3

The Skylab Computer System

Skylab, America's first orbital workshop, carried a highly successful computer system. For much of the operating life of the space station, the computer was not just the fourth crew member but the *only* crew member. It made a large contribution to saving the mission during the 2 weeks after the troubled launch and later helped control Skylab during the last year before re-entry. The entire system functioned without error or failure for over 600 days of operation, even after a 4-year and 30-day interruption. It is significant as the first spaceborne computer system to have redundancy management software. The software development for the system followed strict engineering principles, producing a fully verified and reliable real-time program.

The record of the computer system stands in contrast to that of the workshop itself. NASA launched Skylab on May 14, 1973 on a Saturn V booster. The first two stages put the modified S-IVB third stage into orbit. The S-IVB contained the workshop, which included a solar telescope mount and living and working quarters. The plan was to launch the first crew the next day aboard a Saturn IB carrying an Apollo command and service module. However, shortly after achieving orbit, telemetry from the unmanned Skylab indicated that one of the two wings of solar panels was missing and the other had not deployed. The panels on the Apollo Telescope Mount (ATM) had opened properly but they were too small to supply power for the whole workshop. In addition, the gyros were drifting and the thermal shield was damaged. These failures caused concern that the interior of the space station would overheat and destroy the equipment. The damage was so serious that for the first 3 or 4 hours the ground controllers felt that NASA would be fortunate if the systems were to function for 1 day[1]. However, by using the computer system that controlled the workshop's attitude, the ground controllers were able to keep the Skylab at angles to the sun such that the equipment would be exposed to tolerable temperatures in the laboratory in concert with generating adequate power from the remaining solar panels. At times these were conflicting requirements. This had to be done for 2 weeks while engineers prepared repair materials for the crew to fix the workshop. Controller Steven Bales remembered that time as "the hardest 2 weeks I have ever spent," since a 24-hour watch had to be maintained on the attitude and temperature[2].

The computer system again served as "captain" during the entire Skylab reactivation. The workshop systems were shut down on February 9, 1974, after the last crew left. NASA expected that the Skylab would stay in orbit until the mid-1980s. By that time the Space Shuttle would be operational and, it was thought, could be used to bring up rockets to boost the laboratory into a higher orbit. However, unexpected solar activity in the mid-1970s resulted in an increase in the density of the atmosphere, so the Skylab's orbit decayed at a much faster rate than projected[3].

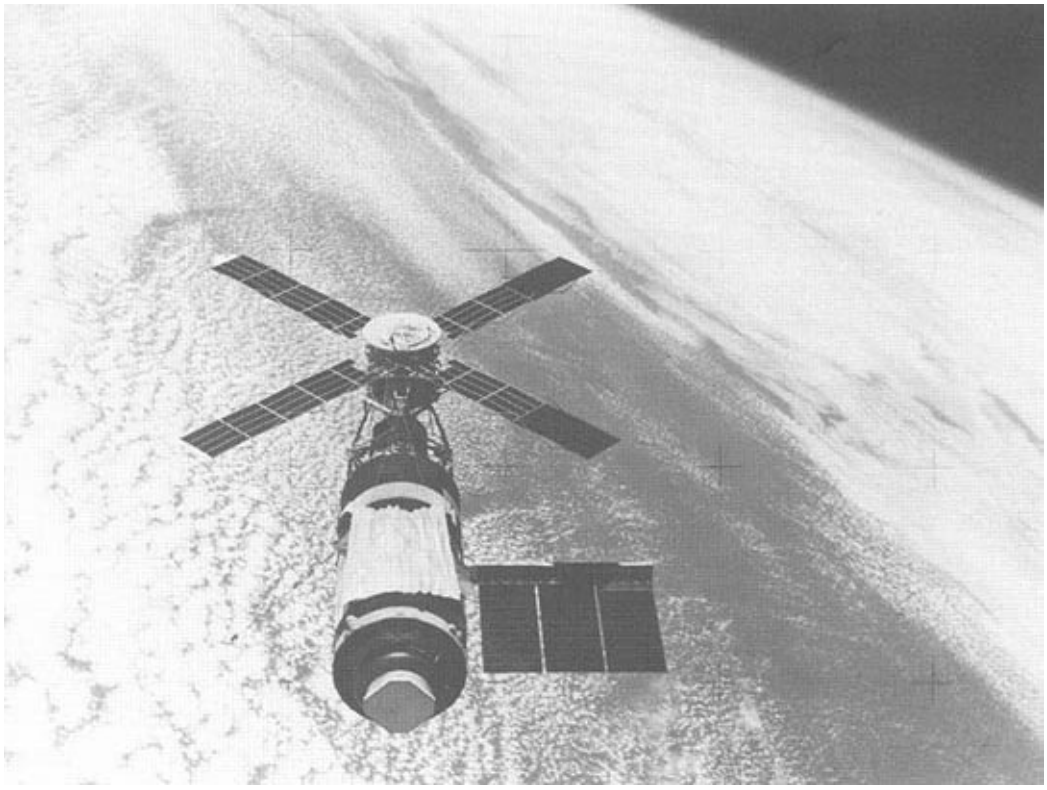

Figure 3–1. Skylab in orbit. Note the foil sun shield above the center section and the missing large solar panel. The Apollo Telescope Mount is the section with the "windmill" solar panels. (NASA photo 74-H-98)

By 1978, the predicted re-entry time was to be late that year or in early 1979. NASA decided to attempt to change the attitude of the workshop so that minimal drag would ensue. In this way, the orbit might be maintained until the Shuttle could rescue the space station. Engineers reactivated and reprogrammed the computer to maintain the proper attitude and, later, to control the re-entry when NASA abandoned the attempt to maintain orbit. They accomplished this over 4 years after the computer was shutdown.

The need for the computer system that served Skylab so well was not apparent until the original "wet workshop" concept (the laboratory to be assembled in space inside of the empty propellant tanks of the last stage of the launch vehicle) had progressed through more sophisticated designs to the eventual "dry workshop"[4]. In December 1968, NASA decided to acquire a dual computer system to help control attitude while in orbit[5]. Attitude control was crucial to the success of the solar experiments. In fact, the name of the computer reflects this: Apollo Telescope Mount Digital Computer (ATMDC). Two of these computers were a part of the Skylab Attitude and Pointing Control System (APCS), which consisted of a number of other components, such as an interface unit, magnetic tape memory, control moment

gyros, the thrustor attitude control system, sun sensors, a star tracker, and nine rate gyros[6].

Marshall Space Flight Center devised this complex system—a pioneering effort because it represents the first fully digital control system on a manned spacecraft[7]. Its mission-critical status led to the use of extensive redundancy in its design, in both hardware and software. The computer system not only managed its own redundancy, but all redundant hardware on the spacecraft[8]. The uniqueness and complexity of the control laws associated with the control moment gyro attitude system led one NASA engineer to refer to it as "a crazy animal"[9]. It was up to the Skylab computer system to tame it.

HARDWARE

The choice of a central processor for the Skylab computer system marked a break from NASA's previous practice. The Gemini and Apollo computer systems were custom-built processors. Apollo did have an immediate predecessor, but the number of changes necessary before flight negated most of its resemblance to the Polaris system. To the contrary, Skylab and, later, the Shuttle, used "off-the-shelf" IBM 4Pi series processors, though they both needed the addition of a customized I/O system, a simpler and necessarily idiosyncratic component. By using existing computers, NASA avoided the serious problems associated with man-rating a new system encountered during the Apollo program.

The 4Pi descended directly from the System 360 architecture IBM developed in the early 1960s. Some 4Pis were at work in aircraft by the latter part of that decade. The top-of-the-line 4Pi is the AP-101, eventually used in the F-15, B-52, and Shuttle. The version on board Skylab was the TC-1, which used a 16-bit word, in contrast to the AP-101's 32 bits. A TC-1 processor, an interface controller, an I/O assembly, and a power supply made up an ATMDC[10]. Each flight computer had a memory of 16,384 words[11]. This memory was a destructive readout core memory, which means that the bits were erased as they were read and that the memory location had to be refreshed with the contents of a buffer register, which saved a copy of the bits before they were passed on to the processor. The memory was in two modules of 8K words each[12]. Addressing ranged from 0 to 8K, with a hardware switch determining which module was being accessed[13]. The redundant computer system was composed of two processors attached to a single Workshop Computer Interface Unit. The unit consisted of two I/O sections (one for each computer), a common section, and a power supply[14]. Only the I/O section connected to the active computer was powered. The inactive computer and its

I/O section of the interface unit were not powered. The common section contained a 64-bit transfer register and timer associated with redundancy management[15]. The transfer register and timer were the only parts of Skylab that consisted of triple modular redundant (TMR) circuits[16]. Basically, TMR circuits sent signals in triplicate on separate channels and then voted. The single output from a TMR voter represented either two or three identical inputs.

The final component of the computer subsystem was the Memory Load Unit. The original design did not contain one, but, like the Gemini Auxiliary Tape Memory, engineers later added it. Whereas the Gemini tape unit was useful in handling memory overloads, designers included the Skylab tape unit to further increase the reliability of the system. It carried a 16K software load and an 8K load that could be written into either module of either memory of the ATDCs. If up to three modules failed, the mission could continue with reduced capabilities with an 8K program loaded into the remaining module. This raised the total reliability of the system from a factor of 0.87 to 0.97[17]. The tape load would take a maximum of 11 seconds[18].

NASA decided to add the Memory Load Unit in the summer of 1971, when both IBM and Marshall realized that a Borg–Warner tape unit, like the two already used as telemetry recorders, could be upgraded for program storage. IBM imposed some manufacturing changes on the recorders (primarily piece part screening) to make the process more nearly match the care taken in constructing the computers[19].

NASA awarded the contract for the computer system to IBM on March 5, 1969[20]. By October, designers froze the choice of processors and their configuration, a decision heavily influenced by the concern for redundancy and reliability[21]. The first computer was delivered on December 23, 1969. IBM eventually built 10, the final 2 being the flight versions, which went to NASA on February 11, 1972, over a year before launch. Two of the ATMDCs and an interface unit were turned over to IBM for use in testing both hardware and software, ensuring that the final verification would be on actual equipment rather than simulators[22].

IBM took great pride in delivering on time without sacrificing reliability. In applying Saturn development techniques to the Skylab equipment, for example, IBM required all piece parts to exceed expected stress levels[23], and prepared the ATMDC for thermal conditions, the most dangerous stress to electronic components[24]. A number of design problems, including thermal and vibration difficulties, analog conversion inaccuracies, and interconnection failures, had to be overcome[25]. To make up time lost handling these problems, IBM sometimes went to a 7-day, three-shift debugging cycle[26].

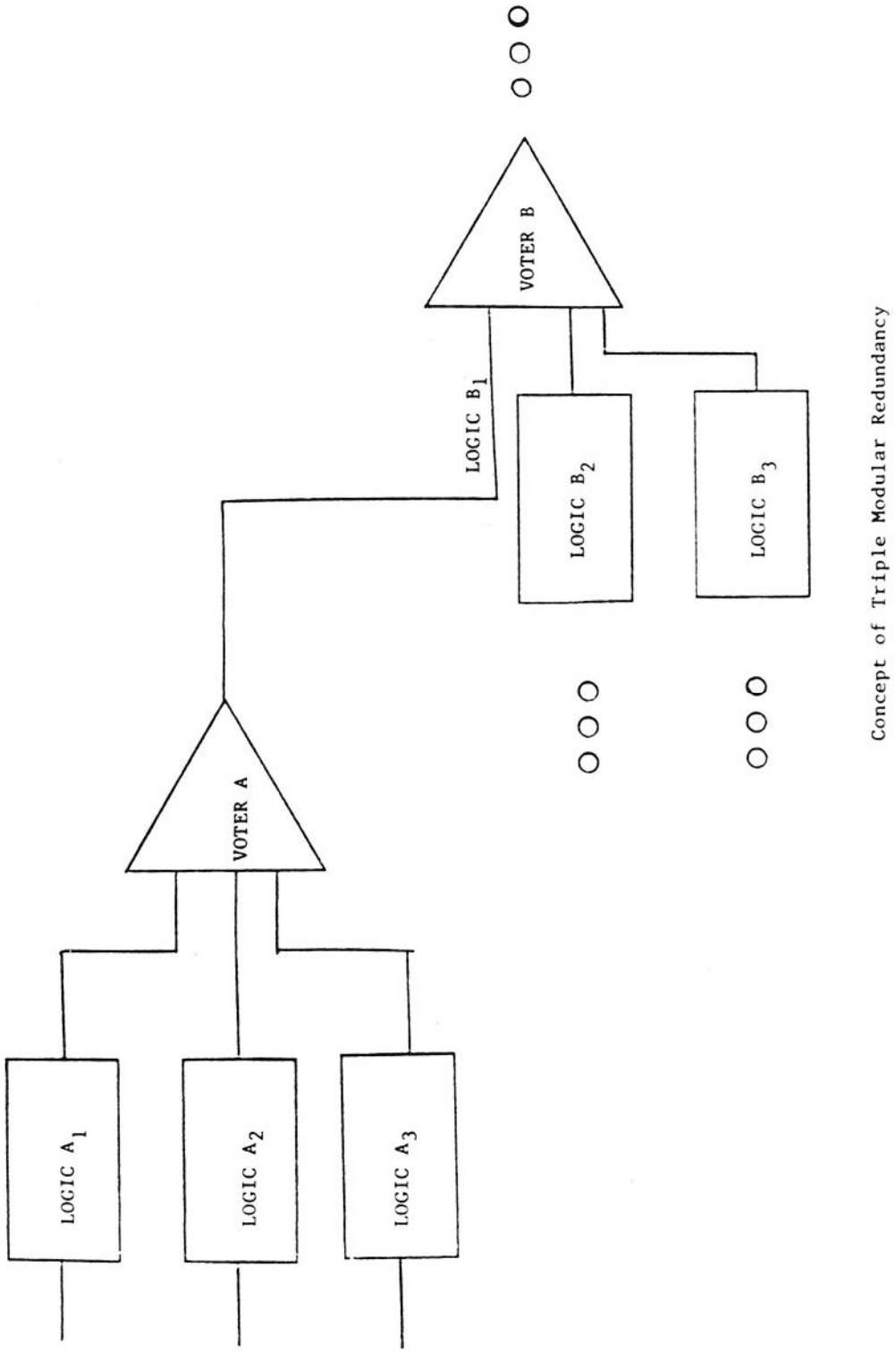

Figure 3–2. The concept of Triple Modular Redundancy.

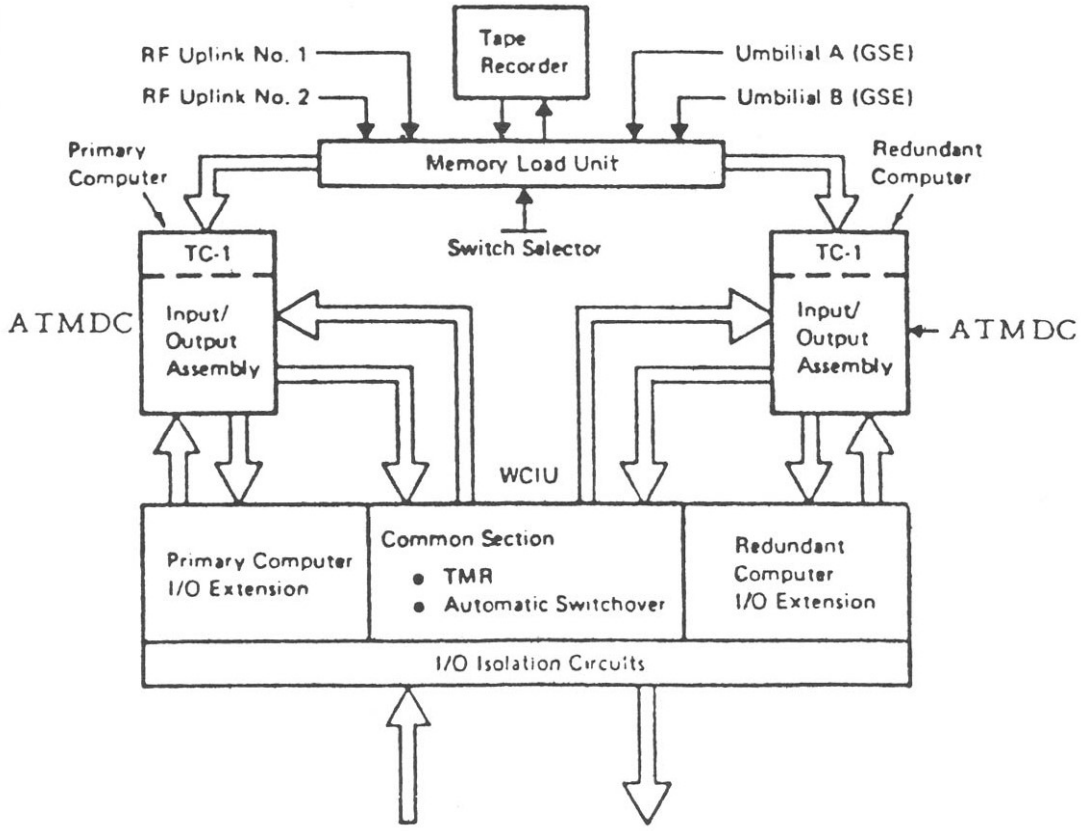

Figure 3–3. A block diagram of the Skylab Computer System with the dual ATMDCs, tape memory, and common section shown. (From IBM, *Skylab Operation Assessment, ATMDC*, 1974)

Probably due to the care taken in manufacture, the computer system had no failures. A planned ground-initiated switch-over from the primary to the secondary computer occurred after 630 hours of orbital operations. The second computer then ran the remainder of the 271-day mission[27]. On the final day, the system did another switch-over and used the tape unit for the first time, primarily to prove that it would work. A transmission of software from the ground to the computer was also practiced. IBM's reports of the performance of the hardware are quite self-congratulatory but, based on the actual record, justified.

