- Chapter Two -
- Computers On Board The Apollo Spacecraft -

The Apollo guidance computer: Software

[40] Development of the on-board software for the Apollo program was an important exercise both for NASA and for the discipline of software engineering. NASA acquired considerable experience in managing a large, real-time software project that would directly influence the development of the Shuttle on-board software. Software engineering as a specific branch of computer science emerged as a result of experiences with large-size military, civilian, and spaceborne systems. As one of those systems, the Apollo software effort helped [41] provide examples both of failure and success that could be incorporated into the methodology of software engineering.

In the Apollo program, as well as other space programs with multiple missions, system software and some subordinate computer programs are only written once, with some modifications to help integrate new software. However, each mission generates new operational requirements for software, necessitating a design that allows for change. Since 1968, when designers first used the term software engineering, consciousness of a software life cycle that includes an extended operational maintenance period has been an integral part of proper software development.

Even during the early 1960s, the cycle of requirements definition, design, coding, testing, and maintenance was followed, if not fully appreciated, by software developers. A Bellcomm report prepared for the Apollo program and dated November 30, 1964 could serve as an excellent introduction to the concept today[71]. The important difference from present practice was the report's recommendation that modules of code be limited to 200 to 300 lines, about five times larger than current suggestions. The main point of the report (and the thrust of software engineering) was that software can be treated the same way as hardware, and the same engineering principles can apply. However, NASA was more used to hardware development than to large-scale software and, thus, initially failed adequately to control the software development. MIT, which concentrated on the overall guidance system, similarly treated software as a secondary occupation[72]. This was so even though MIT manager A.L. Hopkins had written early in the program that "upon its execution rests the efficiency and flexibility of the Apollo Guidance and Navigation System"[73]. Combined with NASA's inexperience, MIT's non-engineering approach to software caused serious development problems that were overcome only with great effort and expense. In the end NASA and MIT produced quality software, primarily because of the small-group nature of development at MIT and the overall dedication shown by nearly everyone associated with the Apollo program[74].

Managing the Apollo Software Development Cycle

One purpose of defining the stages in the software development cycle and of providing documentation at each step is to help control the production of software. Programmers have been known to inadvertently modify a design while trying to overcome a particular coding difficulty, thus making it impossible to fulfill the specification. Eliminating communication problems and preventing variations from the designed solution are among the goals of software engineering. In [42] the Apollo program, with an outside organization developing the software, NASA had to provide for quality control of the product. One method was a set of standing committees; the other was the acceptance cycle.

Three boards contributed directly to the control of the Apollo software and hardware development. The Apollo Spacecraft Configuration Control Board monitored and evaluated changes requested in the design and construction of the spacecraft itself, including the guidance and control system, of which the computer was a part. The Procedures Change Control Board, chaired by Chief Astronaut Donald K. Slayton, inspected items that would affect the design of the user interfaces. Most important was the Software Configuration Control Board, established in 1967 in response to continuing problems and chaired for a long period by Christopher Kraft. It controlled the modifications made to the on-board software[75]. All changes in the existing specification had to be routed through this board for resolution. NASA's Stan Mann commented that MIT "could not change a single bit without permission"[76].

NASA also developed a specific set of review points that paralleled the software development cycle. The Critical Design Review (CDR) resulted in acceptance of specifications and requirements for a given mission and placed them under configuration control. It followed the preparation of the requirements definition, guidance equation development, and engineering simulations of the equations. Next came a First Article Configuration Inspection (FACI). Following the coding and testing of programs and the production of a validation plan, it marked the completion of the development stage and placed the software code under configuration control. After testing was completed, the Customer Acceptance Readiness Review (CARR) certified that the validation process resulted in correct software. After the CARR, the code would be released for core rope manufacture. Finally the Flight Readiness Review (FRR) was the last step in clearing the software for flight[77]. The acceptance process was mandatory for each mission, providing for consistent evaluation of the software and ensuring reliability. The unique characteristic of ICs of the Apollo software appeared at each stage of the software life cycle.

