that if a mistake was found, you should not just fix the mistake but also fix whatever permitted the mistake in the first place. The process that followed the identification of a software error was: (1) fix the error; (2) identify the root cause of the fault; (3) eliminate the process deficiency that let the fault be introduced and not detected earlier; and (4) analyze the rest of the software for other, similar faults.[30] The goal was to not blame people for mistakes but to blame the process. The development process was a team effort; no one person was ever solely responsible for writing or inspecting the code. Thus there was accountability, but accountability was assigned to the group as a whole.

Configuration management and error databases: Configuration management is critical in a long-lasting project. The PASS software had a sophisticated configuration management system and databases that provided important information to the developers and maintainers. One database contained the history of the code itself, showing every time it was changed, why it was changed, when it was changed, what the purpose of the change was, and what specification documents detailed the change. A second database contained information about the errors that were found in the code. Every error made while writing or changing the software was recorded with information about when the error was discovered, how the error was revealed, who discovered it, what activity was going on when it was discovered (testing, training, or flight), how the error was introduced into the software, how the error managed to slip past the filters set up at every stage to catch errors, how the error was corrected, and whether similar errors might have slipped through the same holes.

Testing and Code Reviews: The complexity and real-time nature of the software meant that exhaustive testing of the software was impossible, despite the enormous effort that went into the test and certification program. There were too many interfaces and too many opportunities for asynchronous input and output. But the enormous amount of testing that went into the Shuttle software certainly contributed greatly to its quality.

Emphasis was placed on early error detection, starting with requirements. Extensive developer and verifier code reviews in a moderated environment were used. It is now widely recognized that human code reviews are a highly effective way to detect errors in software, and they appear to have been very effective in this environment too. Both the developers and the testers used various types of human code reviews. Testing in the SPF and SAIL was conducted under the most flight-like conditions possible (but see the STS-126 communications software error described below).

One interesting experience early in the program convinced NASA that extensive verification and code inspections paid off handsomely. For a year prior to STS-1, the software was frozen and all mandatory changes were made using machine language patches. In parallel, the same changes were made in the STS-2 software. Later it was determined that the quality of the machine language patches for STS-1 was better than the corresponding high-level language (HAL/S) changes in STS-2.[23] This result seemed to defy common beliefs about the danger of patching software. Later the difference was explained by the fact that nervousness about the patching led to the use of much more extensive verification for the patches than for the high-level language changes.

Another lesson about the importance of verification was learned from the cutbacks in staffing initiated by IBM after 1985. At that time, IBM transitioned from long development time to shorter but more frequent operational increments. The result was less time spent on verification and the introduction of a significant number of software errors that were discovered in flight, including three that affected mission objectives and some Severity 1 errors.[23,31]

Learning from these experiences (and others), NASA and its contractors implemented more extensive verification and code inspections on all changes starting with STS-5.

There was some controversy, however, about the use of independent verification and validation. Before the Challenger accident, all software testing and verification was done by IBM, albeit by a group separate from the developers. Early in 1988, as part of the response to the accident, the House Committee on Science, Space, and Technology expressed concern about the lack of independent oversight of the Shuttle software development.[32] A National Research Council committee later echoed the concern and called for Independent Verification and Validation (IV&V).[33] NASA grudgingly started to create an IV&V process. In 1990, the House committee asked the GAO to determine how NASA was progressing in improving independent oversight of the Shuttle software development. The GAO concluded that NASA was dragging its feet in implementing the IV&V program they had reluctantly established.[7] NASA then asked another NRC study committee to weigh in on the controversy, hoping that committee would agree with them that IV&V was not needed. Instead, the second NRC committee, after looking at the results that IV&V had attained during its short existence, recommended that it be continued.[34] After that, NASA gave up fighting it.

Software Development Culture: A final important contributor to the software quality was the culture of the software development organizations. There was a strong sense of camaraderie and a feeling that what they were doing was important. Many of the software developers worked on the project for a long time, sometimes their whole career. They knew the astronauts, many of whom were their personal friends and neighbors. These factors led to a culture that was quality focused and believed in zero defects.