SOFTWARE

IBM wanted to do a careful job on the software for Skylab. In the late 1960s and early 1970s, the company internally pushed the development and implementation of software engineering techniques. IBM learned many lessons from the creation of the OS/360 operating system, and various government-related projects. Two IBM software management experts, Harlan Mills and Frederick Brooks, circulated these lessons both within IBM and to the computing public[28]. The small size (16K) of the Skylab software and correspondingly small group of programmers assigned to write it (never more than 75 people, not all of whom were programmers, and only 5 or 6 for the reactivation software), meant that the difficulties in communication and configuration control associated with large projects were not as much of a factor. Also the IBM programmers were specialists. MIT assigned engineers to the programming of the Apollo computer, assuming that it was easier to teach an engineer to program than to teach a programmer the nuances of the system. This turned out to be a mistake, which MIT acknowledged[29]. Thus, the stage was set for IBM to produce a superb real-time program. However, the complexity of the control moment laws, the redundancy management needs, and the inevitable memory overrun kept the development from being simple.

Requirements Definition and Design

IBM and NASA jointly defined the requirements for the Skylab software. Marshall Space Flight Center delivered the detailed requirements for the control laws, navigation, and momentum management, leaving lesser items such as I/O handling to the contractor. IBM and NASA made a parallel effort to determine if the equations actually worked[30]. The result was the Program Requirements Document (PRD), issued July 1, 1970[31].

The actual design, the Program Definition Document (PDD), was released later and served as the baseline for the software, which meant that the design could not be changed without formal review. The software resulting from these documents ranged from 9,000 words to nearly 20,000 words of memory. Since the memory size of the computer was just over 16,000 words, a “scrub” was necessary, continuing the NASA tradition of exceeding the memory size of an already-procured computer by the time the planners knew the final requirements. Managers had not yet learned that software needs should drive the hardware choices. Engineers changed the control moment gyro logic to reduce core usage and made other cuts[32]. Memory

became the prime consideration in allowing requirements changes[33].

Architecture and Coding

Skylab gave IBM an opportunity to demonstrate how to do software development right. The company carefully separated the production process into strictly designed phases. Two different flight loads resulted: one full-function program that filled the 16K memory, and an 8K version as a backup that needed only one module for storage. These two programs needed slightly different architectures, or schemes for organizing the execution of functions, which made the job tougher. Also increasing complexity was the requirement for redundancy management. An advanced development environment helped keep the complexity under control.

Production Phases

IBM developed the software load for Skylab in four baselined phases. Originally, three were planned: Phase I, Phase II, and Final, but numerous changes made during Phase I required an intermediate stage Phase IA. Crews used the software resulting from Phase IA for training in the simulators in Houston[34].

The PDD for Phase I was released on November 4, 1970, and coding began[35]. The Phase I program contained most of the major components of the eventual flight load, including discrete I/O and interrupt processing, command system processing, initialization, redundancy management, attitude reference determination, attitude control, momentum desaturation, maneuvering, navigation and timing, ATM experiment control, displays, telemetry, and algorithms for utilities[36]. IBM's programming team completed and released the Phase I program for verification on June 23, 1971. It consisted of 16,224 words, filling about 99% of the computer's memory[37].

It was this situation that led to the added phase, which was chiefly a memory scrub. Not only was Phase I a fairly extensive program, three modules still had to be coded and many changes would likely occur in the nearly 20 months remaining before launch[38]. By the time IBM delivered Phase IA on February 9, 1972, it had incorporated 45 waivers and 105 software change requests (SWCR) made after the thirteenth revision of the design[39]. This meant that nearly 40% of the original program was changed. Even with the attention to memory size, the new software amounted to 16,111 words, or 98.3% of the locations.

Phase II represented another extensive revision of the software. The baseline for it was Phase IA plus 49 approved change requests. By delivery on August 28, 1972, 102 additional changes had been incorporated and the design was up to revision 19[40]. Therefore, software engineers modified about 35% of the program. The memory usage rose to 99.7%, or 16,338 locations. The final version reduced this to 16,329 words. The difference between Phase II and the flight release was only 17 additional changes. IBM made the delivery March 20, 1973, 2 months before the launch.

Architecture: The 16K Program

The ATMDC software divided into an executive and applications modules. The executive module handled the priority multitasking, interrupt processing, supporting the interval timer and also basic timekeeping chores[41]. Applications consisted of three major groups: time-dependent functions, asynchronous functions, and utilities. Time-dependent functions were executed in three cycles, with the possibility of higher priority jobs interrupting the currently running module. The cycles were differentiated by time: There was a "slow loop" each second, an "intermediate loop" executing five times each second, and the switch-over processor running each half second[42]. Designers grouped appropriate modules in a cycle. An exception to the cycle groupings, but nevertheless time dependent, was the output-write routine, which was run between intermediate loops in order to take more efficient advantage of the system resources. The switch-over process aided in redundancy management, as explained below. Asynchronous functions could be called at any time, one of which was telemetry, which sent 24 strings of 50 bits per second. The other was the command system, which could receive signals from either the ground or the Digital Address System (DAS, the crew interface) in the workshop. Those signals resulted in interrupts. Utility functions included such common algorithms as square root, sine, and cosine, and unique functions such as gimbal angle computations and quaternion multiplication[43].

Interrupt handling was quite straightforward. Each application module had a specific priority ranking. Tasks could be requested by several means, such as interrupts, discrete signals, elapsed time, or by the direct request of another program. Any current task could be interrupted when a new task was requested. The priority of the new task was immediately entered into the priority level control tables. If the new task was of a higher priority than the current task, the computer did the new one first. When telemetry or the command system requested a task, its priority was entered on the table, just like tasks called in the other ways[44]. The standard telemetry signal

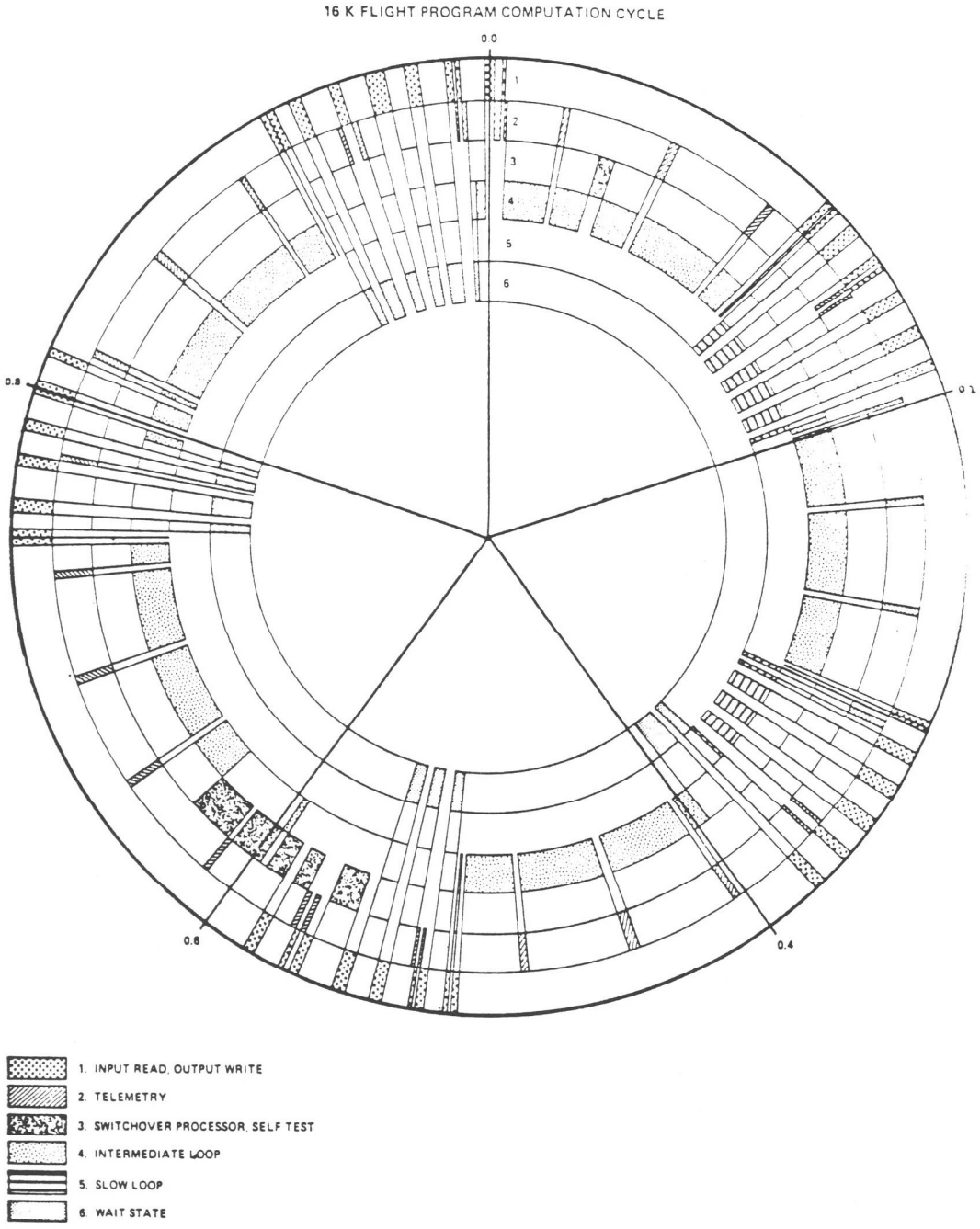

Figure 3–4. The real-time cycle of the Skylab 16K flight program. (From IBM, *Skylab Operation Assessment, ATMDC*, 1974)

functioning as a Digital Command System (DCS) word consisted of 35 bits. Buried in it were an enable bit, an execute bit, and 12 information bits. The enable and execute bits caused an interrupt, making it possible for the data to be stored[45].

The 16K program had a computation cycle consisting of six levels: experiment input, Control Moment Gyro gimbal rates, Workshop Computer Interface Unit tests, and the command system processor; telemetry output; the switch-over timer (reset each second) and 64-bit transfer register (refreshed about once every 17 seconds); the intermediate loop (made up of Control Moment Gyro control); the slow loop (containing timing, navigation, maneuver, momentum management, display, redundancy, self test, and experiment support functions); and the "wait" state (when all functions in a particular cycle finished, about 15% of cycle time in the flight release of the program, depending on the number and nature of interrupts[46].

The 8K Program

The 8K program was strongly related to the 16K program in that the larger version served as the model for the smaller. Its design, released April 3, 1972, developed from the Phase IA version of the software. IBM delivered the 8K program on November 14, 1972 after 10 weeks of verification activity. The functions of the short program were largely limited to attitude control and solar experiment activity and data handling[47]. It was 8,001 words in length. IBM reduced the number of levels in the computation cycle of the 8K program to four: Level I handled command processing and I/O to the Gyros, Level II did telemetry, Level III consisted of the time-dependent functions from both the original intermediate loop and slow loop, and Level IV was the wait state[48].

Redundancy Management

All mission-critical systems in Skylab were redundant. The computer program contained 1,366 words of redundancy management software[49]. At less than 10% of the total memory, it was a bargain. Managing redundancy with stand-alone hardware and solely mechanical switching would have added much more cost, weight, and complexity to the workshop design, with the loss of a certain amount of reliability.

The redundancy management software consisted of two parts: self tests of the computer system and an error detection program for

mission-critical hardware not in the computer system. Self tests of the computer were quite extensive: Logic tests might involve doing a Boolean OR operation on the contents of a register to see if a carry occurred; operation tests required executing EXCHANGE and LOAD instructions; and arithmetic tests meant executing an ADD and checking for planned answer[50]. IBM also designed tests for memory addressing and I/O[51].

The error detection program examined critical signs in several systems. If a failure was detected in attitude control hardware such as the Control Moment Gyros, rate gyros or acquisition sun sensors, then backups or reconfigurations were activated[52]. During the mission, one Control Gyro and several of the rate gyros failed. In fact, a "six-pack" of replacement rate gyros had to be brought up by the second crew.

Switch-over between the two computers was handled by the error detection program or automatically activated by the TMR timer circuits. If self tests indicated a computer hardware failure or that the software was not properly maintaining the workshop's attitude, switch-over would then be initiated. The timers were supposed to be reset about once each second during the computation cycle, after which they then counted down until reset. If two of the three reached zero, then switch-over occurred[53]. Besides automatic switch-over, the crew or the ground could initiate it, as actually happened in mid-mission. So that the secondary computer would be properly activated, a 64-bit transfer register was kept loaded with relevant data. This register, like the timers, consisted of TMR circuits. Great care was taken to ensure that data loaded into the transfer register were uncontaminated. A write operation to the register was restricted in length to a period of 672 microseconds plus or minus 20%, which was just about how long it took to write 64 bits into a redundant circuit. This operation could only take place after 1.5 to 2.75 seconds had elapsed since the last write, so the computer would not accept transient signals as correct data and a new write could not interfere with an earlier write[54]. Besides this "time-out feature," the transfer register could only be refreshed *after* a successful execution of the error detection program[55]. This way, data could not be written to the register from a failed computer.

The redundancy management software was a step toward the eventual Shuttle redundancy management scheme. Previously, IBM had used TMR hardware to ensure reliability. This system, with its watchdog timer, was software based and, in effect, saved space and weight. Two ATMDCs were smaller and required less power than a single TMR computer of equal reliability.

The Development Environment and Integration

The Skylab software development was done in a programming environment that took advantage of useful software tools and proper integration techniques. Binary code for the computer was in hexadecimal (base 16) format, and loaded in that format[56]. Hand coding in hex is rather tedious, so IBM prepared an assembler to translate mnemonics into it. They also provided a relocatable loader for placing separately coded modules in contiguous memory locations. Macros, blocks of frequently used code, were kept in common libraries. Listings of programs and the original source resided in an IBM System 360/75[57]. This environment was small compared with the later Software Production Facility for the Shuttle, but the concept of a good tool set, promoted by IBM's Mills and Brooks, was well realized.

Integration of the Skylab software followed a top-down approach: The program was highly modular so as to keep individual functions separate for easy modification and also simple enough for a single programmer to handle. The executive and major subprocesses were coded and integrated first; then the remaining modules were added. The modules were grouped into three batches, so all the modules in a batch were added and tested, then the next batch would be added, and so on[58]. This helped in the integration process.

Verification

The software for Skylab was one of the most extensively verified systems of its era. Since it was a real-time program, verification was more difficult than a corresponding batch program because it is hard to replicate test inputs when interrupts can occur at any time; thus, a combination of simulators is needed to properly verify a real-time program.

IBM used a number of different simulation configurations in the verification process. The AS-II simulator consisted of a System 360/75 used for analysis of the Skylab while it was in orbit. It could evaluate the effects of changes to the flight program. The Skylab Workshop Simulator (SWS) was an all-digital simulation used in developing the initial software, as well as verification. It ran at a 3.5/1 ratio of execution time to real time. The SWS was so effective that it once correctly identified a deficiency in the requirements relating to the Control Moment Gyro system. The Skylab Hybrid Simulator (SHS) included some analog circuits for greater fidelity. One of the most effective simulators was a System 360/44 connected to an actual ATMDC; the program in the 44 could simulate six degrees of freedom[59].

The verification process was scheduled for the final 10 weeks prior to the delivery of any software phase. The process included validation of the baseline program to the requirements, coding analysis, logic analysis, equation implementation tests, performance evaluations, and mission procedure validation. The AS-II did the logic analysis and was designed to trace all logic paths through the software. The 360/44 and ATMDC system did performance tests since it was near real time in operation[60]. The digital simulators could be stopped in order to insert program changes. Tracing was also possible[61]. Combining simulators and software verification tools contributed to a high level of confidence that was confirmed in actual performance.

USER INTERFACES

NASA and IBM designed the computer system to operate autonomously. One crewman reported "not much interaction" with the system at all[62], but the capability was present for significant activity if needed[63]. The crew could enter data and actually make changes in the software through a keyboard located in the DAS on the ATM Control and Display Console.

The DAS had only 10 keys and a three-position switch. The keys were the digits 0–7 (all entries were in octal), a clear key, and an enter key. The switch could select either power bus one or two, or be off. Above the DAS was an "Orbit Phase" panel containing a digital readout of minutes and seconds to the next orbital benchmark. When the first keystroke of a five-digit command was made, the uplink DCS commands were inhibited, and the time remaining clock inputs were inhibited, so that the clock digits could be used for displaying the keystrokes. In that mode, five digits would be lit instead of four. The remaining four keystrokes were the data/command inputs[64]. The display of the keystrokes represented an echo. If the sequence was correct, the astronaut pressed the enter key, or else he would restart the input process. Pressing the clear key brought back the digital clock. The rather limited nature of this command system indicates that it was intended for sparing use.