Requirements Definition

Defining requirements is the single most difficult part of the software development cycle. The specification is the customer's statement of what the software product is to do. Improperly prepared or poorly defined requirements mean that the resulting software will likely be incomplete and unusable. Depending on the type of project, the customer may have little or a lot to do with the preparation of the [43] specification. In most cases, a team from the software developers works with the customer.

MIT worked closely with NASA in preparing the Guidance and Navigation System Operations Plan (GSOP), which served as the requirements document for each mission. NASA's Mission Planning and Analysis Division at the Manned Spacecraft Center provided detailed guidance requirements right down to the equation level[78]. Often these requirements were in the form of flow charts to show detailed logic[79]. The division fashioned these requirements into a controlled document that contained specific mission requirements, preliminary mission profile, preliminary reference trajectory, and operational requirements for spacecraft guidance and navigation. NASA planned to review the GSOP at launch minus 18 months, 16 months, 14 months and then to baseline or "freeze" it at 13.5 months before launch. The actual programs were to be finished at launch minus 10.5 months and tested until 8 months ahead, when they were released to the manufacturer, with tapes also kept at MIT and sent to Houston, North American (CM manufacturer), and Grumman (LEM manufacturer) for use in simulations. At launch minus 4 months the core ropes were to be completed and used throughout the mission[80].

In software engineering practice today, the specification document is followed by a design document, from which the coding is done. Theoretically, the two together would enable any competent programmer to code the program. The GSOPs contained characteristics of both a specification and design document. But, as one of the designers of the Apollo and Shuttle software has said, "I don't think I could give you the requirements for Apollo and have you build the flight software"[81]. In fact, the plans varied both in what they included and in the level of detail requirements. This variety gave MIT considerable latitude when actually developing the flight software, thus reducing the chance that it would be easily verified and validated.

Coding: Contents of the Apollo Software

By 1963, designers determined that the Apollo computer software would have a long list of capabilities, including acting as backup to the Saturn booster, controlling aborts, targeting, all navigation and flight control tasks, attitude determination and control, digital autopilot tasks, and eventually all maneuvers involving velocity changes[82]. Programs for these tasks had to fit in the memories of two small computers, one in the CM and one in the LEM. Designers developed the programs using a Honeywell 1800 computer and later an IBM 36O, but never with the actual flight hardware. The development computers generated binary object code and a listing[83]. The tape [44] containing the object code would be tested and eventually released for core rope manufacture. The listing served as documentation of the code[84].

Operating System Architecture

The AGC was a priority-interrupt system capable of handling several jobs at one time. This type of system is quite different from a "round-robin executive." In the latter programs have a fixed amount of time in which to run before being suspended while the computer moves on to the remaining pending jobs, thus giving each job the same amount of attention. A priority-interrupt system is always executing the one job with the highest priority; it then moves on to others of equal or lower priority in its queue.

The Apollo control programs included two related to job scheduling: the Executive and the Waitlist. The Executive could handle up to seven jobs at once while the Waitlist had a limit of nine short tasks[85]. Waitlist tasks had execution times of 4 milliseconds or less. If a task ran longer than that, it would be promoted by the Waitlist to "job" status and moved to the Executive's queue[86]. The Executive checked every 20 milliseconds for jobs or tasks with higher priorities than the current ones[87]. It also managed the DSKY displays88. If the Executive checked the priority list and found no other jobs waiting, it executed a program called DUMMY JOB continuously until another job came into the queue[89].

The Executive had other duties as part of controlling jobs. One solution to the tight memory in the AGC was the concept of time-sharing the erasable memory[90]. No job had permanent claim to any registers in the erasable store. When a job was being executed, the Executive would assign it a "coreset" of 12 erasable memory locations. AISO7 when interpretive jobs were being ran (the Interpreter is explained below), an additional 43 cells were allocated for vector accumulation (VAC). The final lunar landing programs had eight coresets in the LEM computer and just seven in the CM. Both had five VACs[91]. Moreover, memory locations were given multiple assignments where it was assured that the owning processes would never execute at the same time. This approach caused innumerable problems in testing as software evolved and memory conflicts were created due to the changes.