Smith and Cusamano note that "these were not the hot-shot, up-all-night coders often thought of as the Silicon Valley types."[35]. The Shuttle software development job entailed regular 8 am to 5 pm hours, where late nights were an exception. The atmosphere and the people were very professional and of the highest caliber. Words that have been used to describe them include businesslike, orderly, detail-oriented, and methodical. Smith and Cusamano note that they produced "grownup software and the way they do it is by being grown-ups."[35]

The culture was intolerant of "ego-driven hotshots": "In the Shuttle's culture, there are no superstar programmers. The whole approach to developing software is intentionally designed not to rely on any particular person."[35] The cowboy culture that flourishes in some software development companies today was discouraged. The culture was also intolerant of creativity with respect to individual coding styles. People were encouraged to channel their creativity into improving the process, not violating strict coding standards.[23] In the few occasions when the standards were violated, such as the error manifested in STS-126 (see below), they learned the fallacy of waiving standards for small short-term savings in implementation time, code space, or computer performance.

Unlike the current software development world, there were many women involved in Shuttle software development, many of them senior managers or senior technical staff. It has been suggested that the stability and professionalism may have been particularly appealing to women.[35]

The importance of culture and morale on the software development process has been highlighted by observers who have noted that during periods of low morale, such as the period in the early 1990s when the PASS development organization went through several changes in ownership and management, personnel were distracted and several serious errors were introduced. During times of higher morale and steady culture, errors were reduced.[23]

Gaps in the Process

The software development process was evolved and improved over time, but gaps still existed that need to be considered in future projects. The second National Research Council study committee, created to provide guidance on whether IV&V was necessary, at the same time examined the Shuttle software process in depth, as well as many of the software errors that had been found, and it made some suggestions for improvement.[34] Three primary limitations were identified. One was that the verification and validation inspections by developers did not pay enough attention to off-nominal cases. A study

sponsored by NASA had determined that problems associated with rare conditions were the leading cause of software errors found during the late testing stage.[36] The NRC committee recommended that verification activities by the development contractors include more off-nominal scenarios, beyond loop termination and abort control sequence actions and that they also include more detailed coverage analysis.

A second deficiency the NRC committee identified was a lack of system safety focus by the software developers and limited interactions with system safety engineering. System level hazards were not traced to the software requirements, components or functions. The committee found several instances where potentially hazardous software issues were signed off by a flight software manager and not reported to the responsible people or boards.

A final identified weakness related to system engineering. The NRC committee studying Shuttle safety after the Challenger accident had recommended that NASA implement better system engineering analysis:

> "A top-down integrated systems engineering analysis, including a system-safety analysis, that views the sum of the STS elements as a single system, should be performed to identify any gaps that may exist among the various bottom-up analyses centered at the subsystem and element levels."[33]

The IV&V contractor (Intermetrics and later Ares) added after this report was, in the absence of any other group, doing this system engineering task for the software. The second NRC committee concluded that the most important benefits of the IV&V process forced on NASA and the contractors were in system engineering. By the time of the second NRC committee report, the IV&V contractor had found four Severity 1 problems in the interaction between the PASS and the BFS. One of these could have caused the shutdown of all the Shuttle's main engines and the other three involved errors that could have caused the loss of the orbiter and the crew if the backup software had been needed during an ascent abort maneuver. The need for better systems engineering and system safety was echoed by the second NRC committee and hopefully is a lesson NASA will learn for the future.

Learning from Errors

While some have misleadingly claimed that the process used to produce the Shuttle software led to perfect or bug-free software, this was not in fact the case. Errors occurred in flight or were found in other ways in software that had flown. Some of these errors were Severity 1 errors (potential for losing the Shuttle). During the standdown after the Challenger accident, eight PASS Severity 1 errors were discovered in addition to two found in 1985. In total, during the first ten years of Shuttle flights, 16 Severity 1 errors were found in released PASS software, eight of which remained in code used in flight. An additional 12 errors of Severity 2, 3, or 4 occurred during flight in this same period. None of these threatened the crew, but three threatened the mission, and the other nine were worked around.[34] In addition, the Shuttle was flown with known software errors: for example, there were 50 waivers written against the PASS on STS-52, all of which had been in place since STS-47. Three of the waivers covered severity 1N errors. These errors should not detract from the excellent processes used for the Shuttle software development; they simply attest to the fact that developing real-time software is extremely difficult.