Besides the DAS, one other switch on the control panel related to the computer system. In the "Attitude Control" area of the panel was a three-position switch that controlled which computer was in actual use. It could be set for automatic (and usually was), in which case the redundancy management software would take care of matters. Alternately, the crew could purposely select either the primary or secondary computer. If either of these was selected, then automatic changeover was inhibited[65]. The switch gave the crew protection from

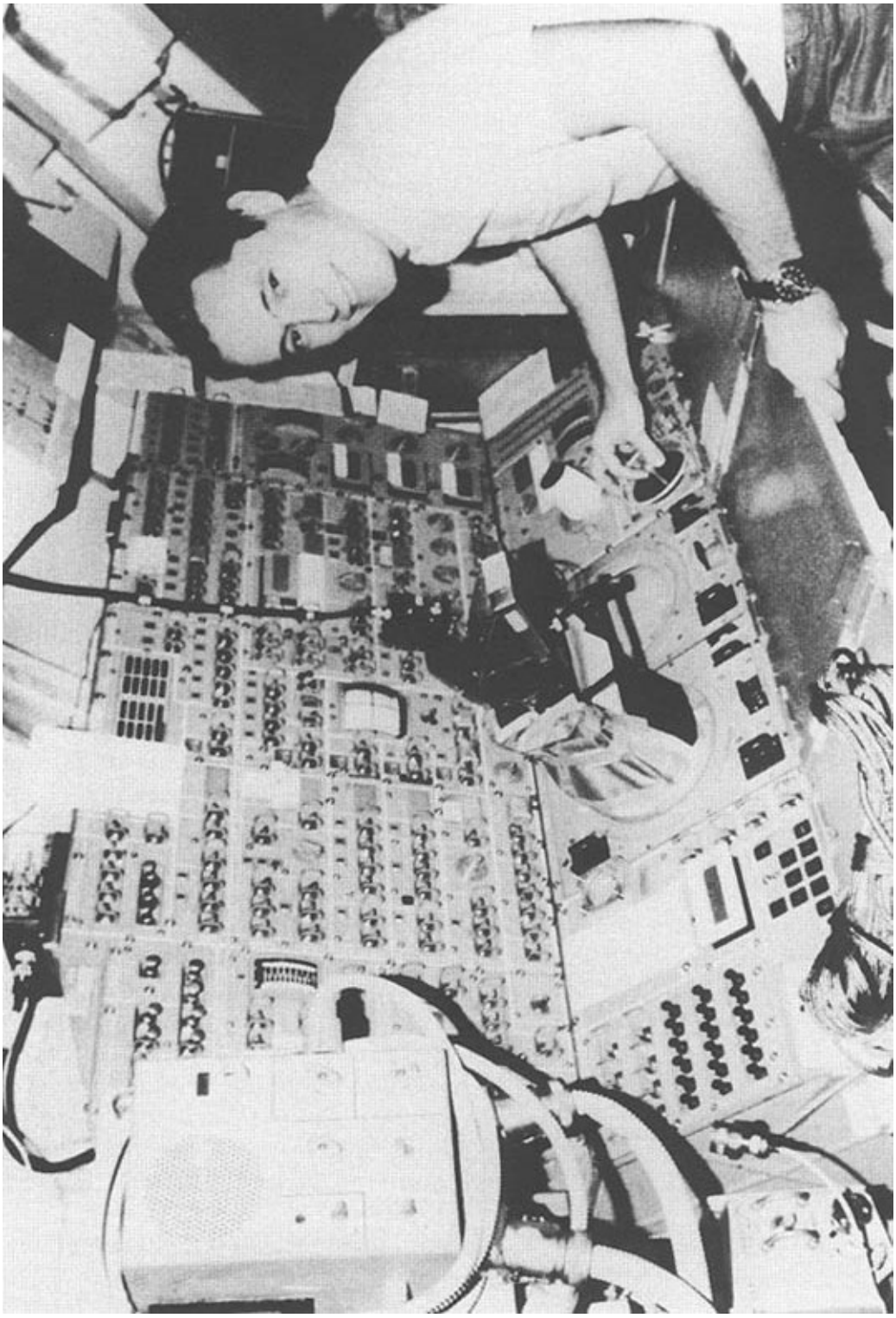

Figure 3–5. Dr. Edward Gibson at the Apollo Telescope Mount Control console. The interface to the digital computer is at lower left, on the panel immediately above the coil of cable. (NASA photo 4-60352)

failure of the redundancy management software. Incidentally, the

switch was not a common three-position toggle switch but, instead, required the crew to pull out and rotate the post. This protected the crew from accidental switching.

THE REACTIVATION MISSION

The Skylab Reactivation Mission represents one of the most interesting examples of the autonomy and reliability of manned spacecraft computers. The original Skylab mission lasted 272 days with long unmanned periods. The reactivation mission, flown entirely under computer control, lasted 393 days. Therefore, the bulk of the activated life of the space laboratory fully depended on the ATMDCs.

When it was obvious that the Workshop was going to fall to the earth long before a rescue mission could be launched, NASA began studying methods of prolonging the orbital life of the spacecraft. Even though the atmosphere is very thin at the altitude Skylab was flying, the drag produced on the spacecraft was highly related to its attitude with respect to its direction of flight (velocity vector). During most of the manned mission periods Skylab flew in solar inertial (SI) mode, in which the lab was kept perpendicular to the sun to provide maximum exposure for the solar collectors. Momentum desaturation maneuvers were done on the dark side of the earth to compensate for bias momentum buildup resulting from noncyclic torques acting on the spacecraft. The SI mode was high drag, so engineers devised two new modes, end-on-velocity-vector (EOVV) and torque equilibrium attitude (TEA). EOVV pointed the narrow end of the lab in the direction of flight, minimizing the aerodynamic drag on the vehicle. TEA could control the re-entry, using the gravity gradient and gyroscopic torques to counterbalance the aerodynamic torque. Only in this way could the Workshop be controlled below 140 nautical miles altitude[66].

Use of the new modes required that they be coded and transmitted to the computers in orbit. First it was necessary to discover whether or not the computers still functioned. Since the ATMDC used destructive readout core memories, there was some concern that the software might have been destroyed during restart tests if the refreshment hardware had failed. On March 6, 1978, NASA engineers at the Bermuda tracking station ordered portions of Skylab to activate. On March 11, the ATMDC powered up for 5 minutes to obtain telemetry confirmation that it was still functioning. The software resumed the program cycle where it had left off 4 years and 30 days earlier. As far as the computer was concerned, it had suffered a temporary power transient[67]!

When IBM began to make preparations to modify the software, it discovered that there was almost nothing with which to work. The carefully constructed tools used in the original software effort were

dispersed beyond recall, and, worse yet, the last of the source code for the flight programs had been deleted just weeks beforehand. This meant that changes to the software would have to be *hand coded* in hexadecimal, as the assembler could not be used—a risky venture in terms of ensuring accuracy. Eventually it became necessary to repunch the 2,516 cards of a listing of the most recent flight program, and IBM hired a subcontractor for the purpose[68].

Engineers could not test this software with the same high fidelity as during the original development. They abandoned plans for real time simulations because they could not find enough parts of any of the original simulators. Interpretive simulation could be performed because the tapes for that form of testing had been saved. However, the interpretive simulator ran 20 times slower than real time, so less testing was possible[69].

IBM approached the modification using the same principles as in the original production. The baseline software for the reactivation was Flight Program 80, including change request 3091, which was already in the second computer. Software changes for reactivation were simply handled as routine change requests. They placed the EOVV software in memory previously occupied by experiment calibration and other functions useless in the new mission. TEA replaced the command and display software[70].

When the software was ready for flight, NASA uplinked it to a reserve area of memory and then downlinked and manually verified it. If it passed the verification, engineers gave a command to activate it. The reprogramming was generally successful. The four people assigned to the software revision maintained IBM's record of quality throughout the reactivation mission[71].

CONCLUSIONS

The Skylab program demonstrated that careful management of software development, including strict control of changes, extensive and preplanned verification, and the use of adequate development tools, results in quality software with high reliability. Attention to piece part quality in hardware development and the use of redundancy resulted in reliable computers. However, it must be stressed that part of the success of the software management and the hardware development was due to the small size of both. Few programmers were involved in initial program design and writing. This meant that communications between programmers and teams were relatively minimal. The fact that IBM produced just 10 computers and really needed to ensure the success of just 2 of those helped in focusing the quality assurance effort expended on the hardware.

What happened after the manned Skylab program demonstrated the need for foresight and proper attention to storage of mission-critical materials until any possibility of their use had gone away. The dispersal of the verification hardware is understandable, as it is expensive to maintain. However, some provision should have been made for retaining mission-unique capabilities such as actual flight hardware. The destruction of the flight tapes and source code for the software by unknown parties was inexcusable. A single high-density disk pack could have held all relevant material.

Skylab marked the beginning of redundant computer hardware on manned spacecraft. It was also the first project that developed software with awareness of proper engineering principles. The Shuttle continued both these concepts but on a much larger and more complex scale.

4

Computers in the Space Shuttle Avionics System

Computers are used more extensively on the Space Transportation System (STS) than on any previous aircraft or spacecraft. In conventional aircraft, mechanical linkages and cables connect pilot controls, such as the rudder pedals and stick, to hydraulic actuators at the control surfaces. However, the Shuttle contains a fully digital fly-by-wire avionics system. All connections are electrical and are routed through computers. To give the spacecraft more autonomy, system management functions (fuel levels, life support, etc.), handled on the ground during previous flight programs, are monitored on board. Software can be adjusted to suit increasingly complex and varied payloads. Subsystems, like the main engines, that had no computer assistance before use them for performance improvement. And, as in Gemini and Apollo, guidance and navigation tasks are accomplished on the Shuttle with computers. All these functions, especially flight control, are critical to mission success; therefore, the computers performing the tasks must be made fail-safe by using redundancy. Meeting these requirements has resulted in one of the most complex software systems ever produced and a computer network within the spacecraft with more powerful hardware than many ground-based computer centers in the mid-1960s.

The major differences between the Shuttle computer system and the systems used on Gemini and Apollo were the choice of an "off-the-shelf" main computer instead of a custom-made machine and the pervasiveness of the system within the spacecraft, since the main computers are the heart of any true avionics system. Avionics (*avi*ation plus electr*onics*) grew in the 1950s and 1960s as electronic devices, especially digital devices, replaced mechanical or analog equipment in aircraft. These digital devices were combined into a coherent system, rather than isolated in function and location within the aircraft. Several modern military airplanes have applied this concept to varying degrees. The FB-111, an Air Force tactical bomber, has a complex avionics system that Rockwell International built just before it was awarded the Shuttle contract[1]; the F-15 fighter used an AP-1 computer in its system. A repackaged version of the F-15's computer became the AP-101 used in the shuttle[2]. However, in no aircraft has the Shuttle's avionics system been matched as yet. For instance, its main computers have to interconnect with other computers in subsystems, such as the controllers on each main engine, whereas most aircraft systems are centered on a single set of machines.

Since the Shuttle is completely dependent on the success of its avionics system, each component must be made failure proof. The method chosen to ensure this is absolute redundancy, often to a depth of four duplicate devices. Managing this level of redundancy became a large problem in itself.

Another result of the pervasive avionics system is that the frequency and sophistication of the crew interaction with the computers exceeds any previous manned space program. A large portion of the software is directed at easing the necessary commanding of the

Figure 4–1. The first launch of the Shuttle *Challenger*, one of a fleet of the most computationally intensive spacecraft ever built. (NASA photo)

computers. In general, software development for the Shuttle has far outstripped any previous NASA ground or flight system in effort and cost. The combination of requirements forced the Agency to pioneer techniques in digital avionics, redundancy management, computer interconnection, and real-time software development.

EVOLUTION OF THE SHUTTLE COMPUTER SYSTEM

Planning for the STS began in the late 1960s, before the first moon landing. Yet, the concept of a winged, reusable spacecraft went back at least to World War II, when the Germans designed a suborbital bomber that would "skip" along the upper atmosphere, dropping bombs at low points in its flight path. In America in the late 1940's, Wernher von Braun, who transported Germany's rocket knowhow to the U.S. Army, proposed a new design that became familiar to millions in the pre-Sputnik era because Walt Disney Studios popularized it in a series of animated television programs about spaceflight. It consisted of a huge booster with dozens of upgraded V-2 engines in the first stage, many more in the second, and a single-engine third stage, topped with a Shuttle-like, delta-winged manned spacecraft.

Because the only reusable part of the von Braun rocket was the final stage, other designers proposed in its place a one-piece shuttle consisting of a very large aerospacecraft that was intended to fly on turbojets or ramjets in the atmosphere before shifting to rocket power when the atmospheric oxygen ran out. Once it returned from orbit, it would fly again under jet power. However, the first version of the reusable spacecraft to actually begin development was the Air Force Dyna-Soar, which had a lifting body orbital vehicle atop a Titan III booster. That project died in the mid-1960s, just before NASA announced a compromise design of desirable features: the expensive components (engines, solid rocket shells, the orbiter) to be reusable; the relatively inexpensive component, the external fuel tank, to be expendable; the orbiter to glide to an unpowered landing[3].

The computer system inside the Shuttle vehicle underwent an evolution as well. NASA gained enough experience with on-board computers during the Gemini and Apollo programs to have a fair idea of what it wanted in the Shuttle. Drawing on this experience, a group of experts on spaceborne computer systems from the Jet Propulsion Laboratory, the Draper Laboratory (renamed during its Apollo efforts) at MIT, and elsewhere collaborated on an internal NASA publication that was a guide to help the designer of embedded spacecraft computers[4]. Individuals contributed additional papers and memos. Preliminary design proposals by potential contractors also influenced the eventual computer system. In one, Rockwell International

teamed up with IBM to submit a system[5]. Previously, in 1967, the Manned Spacecraft Center contracted with IBM for a conceptual study of spaceborne computers[6] and two Huntsville IBM engineers did a shuttle-specific study in 1970[7]. Coupled with IBM Gemini and Saturn experience, the Rockwell/IBM team was hard to beat for technical expertise. NASA also sought further advice from Draper, as it was still heavily involved in Apollo[8]. These varied contributions shaped the final form of the Shuttle's computer system.

There were two aspects of the computer design problem: functions and components. Previous manned programs used computers only for guidance, navigation, and attitude control, but a number of factors in spacecraft design caused the list of computable functions to increase. A 1967 study projected that post-Apollo computing needs would be shaped by more complex spacecraft equipment, longer operational periods, and increased crew sizes[9]. The study suggested three approaches to handling the increased computer requirements. The first assigned a small, special-purpose computer to each task, distributing the processes so that the failure of one computer would not threaten other spacecraft systems. The second approach proposed a central computer with time-sharing capability, thus extending the concepts implemented in Gemini and Apollo. Finally, the study recommended several processors with a common memory (a combination of the features of the first two ideas). This last concept was so popular that by 1971 at least four multiprocessor systems were being developed for NASA's use[10].* The greater appeal of the multiprocessors, and the production of the Skylab dual computer system, replaced the idea of using simplex computer systems that could not be counted on to be 100% reliable on long-duration flights.

On a more detailed level than the overall configuration, experts also realized that increased speed and capacity were needed to effectively handle the greater number of assigned tasks[11]. One engineer suggested that a processor 50% to 100% more powerful than first indicated be procured[12]. This would provide insurance against the capacity problems encountered in Gemini and Apollo and be cheaper than software modifications later. A further requirement for a new manned spacecraft computer was that it be capable of floating-point arithmetic. Previous computers were fixed-point designs, so scaling of the calculations had to be written into the software. Thirty percent of the Apollo software development effort was spent on scaling[13].

*These were: EXAM (*Ex*perimental *A*erospace *M*ultiprocessor) at Johnson Space Center, the Advanced Control, Guidance, and Navigation Computer at MIT, SUMC (*S*pace *U*ltrareliable *M*odular *C*omputer) at Marshall Space Flight Center, and PULPP (*P*arallel *U*ltra *L*ow *P*ower *P*rocessor) at the Goddard Space Flight Center.

One holdover component from the Gemini, Apollo, and Skylab computers remained: core memory. Mostly replaced by semiconductor memories on IC chips, core memory was made up of doughnut-shaped ferrite rings. In the mid-1960s, core memories were determined to be the best choice for manned flight for the indefinite future, because of their reliability and nonvolatility[14]. Over 2,000 core memories flew in aircraft or spacecraft by 1978[15]. The NASA design guide for spacecraft computers recommended use of core memory and that it be large enough to hold all programs necessary for a mission[16]. That way, in emergencies, there would be no delay waiting for programs to be loaded, as in Gemini 8, and the memory could be powered down when unneeded without losing data.

By 1970, several concepts to be used in the Shuttle were chosen. One of these was the use of buses, which Johnson Space Center's Robert Gardiner considered for moving large amounts of data[17]. Instead of having a separate discrete wire for every electronic connection, components would send messages on a small number of buses on a time-shared basis. Such buses were already in use in cabling from the launch center to rockets on the launch pads. Buses were also being considered for military and commercial aircraft, which were becoming quite dependent on electronics. Additionally, there would be two redundant computer systems—though no decision had been made as to how the systems would communicate. In the LEM, the PGNCS had an active backup in the Abort Guidance System (AGS). This was not true redundancy in that the AGS contained a computer with less capacity than the AGC, and so could not complete a mission, just safely abort one. True redundancy, however, meant that each computer system would be capable of doing all mission functions.

Redundancy grew out of NASA's desire to be able to complete a mission even after a failure. In fact, early studies for the Shuttle predicated the concept of "fail operational/fail operational/fail-safe." One failure and the flight can continue, but two failures and the flight must be aborted because the next failure reduces the redundancy to three machines, the minimum necessary for voting. In the 1970 computer arrangement, one special-purpose computer handled flight control functions (the fly-by-wire system), and another general-purpose computer performed guidance, navigation, and data management functions. These two computers had twins and the entire group of four was duplicated to provide the desired layers of redundancy[18].