Programming the AGC

[45] One can program a computer on several levels. Machine code, the actual binary language of the computer itself, is one method of specifying instructions. However, it is tedious to write and prone to error. Assembly language, which uses mnemonics for instructions (e.g., ADD in place of a 3-bit operation code) and, depending on its sophistication, handles addressing, is at a higher level. Most programmers in the early 1960s were quite familiar with assembly languages, but such programs suffered from the need to put too much responsibility in the hands of the programmer. For Apollo, MIT developed a special higher order language that translated programs into a series of subroutine linkages, which were interpreted at execution time. This was slower than a comparable assembly language program, but the language required less storage to do the same job[92]. The average instruction required two machine cycle&emdash;about 24 milliseconds&emdash;to execute[93].

The interpreter got a starting location in memory, retrieved the data in that location, and interpreted the data as though it were an instruction[94]. Instead of having only the 11 instructions available in assembler, up to 128 pseudoinstructions were defined[95]. The larger number of instructions in the interpreter meant that equations did not have to be broken down excessively[96]. This increased the speed and accuracy of the coding.

The MIT staff gave the resulting computer programs a variety of imaginative names. Many, such as SUNDISK, SUNBURST, and SUNDIAL, related to the sun because Apollo was the god of the sun in the classical period. But the two major lunar flight programs were called COLOSSUS and LUMINARY. The former was chosen because it began with "C" like the CM, and the latter because it began with "L" like the LEM[97]. Correspondence between NASA and MIT often shortened these program names and appended numbers. For example, SOLRUM55 was the 55th revision of SOLARIUM for the AS501 and 502 missions. BURST116 was the 116th revision of SUNBURST[98]. Although these programs had many similarities, COLOSSUS and LUMINARY were the only ones capable of navigating a flight to the moon. On August 9, 1968, planners decided to put the first released version of COLOSSUS on Apollo 8, which made the first circumlunar flight possible on that mission[99].

Handling Restarts

One of the most significant differences between batch-type [46] computer systems and real-time systems is the fact that in the latter, an abnormal termination of a program is not acceptable. If a ground-based, non-real-time computer system suffers a software failure ("goes down") due to overloads or mismanagement of resources, it can usually be brought up again without serious damage to the users. However, a failure in a real-time system such as that in an aircraft may result in loss of life. Such systems are backed up in many ways, but considerable emphasis is still placed on making them failure proof from the start. Obviously, the AGC had to be able to recover from software failures. A worstcase example would be a failure of the computer during an engine burn. The system had to have a method of staying "up" at all times.

The solution was to provide for restarts in case of software failures. Such restarts could be caused by a number of conditions: voltage failures, clock failure, a "rupt lock" in which the system got stuck in interrupt mode, or a signal from the NIGHT WATCHMAN program, which checked to see if the NEWJOB register had not been tested by the EXECUTIVE, indicating that the operating system was hung up in some way[100].

An Apollo restart transferred control to a specified address, where a program would begin that consulted phase tables to see which jobs to schedule first. These jobs would then be directed to pick up from the last restart point. The restart point addresses were kept in a restart table. Programmers had to ensure that the restart table entries and phase table entries were kept up to date by the software as it executed[101]. The restart program also cleared all output channels, such as control jet commands, warning lights, and engine on and off commands, so that nothing dangerous would take place outside of computer control[102] .

A software failure causing restarts occurred during the Apollo 11 lunar landing. The software was designed to give counter increment requests priority over instructions[103]. This meant that if some item of hardware needed to increment the count in a memory register, its request to do so would cause the operating system to interrupt current jobs, process the request, and then pick up the suspended routines. It had been projected that if 85,O00 increments arrived in a second, the effect would be to completely stop all other work in the system[104]**.** Even a smaller number of requests would slow the software down to the point at which a restart might occur. During the descent of Apollo 11 to the moon, the rendezvous radar made so many increment requests that about 15% of the computer systems resources were tied up in responding[105]. The time spent handling the interrupts meant that the interrupted jobs did not have enough computer time to complete before they were scheduled to begin again. This situation caused restarts to occur, three of which happened in a 40-second period while program P64 of LUMINARY ran during descent[106]. The restarts [47] caused a series of warnings to be displayed both in the spacecraft and in Mission Control. Steven G. Bales and John R. Garman, monitoring the computer from Mission Control, recognized the origin of the problem. After consultation, Bales, reporting to the Flight Director, called the system GO for landing[107]. They were right, and the restart software successfully handled the situation. The solution to this particular problem was to correct a switch position on the rendezvous radar which, through an arcane series of circuitry, had caused the analog-to-digital conversion circuitry to race up and down[108]. This incident proved the need for and effectiveness of built-in software recovery for unknown or unanticipated error conditions in flight software&emdash;a philosophy that has appeared deeply embedded in all NASA manned spaceflight software since then.