IBM and NASA were aware that effort expended on quality at the early part of a project would be much cheaper and simpler that trying to put quality in toward the end. They tried to do much more at the beginning of the Shuttle software development than in previous efforts, as had been recommended by Mueller's Apollo software task force, but it still was not enough to ensure perfection. Tomayko quotes one IBM software manager explaining that "we didn't do it up front enough", the "it" being thinking through the program logic and verification schemes.[3]

Obviously none of the software errors led to the loss of the Shuttle although some almost led to the loss of expensive hardware and some did lead to not fully achieving mission objectives, at least using the software. Because the orbital functions of the Shuttle software were not fully autonomous, astronauts or Mission Control could usually step in and manually recover from the few software problems that did occur. For example, a loss was narrowly averted during the maiden flight of Endeavor (STS-49) on May

12, 1992 as the crew attempted to rendezvous with and repair an Intelsat satellite.[34] The software routine used to calculate rendezvous firings, called the Lambert Targeting Routine, did not converge on a solution due to a mismatch between the precision of the state vector variables, which describe the position and velocity of the Shuttle, and the limits used to bound the calculation. The state vector variables were double precision while the limit variables were single precision. The satellite rescue mission was nearly aborted, but a workaround was found that involved relaying an appropriate state vector value from the ground.

Shortly before STS-2, during crew training, an error was discovered when all three Space Shuttle main engines were simulated to have failed in a training scenario. The error caused the PASS to stop communicating with all displays and the crew engaged the BFS. An investigation concluded the error was related to the specific timing of the three SSME failures in relation to the sequencing to connect the Reaction Control System (RCS) jets to an alternate fuel path. Consistent with the continuous learning and improvement process used, a new analysis technique (called Multi-Pass Analysis) was introduced to prevent the same type of problem in the future.[23]

As another example, during the third attempt to launch Discovery on August 29, 1984 (STS-41D), a hardware problem was detected in the Shuttle's main engine number 3 at T–6 seconds before launch, and the launch was aborted. However, during verification testing of the next operational software increment (before the next attempt to launch Discovery), an error was discovered in the master event controller software related to solid rocket booster fire commands that could have resulted in the loss of the Shuttle due to inability to separate the solid rocket boosters and external tank. That discrepancy was also in the software for the original Discovery launch that was scrubbed due to the engine hardware problem. Additional analysis determined that the BFS would not have been a help because it would not have been able to separate the solid rocket boosters either if the condition occurred. The occurrence of the conditions that would have triggered the PASS error were calculated to be one in six launches. A software patch was created to fix the software error and assure all three booster fire commands were issued in the proper time interval. The problem was later traced to the requirements stage of the software development process and additional testing and analysis introduced into the process to avoid a repetition.

Another timing problem that could have resulted in failure to separate the external tank was discovered right before the launch of STS-41D in August 1984. In the 48 hours before the launch, IBM created, tested, and delivered a 12-word code patch to ensure sufficient delay between the PASS computed commands and the output of those commands to the hardware.

With the extensive testing that continued throughout the Shuttle program, the number of errors found in the software did decrease over the life of the Shuttle, largely because of the sophisticated continuous improvement and learning process used in the software development process, to the point where almost no errors were found in the software for the last few Shuttle flights,[23] although there also were fewer changes made to the software during that period.

There was, however, a potentially serious software error that occurred in April 2009, just two years before the Shuttle's retirement. The error manifested itself in flight STS-126 a few minutes after Endeavor reached orbit.[37] Mission Control noticed that the Shuttle did not automatically transfer two communication processes from launch to orbit configuration mode. Mission Control could not fix the problem during the flight, so they manually operated necessary transfers for the remainder of the flight. The pathway for this bug had been introduced originally in a change made in 1989 with a warning inserted in the code about the potential for that change to lead to misalignment of code in the COMPOOL. As more changes were made, the warning got moved to a place where it was unlikely to be seen by programmers changing the code. The original change violated the programming standards, but that standard was unclear and nobody checked that it was enforced in that case. Avoiding the specific error that was made was considered "good practice," but it was not formally documented and there were no items in the review checklist to detect it. The SPF did not identify the problem either—testers would have needed to take extra steps to detect it. The SAIL could have tested the communication switch but it was not identified as an essential test for that launch. Testing at the SAIL did uncover what hindsight indicated were clear problems in the communication handover, but the test team misinterpreted what happened

during test—they thought it was an artifact of lab setup issues—and no error reports were filed. While *test as you fly and fly as you test* is a standard rule in spacecraft engineering, the difficulty of achieving this goal is demonstrated by this specific escape even given the enormous amounts of money that went into testing in the SAIL lab.