More concrete proposals came in 1971. Draper presented a couple of plans, one fairly conservative, the other more ambitious. The less expensive version used two sets of two AGCs. These models of the AGC would contain 32K of erasable memory and magnetic tape mass memory instead of the core rope in the original[19]. Redundancy would be provided by a full backup that would be automatically switched into action upon failure of the primary (an idea later abandoned since a

software fault could cause a premature switch-over)[20]. Draper's more expensive, but more robust, plan proposed a "layered collaborative computer system," to provide "significant total, modest individual computing power"[21]. A relatively large multiprocessor was at the heart of this system, with local processors at the subsystem level. This had the potential effect of insulating the central computer from subsystem changes.

Unlike Gemini and Apollo, NASA wanted an off-the-shelf computer system for the Shuttle. If "space rating" a system involved a stricter set of requirements than a military standard[22], starting with a military-rated computer made the next step in the certification process a lot cheaper. Five systems were actively considered in the early 1970s: The IBM 4Pi AP-1, the Autonetics D232, the Control Data Corporation Alpha, the Raytheon RAC-251, and the Honeywell HDC-701[23]. The basic profile of the computer system evolved to the point where expectations included 32-bit word size for accurate calculations, at least 64K of memory, and microprogramming capability[24]. Microprograms are called firmware and contain control programs otherwise realized as hardware. Firmware can be changed to match evolving requirements or circumstances. Thus, a computer could be adapted to a number of functions by revising its instruction set through microcoding.

Despite the fact that Draper Laboratory favored the Autonetics machine, and a NASA engineer estimated that the load on the Shuttle computers would "be heavier than the 4Pi [could] support," the IBM machine was still chosen[25]. The 4Pi AP-1's advantages lay in its history and architecture. Already used in aircraft applications, it was also related to the 4Pi computers on Skylab, which were members of the same architectural family as the IBM System 360 mainframe series. Since the instruction set for the AP-1 and 360 were very similar, experienced 360 programmers would need little retraining. Additionally, a number of software development tools existed for the AP-1 on the 360. As in the other spacecraft computers, no compilers or other program development tools would be carried on-board. All flight programs were developed and tested in ground-based systems, with the binary object code of the programs loaded into the flight computer. Simulators and assemblers for the AP-1 ran on the 360, which could be used to produce code for the target machine. In both the Gemini and Apollo programs, such tools had to be developed from scratch and were expensive.

One further aspect of the evolution of the Shuttle computer systems is that previous manned spacecraft computers were programmed using assembly language or something close to that level. Assembly language is very powerful because use of memory and registers can be strictly controlled. But it is expensive to develop assembly language programs since doing the original coding and verifying that the programs work

properly involve extra care. These programs are neither as readable nor as easily tested as programs written in FORTRAN or other higher-level computer languages. The delays and expense of the Apollo software development, along with the realization that the Shuttle software would be many times as complex, led NASA to encourage development of a language that would be optimal for real-time computing. Estimates were that the software development cycle time for the Shuttle could be reduced 10% to 15% by using such a language[26].

The result was HAL/S, a high-level language that supports vector arithmetic and schedules tasks according to programmer-defined priority levels**. No other early 1970s language adequately provided either capability. Intermetrics, Inc., a Cambridge firm, wrote the compiler for HAL. Ex-Draper Lab people who worked on the Apollo software formed the company in 1969[27].

The proposal to use HAL met vigorous opposition from managers used to assembly language systems. Many employed the same argument mounted against FORTRAN a decade earlier: The compiler would produce code significantly slower or with less efficiency than hand-coded assemblers. High-level languages are strictly for the convenience of programmers. Machines still need their instructions delivered at the binary level. Thus, high-level languages use compilers that translate the language to the point where the machine receives instructions in its own instruction set (excepting certain recently developed LISP machines, in which LISP *is* the native code). Compilers generally do not produce code as well as humans. They simply do it faster and more accurately. However, many engineers felt that optimization of flight code was more important than the gains of using a high-level language. To forestall possible criticism, Richard Parten, the first chief of Johnson's Spacecraft Software Division, ordered a series of benchmark tests. Parten had IBM pick its best assembly language programmers to code a set of test programs. The same functions were also written in HAL and then raced against each other. The running times were sufficiently close to quiet objectors to high-level languages on spacecraft (roughly a 10% to 15% performance difference)[28].

**The origins of the name of the language are unclear. Stanley Kubrick's classic film *2001: A Space Odyssey* (1968) was playing in theaters at about the time the language was being defined. A chief "character" in the film was a murderous computer named HAL. NASA officials deny any relationship between the names. John R. Garman of Johnson Space Center, one of the principals in Shuttle onboard software development, said it may have come from a fellow involved in the early development whose name was Hal. Others suggest it is an acronym for Higher Avionics Language. For a full description of the language and sample programs, see Appendix II.

By 1973, work could begin on the software necessary for the shuttle, as hardware decisions were complete. Conceptually, the shuttle software and hardware came to be known as the Data Processing System (DPS).

THE DPS HARDWARE CONFIGURATION

The DPS hardware in the shuttle avionics system consists of four major components: general-purpose computers, the data bus network, the multifunction cathode ray tube display system, and the mass memory units. Each is a substantial improvement over similar systems in any previous spacecraft. Together, they are a model for future avionics developments.

General-Purpose Computers

NASA uses five general-purpose computers in the Shuttle. Each one is an IBM AP-101 central processing unit (CPU) coupled with a custom-built input/output processor (IOP). The AP-101 has the same type of registers and architecture used in the IBM System 360 and throughout the 4Pi series[29]. IBM announced the 4Pi in 1966, so by the early 1970s, when Shuttle procurement was complete, the machine had had extensive operational use[30]. The AP-101 version, which is an upgraded AP-1, has since been used in the B-52 and B-1B military aircraft and the F-8 digital fly-by-wire experimental aircraft. The central processor in each case is the same, but the IOP is adapted to the particular application.

Although one of the reasons for choosing the AP-101 was its familiar instruction set, some modifications were necessary for the Shuttle version. Since the execution of instructions is dependent on microcode, rather than hardware only, the instruction set could be changed somewhat. Microcode is a set of primitives that can be combined to create new logic paths in the hardware. The AP-101 has room for up to 2,048 microinstructions, 48 bits in length[31].

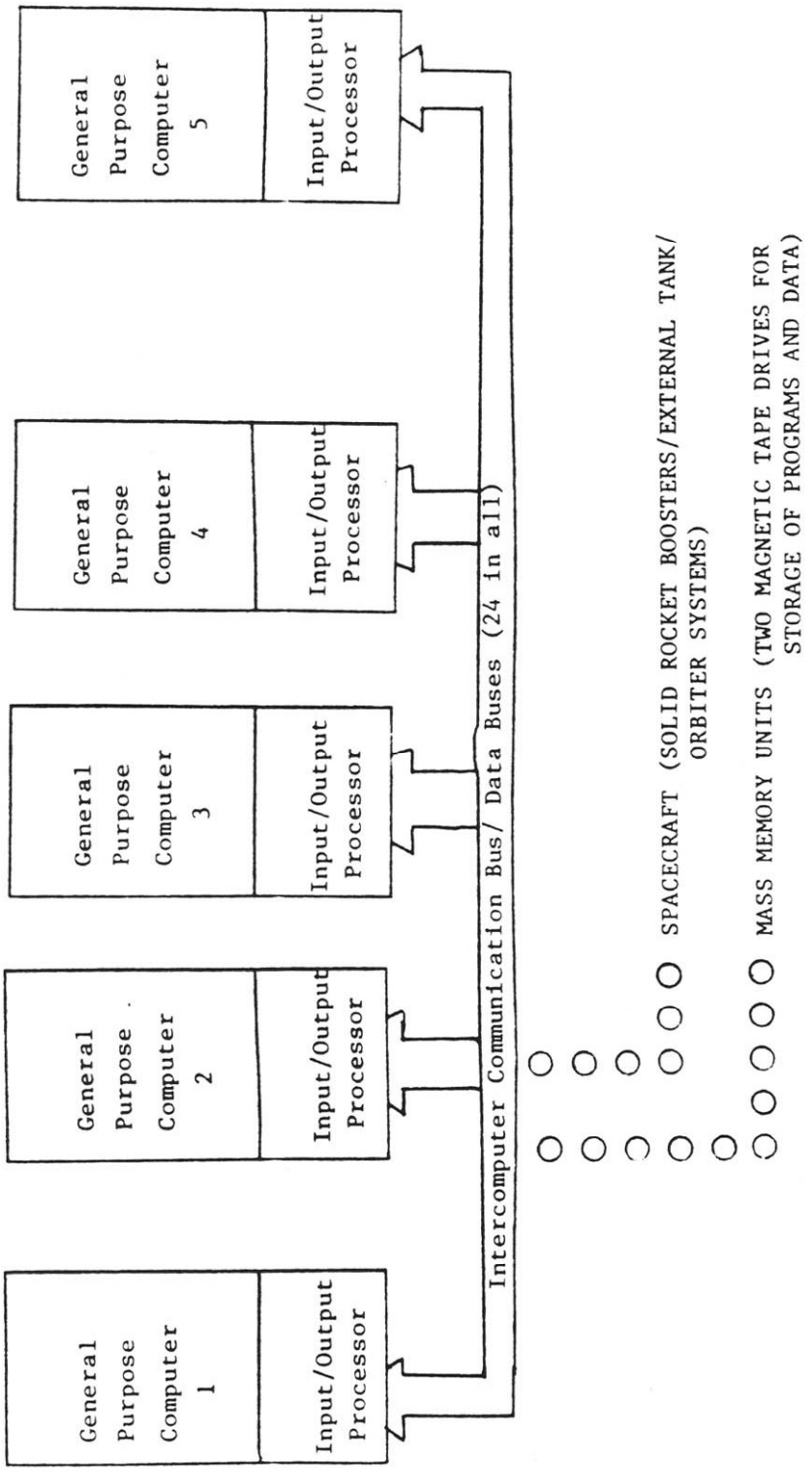

Figure 4–2. A block diagram of the hardware that makes up the Shuttle Data Processing System. The fifth computer is the Backup Flight System computer.

Box 4-1: IBM AP-101 Central Processor and Memory Hardware

Shuttle computers make extensive use of standard ICs. The AP-101 is built using transistor-transistor logic (TTL) semiconductor circuits as the basic building block. The TTL gates are arranged in medium-scale integration (MSI) and large-scale integration (LSI) configurations[32]. The circuits are on boards that can be interchanged as units.

The AP-101 uses a variety of word sizes. Instructions can be either 16 or 32 bits in length. Fixed-point arithmetic, done using fractional numbers stored in two's complement form, also uses 16- and 32-bit lengths. Floating-point arithmetic is done with 32-, 40- and 64-bit words, although the latter are limited to addition and subtraction[33]. Instructions using floating-point take longer to execute than fixed-point arithmetic, and adding is still faster than multiplying; but average speed for the machine is 480,000 instructions per second, compared with 7,000 instructions per second in the Gemini computer[34].

The CPU registers are in three groups. Two sets of eight 32-bit registers are available for fixed-point arithmetic. One set of eight 32-bit registers is for floating point operations[35]. Semiconductor memories are used in the registers instead of discrete components. As a result, the registers are faster than those used on Gemini and Apollo and, since they are available in large sets, can be used to store intermediate results of calculations without having to access core memory. Thus, processing is accelerated and achieves the performance noted above[36].

A program status word (PSW), 64 bits in length, is used to help control interrupts. The PSW contains information such as the next instruction address, current condition code, and any system masks for interrupts[37]. It has to be updated every instruction to stay current[38]. Since the AP-101 allows 61 different interrupt conditions divided into 20 priority levels, it is necessary to have an accurate indication of where a program left off when interrupted[39]. At any given time, several programs are likely to be in a suspended state.

The processor has more than one level of addressing. The common 16-bit address can only directly address 64K words, which was the original memory size of the AP-101. The addressing is extended by replacing the highest order bit with 4 bits from the program status word that indicate which sector of memory to access[40]. This is similar to the scheme used in the AGC when its memory had to be expanded. This configuration allows 131,072 full words (32-bit words) to be addressed. The architecture permits addressing up to 262,144 full words, so memory can be expanded without affecting the processor's design[41].

Due to packaging considerations, the core memory is located partly in the central processor and partly in the IOP (they are boxed separately). However, it is still considered as a single unit for addressing and access. The entire memory is shared, not just the portion located in the individual boxes. Originally, 40K of core were in the CPU and 24K in the IOP. The memory is organized into modules with 18-bit half words. These contain 16 bits of data, a parity bit, and a storage protect bit to prevent unintentional alteration of the data[42]. The original memory modules contained 8K half words, so 6 were needed in the IOP and 10 in the CPU to store 64K full words. Later memory expansion consisted of replacing the CPU memory modules with double-density modules, in which twice the cores are in the same size container as a single-density module[43]. So by the first flight, the Shuttle computer memories were 104K words or 106,496 full words of 32 bits. The memory access time is 400 nanoseconds, quite fast for core.

The eventual Shuttle instruction set contained 154 instructions defined within that 2K memory. However, the expected advantages of the flexibility of microcoding, which influenced the decision to select the AP-101, were lessened by the fact that at least six of the new instructions either did not work properly or performed insufficiently[44]. One NASA manager said that the microcoding was bungled by "the ones and zeroes artists" (referring to the binary numbered nature of microprograms) who apparently tried to do things the tricky way[45]. NASA tried to correct its tendency to underestimate memory size, but was disappointed again on the Shuttle program. One requirement for memory was that it be large enough to contain all the programs necessary for a mission. Therefore, memory estimates became a regular part of preliminary design studies. Most estimates in the 1969 to 1971 period ranged around 28K words[46]. Rockwell International settled on 32K in its bid and won the contract partially because of that estimate[47]. NASA, trying to save itself from later difficulties, bought 64K of memory for each computer, hoping that doubling the estimate would be enough (despite memory increases in previous programs of several hundred percent)[48]. Unfortunately, the software grew to over 700K, requiring not only more computer memory, but the addition of mass memory units to hold programs that would not fit into the extended core. Parten said after this, "I don't know how you ensure proper memory size ahead of time, unless you're incredibly lucky"[49]. From the standpoint of a spacecraft designer worried about power requirements, an interesting feature of the AP-101 memory is that only the module currently being accessed is at full power. If a memory module is used, it remains at full power for 20 microseconds. If no

further accesses are made in that interval, it automatically goes to medium power. If the entire computer is in standby mode, it goes to low power. An estimated 136 watts are saved by doing this switching[50].

The memory can be altered in flight. The ground can uplink bursts of 64 16-bit halfwords at a time, which can replace data already in the specified addresses. The crew can also change up to six 32-bit words simultaneously by using their displays and keyboards. However, those changes must be hand keyed in hexadecimal.

The Shuttle's AP-101 contains one of the most extensive sets of self-testing hardware and software ever used in a flight computer. Its self-test hardware resides in the BITE, or built-in test equipment. When this is coupled with the self-test software, 95% of hardware failures can be detected by the machine itself[51], whereas the other 5% and potential software failures require the use of redundancy.

As evidenced by the component description given here, the IBM AP-101 is a fairly common computer architecture, easily understandable and programmable by anyone familiar with IBM's large commercial mainframes. The IOPS, bus system, and displays contain the characteristics that make the Shuttle DPS unique.

The IOPs and the Bus System

It is difficult to discuss the Shuttle's IOPs without also talking about the data bus network, because the former are designed to manage the latter. All subsystems on the spacecraft are connected redundantly to at least a pair of data buses. There are 24 of these buses, and the subsystems share them, using multiplexers to control the sharing. Eight of the 24 are "flight-critical data buses" that help fly the vehicle; 5 are used for intercomputer communication among the five general-purpose computers; 4 connect to the four display units; 2 run to the twin mass memory units; 2 more are "launch data buses," and connect to the Launch Processing System; 2 are used for payloads, and the final pair for instrumentation[52]. Each bus is individually controlled by a microprogrammed processor, essentially a small special-purpose computer, called a bus control element (BCE). The BCE can access memory and execute independent programs[53]. A twenty-fifth computer, the Master Sequence Controller, is used to control I/O flow on the 24 BCEs[54]. Thus, each IOP contains 25 dedicated computers. In addition, the IOP itself is basically a programmable processor with multiple functions. It shares main memory with the central processor. If a program affecting the IOP is initiated by the central processor, a direct memory access channel is opened to speed up reading core. That, however, creates contention for the memory with the central processor

and may have the effect of actually slowing down the system as a whole[55].

One reason an IOP is needed is that the Shuttle computers transfer data internally in parallel along 18-bit buses. This means that one half word and its associated parity bit are moved from memory to the operation registers and back again all at once. However, data are transferred from orbiter subsystems to the IOP in serial form, one bit at a time. Of course, the serial data are at a high rate (1 megahertz), so transfer speed is not a concern. The conversion of serial data to parallel data is the function of the Multiplexer Interface Adapters in the IOP[56]. The Shuttle DPS also has 16 multiplexer/demultiplexers that convert parallel data to serial for output to the buses[57].

Input and output to each computer is ultimately controlled in two modes: command and listen. In command mode (CM), signals sent from the host processor to subsystems connected to a bus controlled by a commanding BCE will actually effect the commands. In listen mode, the subsystems will ignore the command signals. In both cases input to the computer from any bus is listened to, but the computer's orders are obeyed only by the systems on the buses for which it is the commander. This moding capability means that a single computer can be assigned a set of buses different from another computer, thus spreading out the responsibilities and protecting against failure. It also means that each computer receives all input data all the time, so that it can take over from a failed computer immediately. This is especially important to the backup flight system. The set of controlled buses is called a "string." A typical string for a single computer might be a pair of flight critical buses, one intercomputer bus (always), a display bus, and a bus from the mass memory unit (MMU), payload, launch, and instrumentation group. The strings can be reconfigured by the crew in flight, which is done periodically as missions proceed through various phases.