Verification and Validation

There could be no true certification of the Apollo software because it was impossible to simulate the actual conditions under which the software was to operate, such as zero-G. The need for reliability motivated an extensive testing program consisting of simulations that could be accomplished before flight. Three simulation systems were available for verification purposes: all-digital, hybrid, and system test labs. All-digital simulations were performed on the Honeywell 1 800s and IBM 360s used for software development. Their execution rate was 10% of real time[109]. Technicians did hybrid simulations in a lab that contained an actual AGC with a core rope simulator (as core rope would not be manufactured until after verification of the program) and an actual DSKY. Additionally, an attached Beckman analog computer and various interfaces simulated spacecraft responses to computer commands[110]. Further ad hoc verification took place in the mission trainers located in Houston and at Cape Canaveral, which would run the released programs in their interpretive simulators.

The simulations followed individual unit tests and integrated tests of portions of the software. At first, MIT left these tests to the programmers to be done on an informal basis. It was very difficult at first to get the Instrumentation Laboratory to supply test plans to NASA[111]. The need for formal validation rose with the size of the software. Programs of 2,OOO instructions took between 50 and 100 test runs to be fully debugged, and full-size mission loads took from 1,000 to 1,200 runs[112].

NASA exerted some pressure on MIT to be more consistent in testing, and it eventually adopted a four-level test structure based largely on the verification of the Gemini Mission Control Center developed by IBM in 1964[113]. This is important because formal [48] release of the program for rope manufacture was dependent on the digital simulations only. Raytheon performed the hybrid and system tests after they had the release tape in hand[114]. At that time, MIT would have released an AGC Program Verification Document to NASA. Aside from help from IBM, NASA also had TRW participate in developing test plans. Having an outside group do some work on verification is a sound software engineering principle, as it is less likely to have a vested interest in seeing the software quickly succeed, and it helps prevent generic errors.

Apollo Software Development Problems

Real-time flight software development on this scale was a new experience for both NASA and the MIT Instrumentation Laboratory. Memory limitations affected the software so that some features and functions had to be abandoned, whereas tricky programming techniques saved others. Quality of the initial code was sometimes poor, so verification took longer and was more expensive. Despite valiant validation efforts, software bugs remained in released programs, forcing adjustments by users. Several times, NASA administrators put pressure on MIT, to reduce software complexity because there were real doubts about MIT’s ability to deliver reliable software on time. Apparently, few had anticipated that software would become a pacing item for Apollo, nor did they properly anticipate solutions to the problems.

By early 1966, program requirements even exceeded the Block H computer’s memory. A May software status memo stated that not only would the programs for the AS504 mission (earth orbit with a LEM) exceed the memory capacity by 11,800 words but that the delivery date for the simpler AS207/208 programs would be too late for the scheduled launch[115]. Lack of memory and the need for faster throughput resulted in complicating and delaying the program development effort[116]. One of MIT’s top managers explained

> If you are limited in program capacity ... you have to fix. You have to get ingenious, and as soon as you start to get ingenious you get intermeshing programs, programs that depend upon others and utilize other parts of those, and many things are going on simultaneously. So it gets difficult to assign out little task groups to program part of the computer; you have to do it with a very technical team that understands all the interactions on all these things[117].

The development of obscure code caused problems both in understanding the programs and validating them, and this, in turn, caused delays. MIT’s considerable geographic distance from Houston caused [49] additional problems. Thus, NASA’s contract managers had to commute often. Howard W. "Bill" Tindall, newly assigned from the Gemini Project as NASA’s "watchdog" for MIT software, spent 2 or 3 days a week in Boston starting in early 1966[118].