A final example is a software error that was detected during analysis of post-flight data from STS-79. This error resulted from a "process escape." Hickey, et.al note that most of these errors can be traced to periods of decreasing morale among the IBM programming staff or pressures leading to decreased testing and not following the rigorous procedures that had been developed over the years.[23]

In hindsight, it is easy to see that the challenges NASA and its contractors faced in terms of memory limitations, changing requirements, communication, and software and computer hardware quality and reliability, were enormous, particularly given the state of technology at the time. Luckily, the Shuttle software developers did not have this hindsight when they started and went forward with confidence they could succeed, which they did spectacularly, in the manner of the U.S. manned space program in general.

Lessons Learned, Conclusions, and Analogies

There can always be differing explanations for success (or failure) and varying emphasis placed on the relative importance of the factors involved. Personal biases and experiences are difficult to remove from such an evaluation. But most observers agree that the process and the culture were important factors in the success of the Shuttle software as well as the strong oversight, involvement, and control by NASA.

1. Oversight and Learning from the Past: NASA learned important lessons from previous spacecraft projects about the difficulty and care that need to go into the development of the software, including that software documentation is critical, verification must be thorough and cannot be rushed to save time, requirements must be clearly defined and carefully managed before coding begins and as changes are needed, software needs the same type of disciplined and rigorous processes used in other engineering disciplines, and quality must be built in from the beginning. By maintaining direct control of the Shuttle software rather than ceding control to the hardware contractor and, in fact, constructing their own software development "factory" (the SPF), NASA ensured that the highest standards and processes available at the time were used and that every change to human-rated flight software during the long life of the Shuttle was implemented with the same professional attention to detail.

2. Development Process: The development process was a major factor in the software success. Especially important was careful planning before any code was written, including detailed requirements specification, continuous learning and process improvement, a disciplined top-down structured development approach, extensive record keeping and documentation, extensive and realistic testing and code reviews, detailed standards, and so on.

3. The Software Development Culture: Culture matters. The challenging work, cooperative environment, and enjoyable working conditions encouraged people to stay with the PASS project. As those experts passed on their knowledge, they established a culture of quality and cooperation that persisted throughout the program and the decades of Shuttle operations and software maintenance activities.

With the increasing complexity of the missions anticipated for the future and the increasing role of software in achieving them, another lesson that can be learned is that we will need better system engineering, including system safety engineering. NASA maintained control over the system engineering and safety engineering processes in the Shuttle and employed the best technology in these areas at the time. The two Shuttle losses are reminders that safety involves more than simply technical prowess,

however, and that management can play an important role in accidents and must be part of the system safety considerations. In addition, our system and safety engineering techniques need to be upgraded to include the central role that software plays in our complex spacecraft systems. Unfortunately, the traditional hazard analysis techniques used in the Shuttle do not work very well for software-intensive systems.[27]

Beyond these lessons learned**,** some general conclusions and analogies can be drawn from the Shuttle experience to provide guidance for the future. One is that high quality software is possible but requires a desire to do so and an investment of time and resources. Software quality is often given lip service in other industries, where often speed and cost are the major factors considered, quality simply needs to be "good enough," and frequent corrective updates are the norm.

Some have suggested that the unique factors that separated the Shuttle from other software development projects are that there is one dedicated customer, a limited problem domain, and a situation where cost was important but less so than quality.[1] But even large government projects with a single government customer and large budgets have seen spectacular failures in the recent past such as the new IRS software,[38] several attempted upgrades to the Air Traffic Control system,[39] a new FBI system,[40] and even an airport luggage system.[41] That latter baggage system cost $186,000,000 for construction alone and never worked correctly. The other cited projects involved, for the most part, at least an order of magnitude higher costs than the baggage system and met with not much more success. In all of these cases, enormous amounts of money were spent with little to show for them. They had the advantage of newer software engineering techniques, so what was the significant difference?

One difference is that NASA maintained firm control over and deep involvement in the development of the Shuttle software. They used their experience and lessons learned from the past to improve their practices. With the current push to privatize the development of space vehicles, will the lesser oversight and control lead to more problems in the future? How much control will and should NASA exercise? Who will be responsible for system engineering and system safety?