Display Electronics

The Shuttle's display system, built by the Norden Division of United Technologies Corporation, is the most complex ever used on a flying machine and contains computers of its own. For the first time in a spacecraft, cathode ray tubes (CRTs) are used as the primary display medium, although a wealth of warning lights that supplement the displays still dot the cockpit. The CRTs hold 26 lines of 51 characters on a 5- by 7-inch screen. That screen size is fairly common on portable computers. However, the number of characters per line is smaller (51 vs. the more common 80) and the number of lines larger (26 vs. the usual 24). The net effect is that the individual characters appear slightly larger on the Shuttle's screens, necessary because although the

user of a portable computer is usually about 16 inches from the screen, on the Shuttle the distance between user and screen is well over 2 feet. Information on the Shuttle's screens appears green on black, and characters can be selectively highlighted. Three of these screens are mounted in the forward cockpit between the pilots. A fourth is aft at the mission specialist station. Keyboards, built by Ebonex, are used for crew input. Two are between the pilots, with a third adjacent to the mission specialist's CRT.

Displays placed on the CRTs are controlled by a special-purpose computer with a 16-bit word size and 8K of memory. This computer provides display control and can create circles, lines, intensity changes (highlighting), and flashing messages. The display software is stored on the MMUs until the computer is powered up. The CRT and its associated processor is referred to as the display electronics unit (DEU)[58].

Mass Memory Unit: A Late Addition

The final component of the Shuttle's DPS hardware is the mass memory unit (MMU). Originally acquired only to provide initial loading of the orbiter's computers, the MMU, built by Odetics, Inc., has been used extensively to help resolve the memory growth problem. Two of these units are installed on the orbiter, each capable of containing 8 million 16-bit words, enough for three times the Shuttle software. The tape can be addressed in 512 word blocks, and the crew can alter its contents in flight using a special display[59]. The MMU stores all the Primary Avionics Software System and all the software for the Backup Flight System, the DEUs, and the engine controllers. Thus, the Shuttle continues the same computer/mass memory configuration as the Gemini spacecraft.

This complex network of computer hardware on the orbiter has many possible points of failure. Also, the 700K of flight software may contain undiscovered bugs that could emerge at critical mission times, and self-testing might not be sufficient to protect the spacecraft from such failures. Other schemes for preventing a fatal failure need to be developed if the Shuttle is to fly with the confidence of its crew, passengers, and potential paying customers. Exactly what those schemes would be has occupied many researchers for several years.

COMPUTER SYNCHRONIZATION AND REDUNDANCY MANAGEMENT

One key goal shaping the design of the Shuttle was "autonomy." Multiple missions might be in space at the same time, and large crews, many with nonpilot passengers, were to travel in space in craft much more self-sufficient than ever before. These circumstances, the desire for swift turnaround time between launches, and the need to sustain mission success through several levels of component failure meant that the Shuttle had to incorporate a large measure of fault tolerance in its design. As a result, NASA could do what would have been unthinkable 20 years earlier: put men on the Shuttle's *first* test flight. The key factor in enabling NASA to take such a risk was the redundancy built into the orbiter[60].

Fault tolerance on the Shuttle is achieved through a combination of redundancy and backup. Its five general-purpose computers have reliability through redundancy, rather than the expensive quality control employed in the Apollo program[61]. Four of the computers, each loaded with identical software, operate in what is termed the "redundant set" during critical mission phases such as ascent and descent. The fifth, since it only contains software to accomplish a "no frills" ascent and descent, is a backup. The four actuators that drive the hydraulics at each of the aerodynamic surfaces are also redundant, as are the pairs of computers that control each of the three main engines.

Management of redundancy raised several difficult questions. How are failures detected and certified? Should the system be static or dynamic? Should the computers run separately without communication and be used to replace the primary computer one by one as failures occur? Could the computers, if running together, stay in step? Should redundancy management of the actuators be at the computer or subsystem level? Fortunately, NASA experience on other aircraft and spacecraft programs could provide data for making the final decisions.

Redundant Precursors

Several systems that incorporated redundancy preceded the Shuttle. The computer used in the Saturn booster instrument unit that contained the rocket's guidance system used triple modular redundant (TMR) circuits, which means that there was one computer with redundant components. Disadvantages to using such circuits in larger computers are that they are expensive to produce, and an event such as the

explosion on Apollo 13 could damage enough of the computer that it ceases to function. By spreading redundancy among several simplex circuit computers scattered in various parts of the spacecraft, the effects of such catastrophic failures are minimized[62].

Skylab's two computers each could perform all the functions required on its mission. If one failed, the other would automatically take over, but both computers were not up and running simultaneously. The computer taking over would have to find out where the other had left off by using the contents of the 64-bit transfer register located in the common section built with TMR circuits. The Skylab computers were able to have such a relatively leisurely switch-over system because they were not responsible for navigation or high-frequency flight control functions. If there were a failure, it would be possible for the Skylab to drift in its attitude without serious danger; the Shuttle would have no such margin of safety.

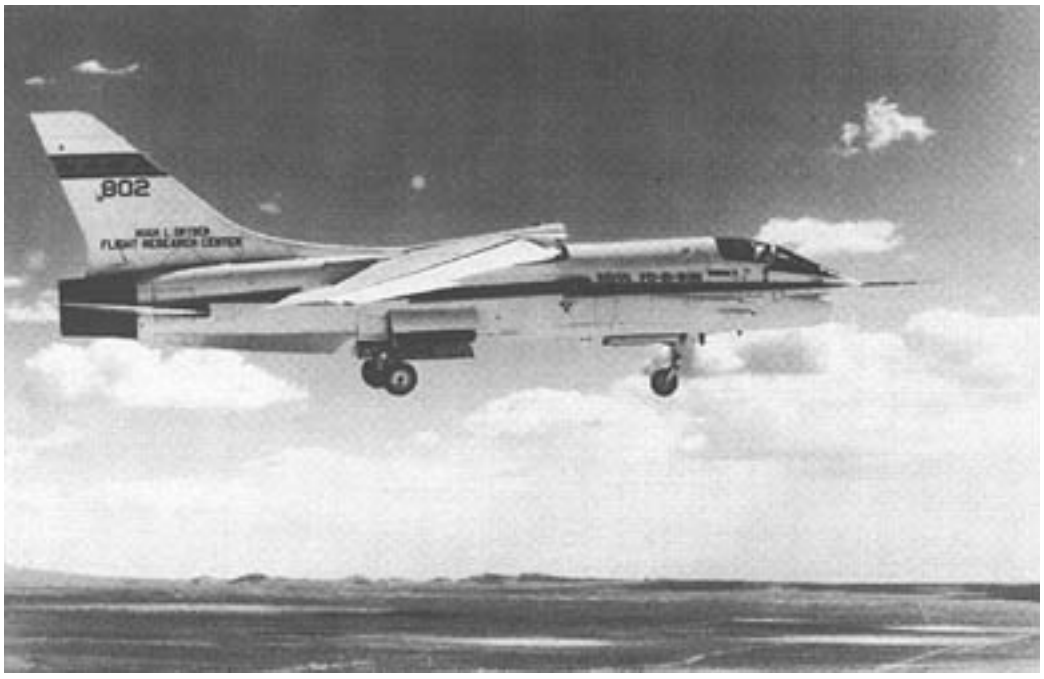

Figure 4–3. The F-8 aircraft that proved the redundant set configuration planned for the Shuttle would work. (NASA photo ECN-6988)

The need for the redundant computers on the Shuttle to process information simultaneously, while still staying closely synchronized for rapid switch-over, seriously challenged the designers of the system. Such a close synchronization between computers had not been done before, and its feasibility would have to be proven before NASA could make a full commitment to a particular design. Most of the necessary

confidence resulted from a digital fly-by-wire testing program NASA started at the Dryden Flight Research Center in the early 1970s[63]. The first computer used in the F-8 "Crusader" aircraft chosen for the program was a surplus AGC in simplex, with an electronic analog backup. Later, the project engineers wanted a duplex system using a more advanced computer. Johnson Space Center avionics people noted the similarities between the digital fly-by-wire program and the Shuttle. Dr. Kenneth Cox of JSC suggested that Dryden go with a triplex system to move beyond simple one-for-one redundancy. By coordinating procurement, NASA outfitted both the F-8 aircraft and the Shuttle with AP-101 processors. Draper Laboratory produced software for the F-8, and its flight tests proved the feasibility of computers operating in synchronization, as it suffered several single point computer failures but successfully flew on without loss of control. This flight program did much to convince NASA of the viability of the synchronization and redundancy management schemes developed for the Shuttle.

How Many Computers?

One key question in redundancy planning is how many computers are required to achieve the level of safety desired. Using the concept of fail operational/fail operational/fail-safe, five computers are needed. If one fails, normal operations are still maintained. Two failures result in a fail-safe situation, since the three remaining prevent the feared standoff possible in dual computer systems (one is wrong, but which?). Due to cost considerations of both equipment and time, NASA decided to lower the requirement to fail operational/fail-safe, which allowed the number of computers to be reduced to four. Since five were already procured and designed into the system, the fifth computer evolved into a backup system, providing reduced but adequate functions for both ascent and descent in a single memory load. NASA's decision to use four computers has a basis in reliability projections done for fly-by-wire aircraft. Triplex computer system failures were expected to cause loss of aircraft three times in a million flights, whereas quadruple computer system failures would cause loss of aircraft only four times in a *thousand million* flights[64].

At first the backup flight system computer was not considered to be a permanent fixture. When safety level requirements were lowered, some IBM and NASA people expected the fifth computer to be removed after the Approach and Landing Test phase of the Shuttle program and certainly after the flight test phase (STS-1 through 4)[65]. However, the utility of the backup system as insurance against a generic software error in the primary system outweighed considerations of the savings in weight, power, and complexity to be made by eliminating it[66]. In

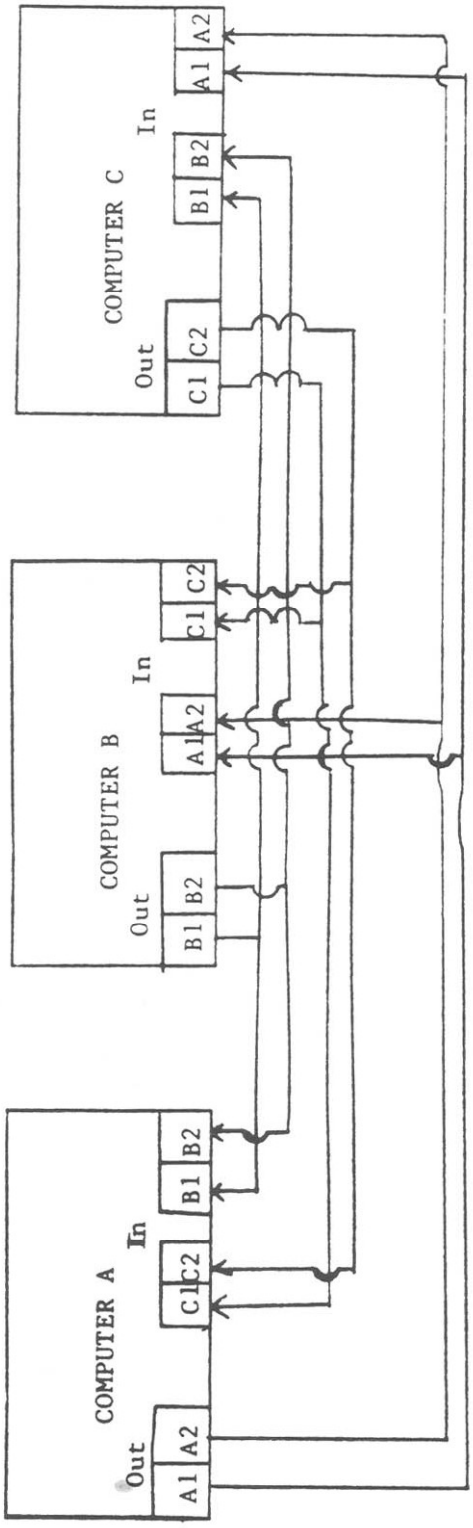

Figure 4–4. The intercommunication system used in the F-8 triplex computer system.

fact, as the first Shuttle flights approached, Arnold Aldrich, Director of the Shuttle Office at Johnson Space Center, circulated a memo arguing for a sixth computer to be carried along as a spare[67]! He pointed out that since 90% of avionics component failures were expected to be computer failures and that since a minimum of three computers and the backup should exist for a nominal re-entry, aborts would then have to take place after one failure. By carrying a spare computer preloaded with the entry software, the primary system could be brought back to full strength. The sixth computer was indeed carried on the first few flights. In contrast with this "suspenders and belt" approach, John R. Garman of the Johnson Space Center Spacecraft Software Division said that "we probably did more damage to the system as a whole by putting in the backup"[68]. He felt that the institution of the backup took much of the pressure off the developers of the primary system. No longer was their software solely responsible for survival of the crew. Also, integrating the priority-interrupt-driven operating system of the primary computers with the time-slice system of the backup caused compromises to be made in the primary.

Synchronization

Computer synchronization proved to be the most difficult task in producing the Shuttle's avionics. Synchronizing redundant computers and comparing their current states is the best way to decide if a failure has occurred. There are two types of synchronization used by the Shuttle's computers in determining which of them has failed: one for the redundant set of computers established for ascent to orbit and descent from orbit, and one for synchronizing a common set while in orbit. It took several years in the early 1970s to discover a way to accomplish these two synchronizations.

The essence of Shuttle redundancy is that each computer in the redundant set could do all the functions necessary at a particular mission phase. For true redundancy to take place, all computers must listen to all traffic on all buses, even though they might be commanding just a few. That way they know about all the data generated in the current phase. They must also be processing that data at the same time the other computers do. If there is a failure, then the failed computer could drop out of the set without any functional degradation whatever. At the start, the Shuttle's designers thought it would be possible to run the redundant computers separately and then just compare answers periodically to make sure that the data and calculations matched[69]. As it turned out, small differences in the oscillators that acted as clocks within the computers caused the computers to get out of step fairly quickly. The Spacecraft Software Division formed a committee, headed

by Garman, made up of representatives from Johnson Space Center, Rockwell International, Draper Laboratory and IBM Corporation, to study the problem caused by oscillator drift[70]. Draper's people made the suggestion that the computers be synchronized at input and output points[71].This concept was later expanded to also place synchronization points at process changes, when the system makes a transition from one software module to another. The decision to put in the synchronization points "settled everyone's mind" on the issue[72].

Intercomputer communication is what makes the Shuttle's avionics system uniquely advanced over other forms of parallel computing. The software required for redundancy management uses just 3K of memory and around 5% or 6% of each central processor's resources, which is a good trade for the results obtained[78]. An increasing need for redundancy and fault tolerance in non-avionics systems such as banks, using automatic tellers and nationwide computer networks, proves the usefulness of this system. But this type of synchronization is so little known or understood by people outside the Shuttle program that carryover applications will be delayed.

One reason why the redundancy management software was able to be kept to a minimum is that NASA decided to move voting to the actuators, rather than to do it before commands are sent on buses. Each actuator is quadruple redundant. If a single computer fails, it continues to send commands to an actuator until the crew takes it out of the redundant set. Since the Shuttle's other three computers are sending apparently correct commands to their actuators, the failed computer's commands are physically out-voted[79]. Theoretically, the only serious possibility is that three computers would fail simultaneously, thus negating the effects of the voting. If that occurs, and if the proper warnings are given, the crew can then engage the backup system simply by pressing a button located on each of the forward rotational hand controllers.

Does the redundant set synchronization work? As described, the F-8 version, with redundancy management identical to the Shuttle, survived several in-flight computer failures without mishap. On the first Shuttle Approach and Landing Test flight, a computer failed just as the *Enterprise* was released from the Boeing 747 carrier; yet the landing was still successful. That incident did a lot to convince the astronaut pilots of the viability of the concept.

Synchronization and redundancy together were the methods chosen to ensure the reliability of the Shuttle avionics hardware. With the key hardware problems solved, NASA turned to the task of specifying the most complex flight software ever conceived.

Box 4-2: Redundant Set Synchronization: Key to Reliability

Synchronization of the redundant set works like this: When the software accepts an input, delivers an output, or branches to a new process, it sends a 3-bit discrete signal on the intercomputer communication (ICC) buses, then waits up to 4 milliseconds for similar discretes from the other computers to arrive. The discretes are coded for certain messages. For example, 010 means an I/O is complete without error, but 011 means that an I/O is complete with error[73]. This allows more information other than just "here I am" to be sent. If another computer either sends the wrong synchronization code, or is late the computer detecting either of these conditions concludes that the delinquent computer has failed, and refuses from then on to listen to it or acknowledge its presence. Under normal circumstances, all three good computers should have detected the single computer's error. The bad computer is announced to the crew with warning lights, audio signals, and CRT messages. The crew must purposely kill the power to the failed computer, as there is no provision for automatic powerdown. This prevents a generic software failure causing all the computers to be automatically shut off.

This form of synchronization creates a tightly coupled group of computers constantly certifying that they are at the same place in the software. To certify that they are achieving the same solutions, a "sumword" is used. While computers are in a redundant set, a sumword is exchanged 6.25 times every second on the ICC buses[74]. A sumword typically consists of a 64 bits of data, usually the least significant bits of the last outputs to the solid rocket boosters, orbital maneuvering engines, main engines, body flap, speed brake, rudder, elevons, throttle, the system discretes, and the reaction control system[75]. If there are three straight miscomparisons of a sumword, the detecting computers declare the computer involved to be failed[76].