Tindall was well known at the Manned Spacecraft Center due to his legendary "Tindallgrams" &emdash; blunt memos regarding software development for Apollo. One of the first to recognize the importance of software to mission schedules, he wrote on May 31, 1966 that "the computer programs for the Apollo spacecraft will soon become the most pacing item for the Apollo flights[119]. MIT was about to make the standard emergency move when software was in danger of being late: to throw more bodies into the project, a tactic that often backfires. As many as 50 people were to be added to the programming staff, and the amount of interaction between programmers and, thus, the potential for miscommunication increased along with the time necessary to train newcomers. MIT tried to protect the tenure of its permanent staff by using contractors who could be easily released. The hardware effort peaked at 600 workers in June of 1965 and fell off rapidly after that, while software workers steadily increased to 400 by August of 1968. With the completion of the basic version of COLOSSUS and LUMINARY, the number of programmers quickly decreased[120]. This method, although in the long-term interests of the laboratory, caused considerable waste of resources in communication and training.

Tindall’s memo also detailed many of NASA’s efforts to improve MIT’s handling of the software development. Tindall had taken Lynwood Dunseith, then head of the computer systems in Mission Control, and Richard Hanrahan of IBM to MIT to brief the Instrumentation Laboratory on the Program Development Plan used for management of software development in the Real-Time Computing Center associated with Mission Control. The objective was to give MIT some suggestions on measuring progress and detecting problem areas early. One NASA manager pointed out that the Instrumentation

Laboratory was protective of the image of MIT, and one way to control MIT was to threaten its self-esteem[121]. The need to call on IBM for advice was apparently a form of negative motivation. A couple of weeks later, Tindall reported that Edward Copps of MIT was leading the development of a Program Development Plan based on one done by IBM[122]. However, by July he was complaining that MIT was implementing it too slowly[123]. In fact, some aspects of configuration control such as discrepancy reporting (when the software does not match the specification) took over a year for MIT to implement[124].

NASA had to be very careful in approving cuts in the program requirements to achieve some memory savings. Some features were obviously "frosting," and could easily be eliminated; for example, the effects of the oblate nature of the earth, formerly figured into lunar orbit [50] rendezvous but actually minimal enough to be ignored[125]. Also cut were some attitude maneuver computations. They therefore left Reaction Control System (RCS) burns to the "feel" of the pilot, which meant slightly greater fuel expenditure[126]. Overall, the cuts resulted in software that saved money and accelerated development but could not minimize fuel expenditures nor provide the close guidance tolerance that was within the capability of the computer, given more memory[127].

Flight AS-204: A Breaking Point

Despite efforts by both MIT and NASA, by the summer of 1966, flight schedules and problems in development put both organizations in a dangerous position regarding the software. A study of the problems encountered with the software for flight AS-204 which was to be the first manned Apollo mission, best demonstrates the urgency. On June 13, Tindall reported that the AS-204 program undergoing integrated tests had bugs in every module. Some had not been unit tested prior to being integrated[128]. This was a serious breach of software engineering practice. If individual modules are unit tested and proven bug-free, then bugs found in integrated tests are most likely located in the interfaces or calling modules. If unit testing has not been done then bugs could be anywhere in the program load, and it is very difficult to identify the location properly. This vastly increases the time and, thus, the cost of debugging. It causes a much greater slip in schedule than time spent on unit tests. Even worse, Tindall said that the test results would not be formally documented to NASA but that they would be on file if needed.

The AS-204 software schedule problems affected other things. All the crew-requested changes in the programs were rejected because including them would cause even further delays[129]. The AS-501 program and others began to slip because the AS-204 fixes were saturating the Honeywell 1800s used in program development[130]. MIT also added another nine programmers to the team, all from AC Electronic, thus increasing communication and training problems.