In addition, software engineering is moving in the opposite direction from the process used for the Shuttle software development, with requirements and careful pre-planning relegated to a less important position than starting to code. Strangely, in many cases, a requirements specification is seen as something that is generated after the software design is complete or at least after coding has started. Many of these new software engineering approaches are being used by the firms designing new spacecraft today. Why has it been so difficult for software engineering to adopt the disciplined practices of the other engineering fields? There are still many software development projects that depend on cowboy programmers and "heroism" and less than professional engineering environments. How will NASA ensure that the private companies building manned spacecraft instill a successful culture and professional environment in their software development groups? Ironically, many of the factors that led to success in the Shuttle software were related to limitations of computer hardware in that era, including limitations in memory that prevented today's common "requirements creep" and uncontrolled growth in functionality as well as requiring careful functional decomposition of the system requirements. Without the physical limitations that impose discipline on the development process, how can we impose discipline on ourselves and our projects?

The overarching question is how will we ensure that the hard learned lessons of past manned space projects are conveyed to those designing future systems and that we are not, in the words of Santayana, condemned to repeat the same mistakes.

Notes

[1] Charles Fishman, "They Write the Right Stuff," *Fast Company*, December 1996

[2] Tarandeep Singh, "Why NASA Space Shuttle Software Never Crashes: Bug-free NASA Shuttles," *The Geeknizer*, July 17, 2011

[3]James Tomayko, "Computers in Spaceflight: The NASA Experience," NASA Contractor Report CR-182505, 1988.

[4] http://www.ibiblio.org/apollo/Gemini.html

[5] Arthus L. Slotkin, *Doing the Impossible: George E. Mueller and the Management of NASA's Human Spaceflight Program*, Springer-Praxis, Chichester, U.K., 2012.

[6] Software "discrepancy" is the common NASA terminology for a software bug or error.

[7] GAO, "NASA Should Implement Independent Oversight of Software Development," GAO/IMTEC-91-20, 1991
[8] In 1992, an NRC committee studying the NASA Shuttle software process was told that the yearly cost for the flight software development contractors was approximately $60 million. Operation of the Shuttle Avionics Integration Lab (SAIL), which is used to test the flight software, required approximately $24 million per year. This total does not include the costs for the Space Shuttle Main Engine software and other support contractors. See An Assessment of Space Shuttle Flight Software Development Processes, Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Processes, Aeronautics and Space Engineering Board, National Research Council, 1993

[9] User notes are provided to the Astronauts to help them work around known software limitations and errors. Waivers are decisions on the part of the Shuttle program to recognize a condition, such as a known software error, as an acceptable risk. Thus, a condition that receives a waiver is set aside, sometimes fixed at a later date when time and resources were available, but were not considered sufficient cause to hold up a flight. The excessive use of waivers got attention after the Columbia loss.

[10] R.M. Mattox and J.B. White, "Space Shuttle Main Engine Controller," Report NASA-TP-1932 M-360, NASA, Nov. 1, 1981.

[11] Gene D. Carlow, Architecture of the Space Shuttle Primary Avionics Software System, *Communications of the ACM*, Vol. 27, No. 9, Sept. 1984, pp. 926–936

[12] A Titan carrying a Milstar satellite was lost in 1999 when a typo in a load tape that had inadvertently never been tested led to an incorrect attitude value being used by the flight control software. See J.G. Pavlovich, *Formal Report of the Investigation of the 30 April 1999 Titan IV B/Centaur/TC-14/Milstar* , U.S. Air Force.

[13] A time-sliced system allocates predefined periods of time for the execution of each task and then suspends tasks unfinished in that time period and moves on to the next time slice.

[14] J.R. Garman, "Software Production Facility: Management Summary Concepts and Schedule Status," NASA Data Systems and Analysis Directorate, Spacecraft Software Division, February 10, 1981, p. 12

[15] Compare this with the approximately 5,000,000 lines of code on commercial aircraft today and 15-20 million lines in military aircraft. The ISS has 2.4 million lines of code on board.

[16] In the preface to the language specification document, the name "HAL" is described as being in honor of Draper Labs Dr. J. Halcombe Laning.

[17] *HAL/S Language Specification*, United Space Alliance, 2005

[18] F.H. Martin, *HAL/S: The Avionics Programming System for the Shuttle*, AIAA, 315, 1977.

[19] ALT (Approach and Landing Tests) were a series of taxi and flight trials of the prototype Space Shuttle *Enterprise* conducted in 1977 to test the vehicle's flight characteristics both on its own and when mated to the Shuttle Carrier Aircraft, prior to the operational debut of the shuttle system. In January 1977, *Enterprise* was taken by road from the Rockwell plant at Palmdale, California, to the Dryden Flight Research Center at Edwards Air Force Base to begin the flight test phase of the program, which had been christened by NASA as the *Approach and Landing Tests* (ALT).