Both the 3-bit synchronization code and sumword comparison are characteristics of the redundant set operations. During noncritical mission phases such as on-orbit, the computers are reconfigured. Two might be left in the redundant set to handle guidance and navigation functions, such as maintaining the state vector. A third would run the systems management software that controls life support, power, and the payload. The fourth would be loaded with the descent software and powered down, or "freeze dried," to be instantly ready to descend in an emergency and to protect against a failure of the two MMUs. The fifth contains the backup flight system. This configuration of computers is not tightly coupled, as in the redundant set. All active computers, however, do continue the 6.25/second exchange of sumwords, called the common set synchronization[77].

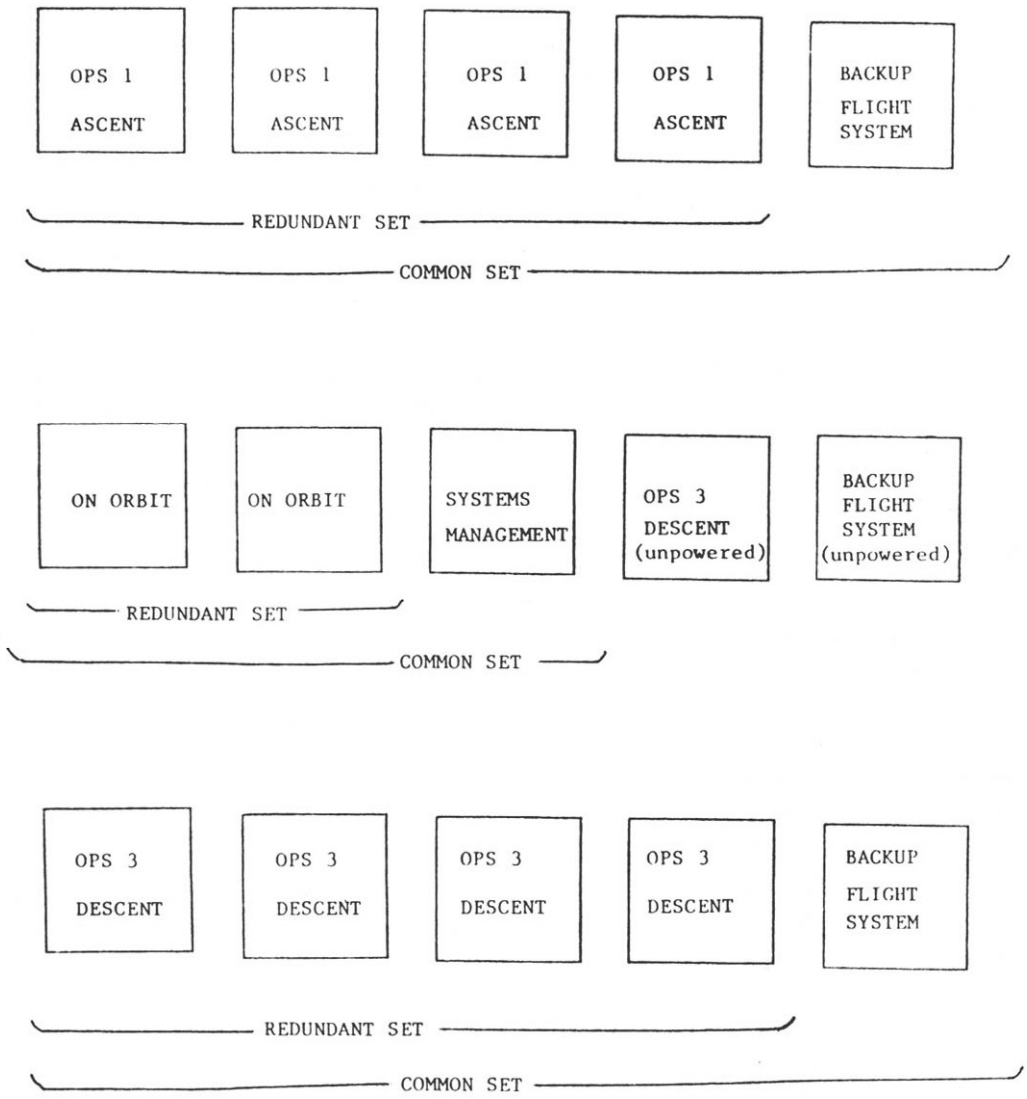

Figure 4–5. The various computer configurations used during a Shuttle mission. The names of the operational sequences loaded into the machines are shown.

DEVELOPING SOFTWARE FOR THE SPACE SHUTTLE

During 1973 and 1974 the first requirements began to be specified for what has become one of the most interesting software systems ever designed. It was obvious from the very beginning that developing the Shuttle's software would be a complicated job. Even though NASA engineers estimated the size of the flight software to be smaller than that on Apollo, the ubiquitous functions of the Shuttle computers meant that no one group of engineers and no one company could do the software on its own. This increased the size of the task because of the communication necessary between the working groups. It also increased the complexity of a spacecraft already made complex by flight requirements and redundancy. Besides these realities, no one could foresee the final form that the software for this pioneering vehicle would take, even after years of development work had elapsed, since there continued to be both minor and major changes. NASA and its contractors made over 2,000 requirements changes between 1975 and the first flight in 1981[80]. As a result, about $200 million was spent on software, as opposed to an initial estimate of $20 million. Even so, NASA lessened the difficulties by making several early decisions that were crucial for the program's success. NASA separated the software contract from the hardware contract, closely managed the contractors and their methods, chose a high-level language, and maintained conceptual integrity.

NASA awarded IBM Corporation the first independent Shuttle software contract on March 10, 1973. IBM and Rockwell International had worked together during the period of competition for the orbiter contract[81]. Rockwell bid on the entire aerospacecraft, intending to subcontract the computer hardware and software to IBM. But to Rockwell's dismay, NASA decided to separate the software contract from the orbiter contract. As a result, Rockwell still subcontracted with IBM for the computers, but IBM had a separate software contract monitored closely by the Spacecraft Software Division of the Johnson Space Center. There are several reasons why this division of labor occurred. Since software does not weigh anything in and of itself, it is used to overcome hardware problems that would require extra systems and components (such as a mechanical control system)[82]. Thus software is in many ways the most critical component of the Shuttle, as it ties the other components together. Its importance to the overall program alone justified a separate contract, since it made the contractor directly accountable to NASA. Moreover, during the operations phase, software underwent the most changes, the hardware being essentially fixed[83]. As time went on, 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, not Rockwell, won the competition for the operations contract. By keeping the software contract separate, NASA could develop the code on a continuing basis. There is a considerable difference between changing maintenance mechanics on an existing hardware system and changing software companies on a not yet perfect system because to date the relationships between components in software are much harder to define than those in hardware. Personnel experienced with a specific software system are the best people to maintain it. Lastly, Christopher Kraft of Johnson Space Center and George Low of NASA Headquarters, both highly influential in the manned spacecraft program during the early 1970's, felt that Johnson had the software management expertise to handle the contract directly[84].

One of the lessons learned from monitoring Draper Laboratory in the Apollo era was that by having the software development at a remote site (like Cambridge), the synergism of informally exchanged ideas is lost; sometimes it took 3 to 4 weeks for new concepts to filter over[85]. IBM had a building and several hundred personnel near Johnson because of its Mission Control Center contracts. When IBM won the Shuttle contract, it simply increased its local force.

The closeness of IBM to Johnson Space Center also facilitated the ability of NASA to manage the project. The first chief of the Shuttle's software, Richard Parten, observed that the experience of NASA managers made a significant contribution to the success of the programming effort[86]. Although IBM was a giant in the data processing industry, a pioneer in real time systems, and capable of putting very bright people on a project, the company had little direct experience with avionics software. As a consequence, Rockwell had to supply a lot of information relating to flight control. Conversely, even though Rockwell projects used computers, software development on the scale needed for the Shuttle was outside its experience. NASA Shuttle managers provided the initial requirements for the software and facilitated the exchange of information between the principal contractors. This situation was similar to that during the 1960s when NASA had the best rendezvous calculations people in the world and had to contribute that expertise to IBM during the Gemini software development. Furthermore, the lessons of Apollo inspired the NASA managers to push IBM for quality at every point[87].

The choice of a high level language for doing the majority of the coding was important because, as Parten noted, with all the changes, "we'd still be trying to get the thing off the ground if we'd used assembly language"[88]. Programs written in high level languages are far easier to modify. Most of the operating system software, which is rarely changed, is in assembler, but all applications software and some of the interfaces and redundancy management code is in HAL/S[89].

Although the decision to program in a high-level language meant that a large amount of support software and development tools had to be written, the high-level language nonetheless proved advantageous, especially since it has specific statements created for real-time programming.

Defining the Shuttle Software

In the end, the success of the Shuttle's software development was due to the conceptual integrity established by using rigorously maintained requirements documents. The requirements phase is the beginning of the software life cycle, when the actual functions, goals, and user interfaces of the eventual software are determined in full detail. If a team of a thousand workers was asked to set software requirements, chaos would result[90]. On the other hand, if few do the requirements but many can alter them later, then chaos would reign again. The strategy of using a few minds to create the software architecture and interfaces and then ensuring that their ideas and theirs alone are implemented, is termed "maintaining conceptual integrity," which is well explained in Frederick C. Brooks' *The Mythical Man Month*[91]. As for other possible solutions, Parten says, "the only right answer is the one you pick and make to work"[92].

Shuttle requirements documents were arranged in three Levels: A, B, and C, the first two written by Johnson Space Center engineers. John R. Garman prepared the Level A document, which is comprised of a comprehensive description of the operating system, applications programs, keyboards, displays, and other components of the software system and its interfaces. William Sullivan wrote the guidance, navigation and control requirements, and John Aaron, the system management and payload specifications of Level B. They were assisted by James Broadfoot and Robert Ernull[93]. Level B requirements are different in that they are more detailed in terms of what functions are executed when and what parameters are needed[94]. The Level Bs also define what information is to be kept in COMPOOLS, which are HAL/S structures for maintaining common data accessed by different tasks[95]. The Level C requirements were more of a design document, forming a set with Level B requirements, since each end item at Level C must be traceable to a Level B requirement[96]. Rockwell International was responsible for the development of the Level C requirements as, technically, this is where the contractors take over from the customer, NASA, in developing the software.

Early in the program, however, Draper Laboratory had significant influence on the software and hardware systems for the Shuttle. Draper was retained as a consultant by NASA and contributed two key items

to the software development process. The first was a document that "taught" NASA and other contractors how to write requirements for software, how to develop test plans, and how to use functional flow diagrams, among other tools[97]. It seems ironic that Draper was instructing NASA and IBM on such things considering its difficulties in the mid-1960s with the development of the Apollo flight software. It was likely those difficult experiences that helped motivate the MIT engineers to seriously study software techniques and to become, within a very short time, one of the leading centers of software engineering theory. The Draper tutorial included the concept of highly modular software, software that could be "plugged into" the main circuits of the Shuttle. This concept, an application of the idea of interchangeable parts to software, is used in many software systems today, one example being the UNIX*** operating system developed at Bell Laboratories in the 1970s, under which single function software tools can be combined to perform a large variety of functions.

Draper's second contribution was the actual writing of some early Level C requirements as a model[98]. This version of the Level C documents contained the same components as in the later versions delivered by Rockwell to IBM for coding. Rockwell's editions, however, were much more detailed and complete, reflecting their practical, rather than theoretical purpose and have been an irritation for IBM. IBM and NASA managers suspect that Rockwell, miffed when the software contract was taken away from them, may have delivered incredibly precise and detailed specifications to the software contractor. These include descriptions of flight events for each major portion of the software, a structure chart of tasks to be done by the software during that major segment, a functional data flowchart, and, for each module, its name, calculations, and operations to be performed, and input and output lists of parameters, the latter already named and accompanied by a short definition, source, precision, and what units each are in. This is why one NASA manager said that "you can't see the forest for the trees" in Level C, oriented as it is to the production of individual modules[99]. One IBM engineer claimed that Rockwell went "way too far" in the Level C documents, that they told IBM too much about *how* to do things rather than just *what* to do[100]. He further claimed that the early portion of the Shuttle development was "underengineered" and that Rockwell and Draper included some requirements that were not passed on by NASA. Parten, though, said that all requirements documents were subject to regular review by joint teams from NASA and Rockwell[101].

The impression one gains from documents and interviews is that both Rockwell and IBM fell victim to the "not invented here" syndrome: If we didn't do it, it wasn't done right. For example,

***UNIX is a trademark of AT&T.

Rockwell delivered the ascent requirements, and IBM coded them to the letter, thereby exceeding the available memory by two and a third times and demonstrating that the requirements for ascent were excessive. Rockwell, in return, argued for 2 years about the nature of the operating system, calling for a strict time-sliced system, which 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 one. The system thus cycles through all scheduled tasks in a fixed period of time, working on each in turn. Rockwell's original proposal was for a 40-millisecond cycle with synchronization points at the end of each[102]. IBM, at NASA's urging, countered with a priority-interrupt-driven system similar to the one on Apollo. Rockwell, experienced with time-slice systems, fought this from 1973 to 1975, convinced it would never work[103].

The requirements specifications for the Shuttle eventually contained in their three levels what is in both the specification and design stage of the software life cycle. In this sense, they represent a fairly complete picture of the software at an early date. This level of detail at least permitted NASA and its contractors to have a starting point in the development process. IBM constantly points to the number of changes and alterations as a continuing problem, partially ameliorated by implementing the most mature requirements first[104]. Without the attempt to provide detail at an early date, IBM would not have had any mature requirements when the time came to code. Even now, requirements are being changed to reflect the actual software, so they continue to be in a process of maturation. But early development of specifications became the means by which NASA could enforce conceptual integrity in the shuttle software.

Architecture of the Primary Avionics Software System

The Primary Avionics Software System, or PASS, is the software that runs in all the Shuttle's four primary computers. PASS is divided into two parts: system software and applications software. The system software is the Flight Computer Operating System (FCOS), the user interface programming, and the system control programs, whereas the applications software is divided into guidance, navigation and control, orbiter systems management, payload and checkout programs. Further divisions are explained in Box 4-3.

Box 4-3: Structure of PASS Applications Software

The PASS guidance and navigation software is divided into major functions, dictated by mission phases, the most obvious of which are preflight, ascent, on-orbit, and descent. The requirements state that these major functions be called OPS, or operational sequences. (e.g., OPS-1 is ascent; OPS-3, descent.) Within the OPS are major modes. In OPS-1, the first-stage burn, second-stage burn, first orbital insertion burn, second orbital insertion burn, and the initial on-orbit coast are major modes; transition between major modes is automatic. Since the total mission software exceeds the capacity of the memory, OPS transitions are normally initiated by the crew and require the OPS to be loaded from the MMU. This caused considerable management concern over the preservation of data, such as the state vector, needed in more than one OPS[105]. NASA's solution is to keep common data in a major function base, which resides in memory continuously and is not overlaid by new OPS being read into the computers.

Within each OPS, there are special functions (SPECs) and display functions (DISPs). These are available to the crew as a supplement to the functions being performed by the current OPS. For example, the descent software incorporates a SPEC display showing the horizontal situation as a supplement to the OPS display showing the vertical situation. This SPEC is obviously not available in the on-orbit OPS. A DISP for the on-orbit OPS may show fuel cell output levels, fuel reserves in the orbital maneuvering system, and other such information. SPECs usually contain items that can be selected by the crew for execution. DISPs are just what their name means, displays and not action items. Since SPECs and DISPs have lower priority than OPS, when a big OPS is in memory they have to be kept on the tape and rolled in when requested[106]. The actual format of the SPECs, DISPs, OPS displays, and the software that interprets crew entries on the keyboard is in the user interface portion of the system software.