The eventual result was that the flight software for the mission was of dubious quality. Tindall predicted such would be the case as early as June 1966, saying that we have every expectation that the flight program we finally must accept will be of less than desirable quality[131]. In other words, it would contain bugs, bugs that would not actually threaten the mission directly but that would have to be worked around either by the crew or by ground control. They found one such bug less than a month before the scheduled February 21, 1967, launch date. Ground computers and the Apollo guidance [51] computer calculated the time for the de-orbit burn that preceded re-entry. Simulations performed during January 1967 and reported on the 23rd indicated that there was a discrepancy between the two calculations of as much as 138 seconds! Since the core rope was already installed in the spacecraft, the only possible fix (besides a delay in the launch time) would be to have the crew ignore the Apollo computer solution. The ground would transmit the Real-Time Computing Center solution, after which an astronaut would have to key the numbers into the Apollo computer[132]. This situation, and other discrepancies, led one NASA engineer to later remark that we were about to fly with flight software that was really suspect [133].

AS-204 did not fly, so that software load was never fully tried. On January 27 1967, during a simulation with the crew in the spacecraft on the pad, a fire destroyed the CM, killed the crew, and delayed the Apollo program for months. The changes in managing software development put into effect by NASA and MIT during 1966 had not had enough time to take effect before the fire. In the ensuing period, with manned launches on indefinite delay, MIT was under the direction of the NASA team led by Tindall and was able to catch up on its work and take steps to make the software more reliable. NASA and MIT split the effort among three programs: CM earth orbit, CM lunar orbit, and lunar module lunar landing (LM earth orbit was dropped)[134]. By October l7, 1967, the SUNDISK earth orbit program was complete, verified, and ready for core rope manufacture, a year before the first manned flight[135]. The time gained by the delay caused by the fire allowed for significant improvements in the Apollo software. Tindall observed at the time, It is becoming evident that we are entering a new epoch regarding development of spacecraft computer programs. No longer would programs be declared complete in order to meet schedules, requiring the users to work around errors. Instead quality would be the primary consideration[136].

The Guidance Software Task Force

Despite postfire improvements, Apollo software had more hurdles to clear. NASA was aware of continuing concern about Apollo s computer programs. Associate Administrator for Manned Spaceflight George E. Mueller formed a Guidance Software Task Force on December 18, 1967 to study ways of improving development and verification . The group met 14 times at various locations before its final report in September 1968[137].

[52] Even while the Task Force was investigating, Mueller took other steps to challenge MIT. A Software Review Board re-examined the software requirements for the lunar mission in early February 1968. The board judged the programs to be too sophisticated and complex, and Mueller requested that they aim for a 50% reduction in the programs, with increased propellant consumption allowed as a tradeoff[138]. An aide reported that Mueller was convinced that MIT "might not provide a reliable, checked-out program on schedule" for the lunar landing mission[139].

The recommended 50% scrub did not occur, and the final report of the Task Force was very sympathetic to the problems involved in developing flight software. It recommended standardization of symbols, constants, and variable names used at both Houston and Huntsville to make communication and coding easier[140]. The Task Force acknowledged that requirements would always be dynamic and that development schedules would always be accelerated, but rather than using this for an excuse for poor quality, the group recommended that software not be slighted in future manned programs. Adequate resources and personnel were to be assigned early to this "vital and underestimated area[141]". This realization would have great effect on managing later software development for the Space Transportation System

Mueller remained concerned about software even after the Task Force dissolved. On March 6, 1969, he wrote a letter to Robert Gilruth, NASA deputy administrator, complaining that software changes were being made too haphazardly and should receive more attention, equal to that given to hardware change requests. Gilruth replied five days later, disagreeing, noting that the Configuration Control Board and other committees formed an interlocking system adequate for change control[142].

Lessons of the Apollo Software Development Process

Overcoming the problems of the Apollo software, NASA did successfully land a man on the moon using programs certifiably adequate for the purpose. No one doubted the quality of the software eventually produced by MIT nor the dedication and ability of the programmers and managers at the Instrumentation Lab. It was the process used in software development that caused great concern, and NASA helped to improve it[143]. The lessons of this endeavor were the same learned by almost every other large system development team of the 1 960s: (a) documentation is crucial, (b) verification must proceed through several levels, (c) requirements must be clearly defined and careful! managed, (d) good development plans should be created and [53] executed, and (e) more programmers do not mean faster development. Fortunately, no software disasters occurred as a result of the rush to the moon, which is more a tribute to the ability of the individuals doing the work than to the quality of the tools they used.