[20] Frederick Brooks, *The Mythical Man Month*, Addison-Wesley, 1973.

[21] William A Madden and Kyle Y. Rone, "Design, Development, Integration: Space Shuttle Primary Flight Software System," *Communications of the ACM*, Vol. 27, No. 9, Sept. 1984, pp. 914–925.

[22] For example, impacts to the Mission Control Center, the Launch Processing System, procedures and training, or flight design requirements.

[23] J. Christopher. Hickey, James B. Loveall, James K. Orr, and Andres L. Klausman, "The Legacy of Space Shuttle Flight Software," *AIAA Space 2011 Conference*, Sept. 2-29, 2011, Long Beach California, 2011.

[24] John C. Knight, J.C. and Nancy G. Leveson, "Experimental evaluation of the assumption of independence in multiversion software," *IEEE Trans.Software Eng.* **SE-12**(1):96-109, 1986.

[25] John C. Knight and Nancy G. Leveson, "A Reply to the Criticisms of the Knight and Leveson Experiment," *ACM Software Engineering Notes*, January 1990.

[26] Nancy Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Publishing Companry, 1996

[27] Nancy Leveson, *Engineering a Safer World*, MIT Press, 2012.

[28] John R. Garman, "The "Bug" Heard 'Round the World," *Software Engineering Notes*, ACM, October 1981, pp. 3–10

[29] Stubs are used in place of modules that have not yet been developed. They act as procedures and return default values so the software can execute before all the procedures are written and interfaces checked.

[30] Keller, T.W. 1993, "Maintenance Process Metrics for Space Shuttle Flight Software" *Forum on Statistical Methods in Software Engineering*, National Research Council, Washington D.C., October 11-12, 1993.

[31] Shuttle flight software errors are categorized by the severity of their potential consequences without regard to the likelihood of their occurrence. Severity 1 errors are defined as errors that could produce a loss of the Space Shuttle or its crew. Severity 2 errors can affect the Shuttle's ability to complete its mission objectives, while severity 3 errors affect procedures for which alternatives, or workarounds, exist. Severity 4 and 5 errors consist of very minor coding or documentation errors. In addition, there is a class of severity 1 errors, called severity 1N, which, while potentially life-threatening, involve operations that are precluded by established procedures, are deemed to be beyond the physical limitations of Shuttle systems, or are outside system failure protection levels.

[32] Letter to Administrator, NASA, from Chairman, House Committee on Science, Space, and Technology, March 31, 1988.

[33] *Post-Challenger Evaluation of Space Shuttle Risk Assessment and Management*, Aeronautics and Space Engineering Board, National Research Council, January 1988.

[34] *An Assessment of Space Shuttle Flight Software Development Processes*, Committee for Review of Oversight Mechanisms for Space Shuttle Flight Software Processes, Aeronautics and Space Engineering Board, National Research Council, 1993.

[35] Stan A. Smith and Michael A Cusumano, Beyond the Software Factory: A Comparison of Classic and PC Software Developers, Massachusetts Institute of Technology, Sloan School WP#3607=93\BPS, September 1, 1993.

[36] H. Hecht, "Investigation of Shuttle Software Errors," SoHar Incorporated, study prepared for Polytechnic University, Brooklyn, New York, and the Langley Research Center, Hampton, Virginia, under NASA Grant NAG1-1272, April 1992

[37] NASA, "Shuttle System Failure Case Studies: STS-126, NASA Safety Center Special Study, NASA, April 2009.

[38] Anne Broache, "IRS Trudges on with Aging Computers," CNET News, April 12, 2007 , http://news.cnet.com/2100-1028_3-6175657.html

[39] Mark Lewyn, "Flying in Place: The FAA's Air Control Fiasco," *Business Week*, April 26, 1993, pp. 87, 90

[40] Dan Eggan and Griff Witte, "The FBI's Upgrade That Wasn't," *The Washington Post*, August 18, 2006, http://www.washingtonpost.com/wp-dyn/content/article/2006/08/17/AR2006081701485.html

[41] Kirk Johnson, "Denver Airport Saw the Future. It didn't work," *New York Times*, August 27, 2005