The most critical part of the system software is the FCOS. NASA, Rockwell, and IBM solved most of the grand conceptual problems, such as the nature of the operating system and the redundancy management scheme, by 1975. The first task was to convert the FCOS from the proposed 40-millisecond loop operating system to a priority-driven [113] system[107]. Priority interrupt systems are superior to time-slice systems because they degrade gracefully when overloaded[108]. In a time-slice system, if the tasks scheduled in the current cycle get bogged down by excessive I/O operations, they tend to slow down the total time of execution of processes. IBM's version of the FCOS actually has cycles, but they are similar to the ones in the Skylab system described in the previous chapter. The minor cycle is the high-frequency cycle; tasks within it are scheduled every 40 milliseconds. Typical tasks in this cycle are those related to active flight control in the atmosphere. The major cycle is 960 milliseconds, and many monitoring and system management tasks are scheduled at that frequency[109]. If a process is still running when its time to restart comes up due to excessive I/O or

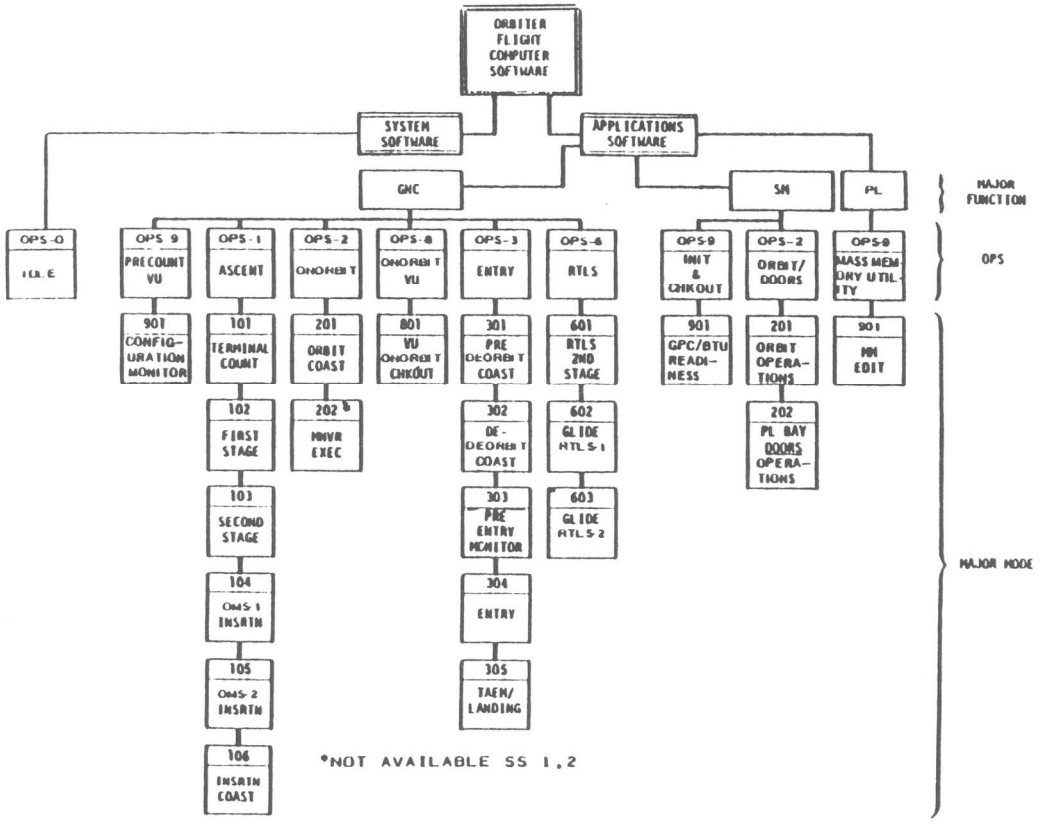

Figure 4–6. A block diagram of the Shuttle flight computer software architecture. (From NASA, *Data Processing System Workbook*)

because it was interrupted, it cancels its next cycle and finishes up[110]. If a higher priority process is called when another process is running, then the current process is interrupted and a program status word (PSW) containing such items as the address of the next instruction to be executed is stored until the interruption is satisfied. The last instruction of an interrupt is to restore the old PSW as the current PSW so that the interrupted process can continue[111]. The ability to cancel processes and to interrupt them asynchronously provides flexibility that a strict time-slice system does not.

A key requirement of the FCOS is to handle the real-time statements in the HAL/S language. The most important of these are SCHEDULE, which establishes and controls the frequency of execution of processes; TERMINATE and CANCEL, which are the opposite of SCHEDULE; and WAIT, which conditionally suspends execution[112]. The method of implementing these statements is controlled by a separate interface control document[113]. SCHEDULE is

generally programmed at the beginning of each operational sequence to set up which tasks are to be done in that software segment and how often they are to be done. The syntax of SCHEDULE permits the programmer to assign a frequency and priority to each task. TERMINATE and CANCEL are used at the end of software phases or to stop an unneeded process while others continue. For example, after the solid rocket boosters burn out and separate, tasks monitoring them can cease while tasks monitoring the main engines continue to run. WAIT, although handy, is avoided by IBM because of the possibility of the software being "hung up" while waiting for the I/O or other condition required to continue the process[114]. This is called a race condition or "deadly embrace" and is the bane of all shared resource computer operating systems.

The FCOS and displays occupy 35K of memory at all times[115]. Add the major function base and other resident items, and about 60K of the 106K of core remains available for the applications programs. Of the required applications programs, ascent and descent proved the most troublesome. Fully 75% of the software effort went into those two programs[116]. After the first attempts at preparing the ascent software resulted in a 140K load, serious code reduction began. By 1978, IBM reduced the size of the ascent program to 116K, but NASA Headquarters demanded it be further knocked down to 80K[117]. The lowest it ever got was 98,840 words (including the system software), but its size has since crept back up to nearly the full capacity of the memory. IBM accomplished the reduction by moving functions that could wait until later operational sequences[118]. The actual figures for the test flight series programs are in Table 4-1[119]. The total size of the flight test software was 500,000 words of code. Producing it and modifying it for later missions required the development of a complete production facility.

Table 4-1: Sizes of Software Loads in PASS.

NAME	K WORDS
Preflight initialization	72.4
Preflight checkout	81.4
Ascent and abort	105.2
On-orbit	83.1
On-orbit checkout	80.3
On-orbit system management	84.1
Entry	101.1
Mass memory utility	70.1

Note: Payload and rendezvous software was added later during the operations phase.

Implementing PASS

NASA planned that PASS would be a continuing development process. After the first flight programs were produced, new functions needed to be added and adapted to changing payload and mission requirements. For instance, over 50% of PASS modules changed during the first 12 flights in response to requested enhancements[120]. To do this work, NASA established a Software Development Laboratory at Johnson Space Center in 1972 to prepare for the implementation of the Shuttle programs and to make the software tools needed for efficient coding and maintenance. The Laboratory evolved into the Software Production Facility (SPF) in which the software development is carried on in the operations era. Both the facilities were equipped and managed by NASA but used largely by contractors.

The concept of a facility dedicated to the production of onboard software surfaced in a Rand Corporation memo in early 1970[121]. The memo summarized a study of software requirements for Air Force space missions during the decade of the 1970s. One reason for a government-owned and operated software factory was that it would be easier to establish and maintain security. Most modules developed for the Shuttle, such as the general flight control software and memory

displays, would be unclassified. However, Department of Defense (DoD) payloads require system management and payload management software, plus occasional special maneuvering modules. These were expected to be classified. Also, if the software maintenance contract moved from the original prime contractor to some different operations contractor, it would be considerably simpler to accomplish the transfer if the software library and development computers were government owned and on government property. Lastly, having such close control over existing software and new development would eliminate some of the problems in communication, verification, and maintenance encountered in the three previous manned programs.

Developing the SPF turned out to be as large a task as developing the flight software itself. During the mid-1970s, IBM had as many people doing software for the development lab as they had working on PASS[122]. The ultimate purpose of the facility is to provide a programming team with sufficient tools to prepare a software load for a flight. This software load is what is put on to the MMU tape that is flown on the spacecraft. In the operations era of the 1980s, over 1,000 compiled modules are available. These are fully tested, and often previously used, versions of tasks such as main engine throttling, memory modification, and screen displays that rarely change from flight to flight. New, mission-specific modules for payloads or rendezvous maneuvers are developed and tested using the SPF's programming tools, which themselves represent more than a million lines of code[123]. The selection of existing modules and the new modules are then combined into a flight load that is subject to further testing. NASA achieved the goal of having such an efficient software production system through an 8-year development process when the SPF was still the Laboratory.

In 1972, NASA studied what sort of equipment would be required for the facility to function properly. Large mainframe computers compatible with the AP-101 instruction set were a must. Five IBM 360/75 computers, released from Apollo support functions, were available[124]. These were the development machines until January of 1982[125]. Another requirement was for actual flight equipment on which to test developed modules. Three AP-101 computers with associated display electronics units connected to the 360s with a flight equipment interface device (FEID) especially developed for the purpose. Other needed components, such as a 6-degree-of-freedom flight simulator, were implemented in software[126]. The resulting group of equipment is capable of testing the flight software by interpreting instructions, simulating functions, and running it in the actual flight hardware[127].

In the late 1970s, NASA realized that more powerful computers were needed as the transition was made from development to operations. The 360s filled up, so NASA considered the Shuttle Mission Simulator (SMS), the Shuttle Avionics Instrumentation Lab (SAIL),

and the Shuttle Data Processing Center's computers as supplementary development sites, but this idea was rejected because they were all too busy doing their primary functions[128]. In 1981, the Facility added two new IBM 3033N computers, each with 16 million bytes of primary memory. The SPF then consisted of those mainframes, the three AP-101 computers and the interface devices for each, 20 magnetic tape drives, six line printers, 66 million bytes of drum memory, 23.4 billion bytes of disk memory, and 105 terminals[129]. NASA accomplished rehosting the development software to the 3033s from the 360s during the last quarter of 1981. Even this very large computer center was not enough. Plans at the time projected on-line primary memory to grow to 100 million bytes[130], disk storage to 160 billion bytes[131], and two more interface units, display units, and AP-101s to handle the growing DOD business[132]. Additionally, terminals connected directly to the SPF are in Cambridge, Massachusetts, and at Goddard Space Flight Center, Marshall Space Flight Center, Kennedy Space Center, and Rockwell International in Downey, California[133].

Future plans for the SPF included incorporating backup system software development, then done at Rockwell, and introducing more automation. NASA managers who experienced both Apollo and the Shuttle realize that the operations software preparation is not enough to keep the brightest minds sufficiently occupied. Only a new project can do that. Therefore, the challenge facing NASA is to automate the SPF, use more existing modules, and free people to work on other tasks. Unfortunately, the Shuttle software still has bugs, some of which are no fault of the flight software developers, but rather because all the tools used in the SPF are not yet mature. One example is the compiler for HAL/S. Just days before the STS-7 flight, in June, 1983, an IBM employee discovered that the latest release of the compiler had a bug in it. A quick check revealed that over 200 flight modules had been modified and recompiled using it. All of those had to be checked for errors before the flight could go. Such problems will continue until the basic flight modules and development tools are no longer constantly subject to change. In the meantime, the accuracy of the Shuttle software is dependent on the stringent testing program conducted by IBM and NASA before each flight.

Verification and Change Management of the Shuttle Software

IBM established a separate line organization for the verification of the Shuttle software. IBM's overall Shuttle manager has two managers reporting to him, one for design and development, and one for verification and field operations. The verification group has just less

than half the members of the development group and uses 35% of the software budget[134]. There are no managerial or personnel ties to the development group, so the test team can adopt an "adversary relationship" with the development team. The verifiers simply assume that the software is untested when received[135]. In addition, the test team can also attempt to prove that the requirements documents are wrong in cases where the software becomes unworkable. This enables them to act as the "conscience" of the entire project[136].

IBM began planning for the software verification while the requirements were being completed. By starting verification activity as the software took shape, the test group could plan its strategy and begin to write its own books. The verification documentation consists of test specifications and test procedures including the actual inputs to be used and the outputs expected, even to the detail of showing the content of the CRT screens at various points in the test[137]. The software for the first flight had to survive 1,020 of these tests[138]. Future flight loads could reuse many of the test cases, but the preparation of new ones is a continuing activity to adjust to changes in the software and payloads, each of which must be handled in an orderly manner.

Suggestions for changes to improve the system are unusually welcome. Anyone, astronaut, flight trainer, IBM programmer, or NASA manager, can submit a change request[139]. NASA and IBM were processing such requests at the rate of 20 per week in 1981[140]. Even as late as 1983 IBM kept 30 to 40 people on requirements analysis, or the evaluation of requests for enhancements[141]. NASA has a corresponding change evaluation board. Early in the program, it was chaired by Howard W. Tindall, the Apollo software manager, who by then was head of the Data Systems and Analysis Directorate. This turned out to be a mistake, as he had conflicting interests[142]. The change control board moved to the Shuttle program office. Due to the careful review of changes, it takes an average of 2 years for a new requirement to get implemented, tested, and into the field[143]. Generally, requests for extra functions that would push out current software due to memory restrictions are turned down[144].

Box 4-4: How IBM Verifies the Shuttle Flight Software

The Shuttle software verification process actually begins before the test group gets the software, in the sense that the development organization conducts internal code reviews and unit tests of individual modules and then integration tests of groups of modules as they are assembled into a software load. There are two levels of code inspection, or "eyeballing" the software looking for logic errors. One level of inspection is by the coders themselves and their peer reviewers. The second level is done by the outside verification team. This activity resulted in over 50% of the discrepancy reports (failures of the software to meet the specification) filed against the software, a percentage similar to the Apollo experience and reinforcing the value of the idea[145]. When the software is assembled, it is subject to the First Article Configuration Inspection (FACI), where it is reviewed as a complete unit for the first time. It then passes to the outside verification group.

Because of the nature of the software as it is delivered, the verification team concentrates on proving that it meets the customer's requirements and that it functions at an acceptable level of performance. Consistent with the concept that the software is assumed untested, the verification group can go into as much detail as time and cost allow. Primarily, the test group concentrates on single software loads, such as ascent, on-orbit, and so forth[146]. To facilitate this, it is divided into teams that specialize in the operating system and detail, or functional verification; teams that work on guidance, navigation, and control; and teams that certify system performance. These groups have access to the software in the SPF, which thus doubles as a site for both development and testing. Using tools available in the SPF, the verification teams can use the real flight computers for their tests (the preferred method). The testers can freeze the execution of software on those machines in order to check intermediate results, alter memory, and even get a log of what commands resulted in response to what inputs[147].

After the verification group has passed the software, it is given an official Configuration Inspection and turned over to NASA. At that point NASA assumes configuration control, and any changes must be approved through Agency channels. Even though NASA then has the software, IBM is not finished with it[148].

The software is usually installed in the SAIL for prelaunch, ascent, and abort simulations, the Flight Simulation Lab (FSL) in Downey for orbit, de-orbit, and entry simulations, and the SMS for crew training. Although these installations are not part of the preplanned verification process, the discrepancies noted by the users of the software in the roughly 6 months before launch help complete the testing in a real environment. Due to the nature of real-time computer systems, however, the software can *never* be fully certified, and both IBM and NASA are aware of this[149]. There are simply too many interfaces and too many opportunities for asynchronous input and output.

Discrepancy reports cause changes in software to make it match the requirements. Early in the program, the software found its way into the simulators after less verification because simulators depend

on software just to be turned on. At that time, the majority of the discrepancy reports were from the field installations. Later, the majority turned up in the SPF[150]. *All* discrepancy reports are formally disposed of, either by appropriate fixes to the software, or by waiver. Richard Parten said, "Sometimes it is better to put in an 'OPS Note' or waiver than to fix (the software). We are dependent on smart pilots"[151]. If the discrepancy is noted too close to a flight, if it requires too much expense to fix, it can be waived *if* there is no immediate danger to crew safety. Each Flight Data File carried on board lists the most important current exceptions of which the crew must be aware. By STS-7 in June of 1983, over 200 pages of such exceptions and their descriptions existed[152]. Some will never be fixed, but the majority were addressed during the Shuttle launch hiatus following the 51L accident in January 1986.

So, despite the well-planned and well-manned verification effort, software bugs exist. Part of the reason is the complexity of the real-time system, and part is because, as one IBM manager said, "we didn't do it up front enough," the "it" being thinking through the program logic and verification schemes[153]. Aware that effort expended at the early part of a project on quality would be much cheaper and simpler than trying to put quality in toward the end, IBM and NASA tried to do much more at the beginning of the Shuttle software development than in any previous effort, but it still was not enough to ensure perfection.

Box 4-5: The Nature of the Backup Flight System

The Backup Flight System consists of a single computer and a software load that contains sufficient functions to handle ascent to orbit, selected aborts during ascent, and descent from orbit to landing site. In the interest of avoiding a generic software failure, NASA kept its development separate from PASS. An engineering directorate, not the on-board software division, managed the software contract for the backup, won by Rockwell[154].

The major functional difference between PASS and the backup is that the latter uses a time-slice operating system rather than the asynchronous priority-driven system of PASS[155]. This is consistent with Rockwell's opinion on how that system was to be designed. Ironically, since the backup must listen in on PASS operations so as to be ready for instant takeover, PASS had to be modified to make it more synchronous[156]. Sixty engineers were still working on the Backup Flight System software as late as 1983[157].

USING THE SHUTTLE DPS

With the level of complexity present in the hardware and software just described, it is not surprising that the crew interfaces to those components are also complex. The complexity is caused not so much by the design of the interfaces but by the limited amount of memory available for graphics displays, automatic reconfiguring of the computers, and other utilities to make the system more cooperative and simpler for the users. There is some difference between the way the users of the system perceive the DPS and the way the designers, both NASA and IBM, perceive it. Some astronauts and trainers are openly critical. 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"[158], Flight trainer Frank Hughes also remarked that "the PASS doesn't do anything for us"[159], noting that such practical items as the time from loss of ground-station signals and acquisition of new stations is not part of the primary software. Both said, "We end up working for the computer, rather than the computer working for us." This comment is something reminiscent of Apollo days, when the number of keystrokes needed to fly a mission was a concern. John Aaron, one of NASA's designers of PASS interfaces and later chief of spacecraft software, said that the Apollo experience influenced Shuttle designers to avoid excessive pilot interaction with the computers. Even so, he found the "crew

Figure 4–7. The forward flight deck of a Shuttle, with the three CRT screens and twin keyboards visible in the center. (NASA photo S80-35133)

interfaces. . . more confusing and complex than I thought they would be"[160]. One statistic that supports his perception is that the 13,000 keystrokes used in a week-long lunar mission are matched by a Shuttle crew in a 58-hour flight[161].

Another aspect of the "working for the computer" problem is that steps normally done by computers using preprogrammed functions are done manually on the Shuttle. The reconfiguration of PASS from the ascent redundant set to the on-orbit groupings has to be done by the crew, a process taking several minutes and needing to be reversed before descent. Aaron acknowledges that the computer interfaces are too close to machine level, but points out that management "would not buy" simple automatic reconfiguration schemes. Even if they had, there is no computer memory to store such utilities.

Tied to the computer memory problem is the fact that many functions have to be displayed together on a screen because of the fact

that such displays are "memory hungry." As a result, many screens are so crowded that reading them quickly is difficult, the process being further affected by the blandness and primitive nature of any graphics available. Astronaut Vance Brand claimed that after initial confusion, several hours with simulators makes things easier to find; he makes a point of checking his entries on the input line before pressing the execute key[162]. Young does that as well, but for additional reasons: The keyboard buffer is so small that entering data too quickly causes some to be lost, and he wants to check whether he is accessing the right screen display with the proper keyboard. This latter concern arises because there are only two keyboards for the three forward CRTs. Since both keyboards can be assigned to the same screen, two CRTs may not be currently set up for input. Even if the two keyboards are assigned to different screens, one CRT is left without capability for immediate crew input. Astronaut Henry Hartsfield termed this situation "prone to error"[163].

Since flying the Shuttle is in many ways flying the Shuttle computers (they provide the active flight control, guidance and navigation, systems management, and payload functions), the astronauts are interested in making suggestions for improving the computer system. Most revolve around more automation, more user friendliness, more color, better graphics, and more functions, such as adding a return-to-launch-site (RTLS) abort with two engines out in addition to the present version with only one engine out[164]. Each of these enhancements is tied to increasing memory. IBM proposed a new version of the Shuttle computers with 256K of memory and software compatibility with the existing system. Johnson Space Center began testing these AP-101F computers in 1985, with the first operational use projected for the resumption of Shuttle missions in 1988.

In the meantime, the astronauts themselves pioneered efforts to use small computers to add functions and back up the primary systems. Early flights used a Hewlett-Packard HP-41C programmable calculator to determine ground-station availability, as well as carry a limited version of the calculations for time-to-retrofire. Beginning with STS-9 in December, 1983, a Grid Systems Compass portable microcomputer with graphics capabilities was carried to display ground stations and to provide functions impractical on the primary computers. Mission Specialist Terry Hart, responsible for programming the HP-41Cs, said that placing the mission documentation on the computer was also being considered[165].

THE SPACE SHUTTLE MAIN ENGINE CONTROLLERS

Among the many special-purpose computers on the Shuttle, the

Box 4-6: Using the Shuttle's Keyboards

The Shuttle's keyboards are different from those found on Gemini and Apollo because they are hexadecimal, or base 16, rather than decimal, so that memory locations can be altered by hex entries from the keyboard. A single hex digit represents 4 bits, so just four digits can fill a half-word memory location. The other keys perform specialized functions. The most often used are

- ITEM: This selects a specific function displayed on a CRT. For example, if the astronaut wishes to perform a function numbered 32 on the screen, he or she presses ITEM, 3, 2, EXEC.
- OPS: This, plus a four-digit number, selects the operational sequence and major mode desired by the crew. For instance, to choose the first major mode of the ascent software, OPS, 1, 1, 0, 1, and PRO is entered.
- SPEC: This key, plus appropriate digits and PRO, selects a specialist function or display function screen. Each OPS has associated with it a number of primary screens that reflect what is happening in the software. The ascent program has a vertical path graphic, for instance. Additionally, special functions can be called from SPEC displays that are overlaid on the primary screens when called. On-orbit, and several other OPS, have a "GPC Memory" display that can be used to read or write to individual memory locations. It cannot be called from either the ascent or descent OPS. Display function screens are just that: used to show various data such as fuel cell levels, but with no crew functions. To return to the primary screen that was on the CRT before the SPEC or DISP call, the RESUME key is used.
- CLEAR: Each time this key is depressed, one character is deleted from the input line on the CRT accessed. This enables an astronaut to erase an error if it is caught before EXEC or PRO is depressed.
- "+" : This sign can be used as a delimiter around numeric data or between a series of function selections.

main engine controllers stand out as a clear "first" in space technology. The Shuttle's three main liquid-propellant engines are the most complex and "hottest" rockets ever built. The complexity is tied to the mission requirements, which state that they be throttleable, a common characteristic of internal combustion engines and turbojets, but rare in the rocket business. They run "hotter" than any other rocket engine because at any given moment they are closer to destroying themselves than their predecessors. Previous engines were overbuilt in the sense that they were designed to burn at full thrust through their entire lifetime of a few minutes with no chance that the continuous

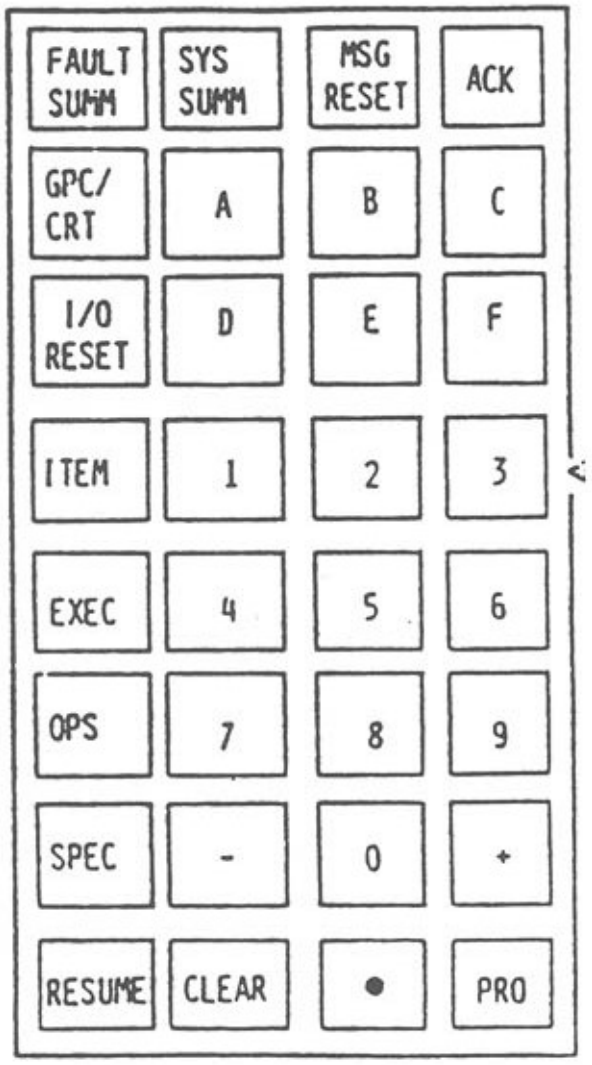

Figure 4–8. Keyboard layout of the Shuttle computer system. (From NASA, *Data Processing System Workbook*)

explosion of fuel and oxidizer would get out of control. To ensure this, engineers designed combustion chambers and cooling systems better than optimum, with the result that the engines weighed more than less-protected designs, thus reducing performance. Engineers also set fluid mixtures and flow rates by mechanical means at preset levels, and levels could not be changed to gain greater performance. 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. Neither the throttleability or the performance enhancements could be accomplished without a digital computer as a control device.

In 1972, NASA chose Rocketdyne as the engine contractor, with

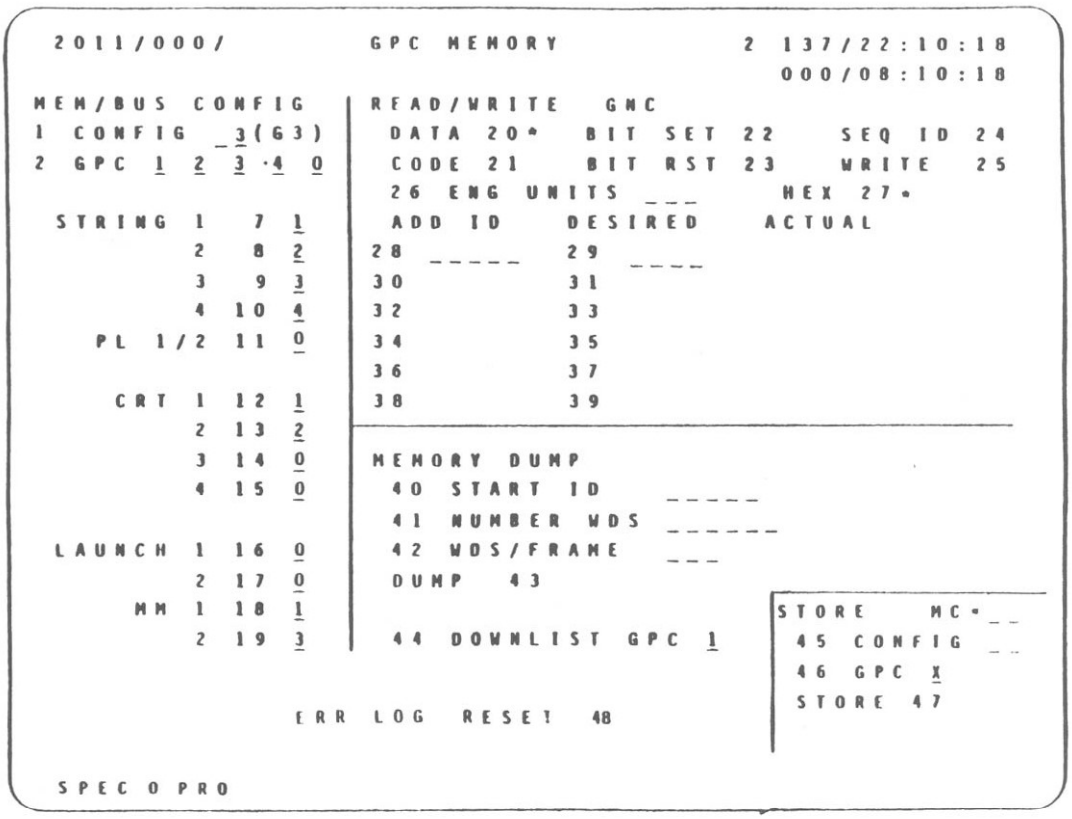

Figure 4–9. A typical display of the Primary Avionics Software System. (From NASA, *Data Processing System Workbook*)

Marshall Space Flight Center responsible for monitoring the design, production, and testing of the engines. Rocketdyne conducted a preliminary study of the engine control problem and recommended that a distributed approach be used for the solution[166]. 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 are best handled with a dedicated computer. Both Marshall and Rocketdyne agreed that a digital computer controller was better than an analog controller for three reasons. First, software allows for greater flexibility. Inasmuch as the control concepts for the engines were far from settled in 1972, NASA considered the ease of modifying software versus hardware a very important advantage[167]. Second, the digital system could respond faster. And third, the failure

detection function could be simpler[168]. Basically, the computer has only two functions: to control the engine and to do self tests.

The concept of fail operational/fail-safe is preserved with the engine controllers because each engine has a dual redundant computer attached to it. Failure of the first computer does not impede operational capability, as the second takes over instantly. Failure of the second computer causes a graceful shutdown of the affected engine[169]. Loss of an engine does not cause any immediate danger to a Shuttle crew, as demonstrated in a 1985 mission that lost an engine and still achieved orbit. If engine loss occurs early in a flight, the mission can be aborted through a RTLS maneuver that causes the spacecraft essentially to turn around and fly back to a runway near the launch pad. Slightly later aborts may lead to a landing in Europe for Kennedy Space Center launches. If the engine fails near orbit it may be possible to achieve an orbit and then modify it using the orbital maneuvering system engines.

Controller Software and Redundancy Management

As with the main computers on the Shuttle, software is an important part of the engine controller system. NASA managers adopted a strict software engineering approach to the controller code. Marshall's Walter Mitchell said, "We try to treat the software exactly like the hardware"[170]. In fact, the controller software is more closely married to engine hardware than in other systems under computer control. The controllers operate as a real-time system with a fixed cyclic execution schedule. Each major cycle has four 5-millisecond minor cycles for a total of 20 milliseconds. This is a high frequency, necessitated by the requirement to control a rapidly changing engine environment. Each major cycle starts and ends with a self test. It proceeds through engine control tasks, input sensor data reads, engine limit monitoring tasks, output, another round of input sensor data, a check of internal voltage, and then the second self test[171]. Some free time is built into the cycle to avoid overruns into the next cycle. So that the controller will not waste processing time handling data requests from the primary avionics system, direct memory access of engine component data can be made by the primary[172].

As with the primary computers in the Shuttle, the memory of the controller cannot hold all the software originally designed for it. A set of preflight checkout programs have to be stored on the MMU and rolled in during the countdown. At T-30 hours, the engines are activated and the flight software load is read from the mass memory[173]. Even this way, fewer than 500 words of the 16K are unused[174].

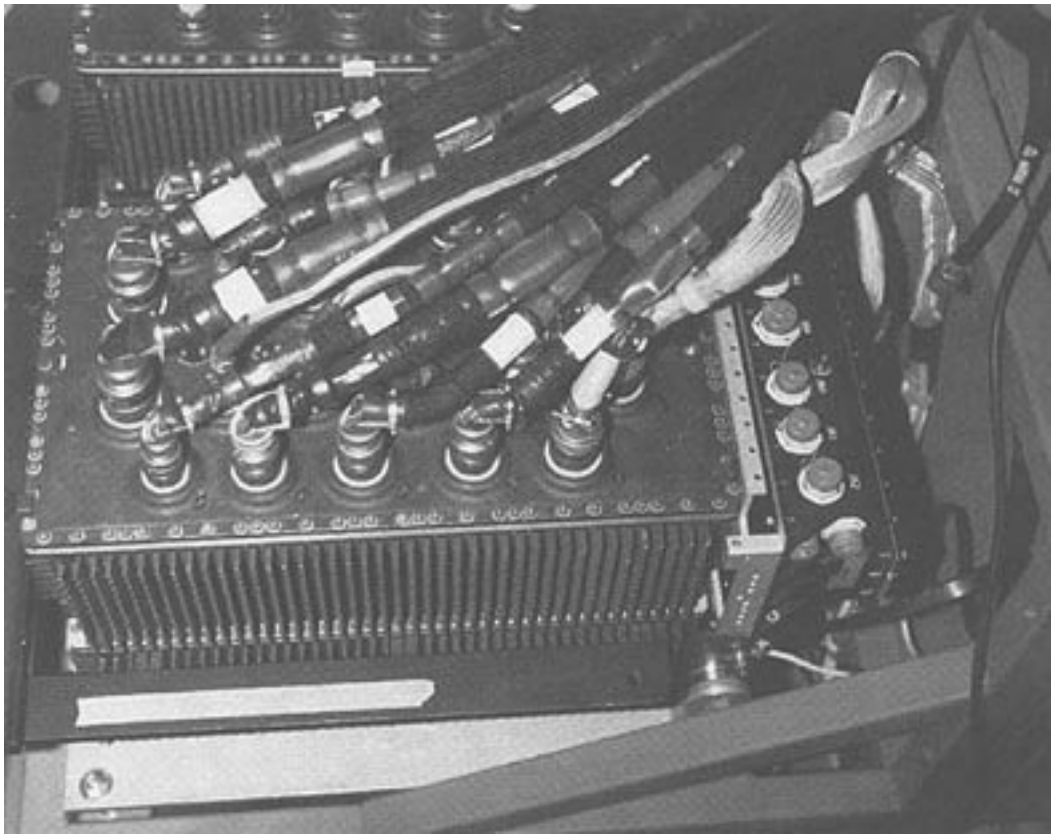

Figure 4–10. A Shuttle Main Engine Controller mounted in an engineering simulator at the Marshal Space Flight Center. (NASA photo)

Although redundant, the controllers are not synchronized like the primary computers. Marshall Space Flight Center studied active synchronization, but the additional hardware and software overhead seemed too expensive[175]. The present system of redundancy management most closely resembles that used by the Skylab computers. Since Marshall also had responsibility for those computers and was making the decision about the controllers at the same time Skylab was operating, some influence from the ATMDC experience is possible. Two watchdog timers are used to flag failures. One is incremented by the real-time clock and the other, by a clock in the output electronics. Each has to be reset by the software. If the timers run out, the software or critical hardware of the computer responsible for resetting them is assumed failed and the Channel B computer takes over at that point. The timeout is set at 18 milliseconds, so the engine involved is "uncontrolled" by a failed computer for less than a major cycle before the redundant computer takes over[176].

Box 4-7: Shuttle Engine Controller Hardware

The computer chosen for the engine controllers is the Honeywell HDC-601. The Air Force was using it in 1972 when the choice was made, so operational experience existed. Additionally, the machine was software compatible with the DDP 516, a ground-based Honeywell minicomputer, so a development environment was available. Honeywell built parts of the controller in St. Petersburg, Florida and shipped those to the main plant in Minneapolis for final assembly; within a couple of years, all the construction tasks moved to St. Petersburg. By mid-1983, Honeywell completed 29 of the computers[177].

The HDC-601 uses a 16-bit instruction word. It can do an add in 2 microseconds, a multiply in 9. Eighty-seven instructions are available to programmers, and all software is coded in assembly language[178]. The memory is 2-mil plated wire, which has been used widely in the military and is known for its ruggedness. It functions much like a core memory in that data are stored as a one or zero by changing the polarity in a segment of the wire. Each machine has 16K of 17 bits, the seventeenth bit used to provide even parity[179]. Plated wire has the advantage of having nondestructive readout capability.

The controllers are arranged with power, central processor, and interfaces as independent components, but the I/O devices are cross strapped. This provides a reliability increase of 15 to 20 times, as modular failures can be isolated. The computers and associated electronics are referred to as Channel A and Channel B. With the cross strapping, if Channel A's output electronics failed, then Channel B's could be used by Channel A's computer[180].

Packaging is a serious consideration with engine controllers, since they are physically attached to a running rocket engine, hardly the benign environment found in most computer rooms. The use of late 1960s technology, which creates computers with larger numbers of discrete components and fewer ICs, means that the engine builders are penalized in designing appropriate packages[181]. Rocketdyne bolted early versions of the controller directly to the engine, resulting in forces of 22*g* rattling the computer and causing failures. The simple addition of a rubber gasket reduced the *g* forces to about 3 or 4. Within the outer box, the circuit cards are held in place by foam wedges to further reduce vibration effects[182].

THE FUTURE OF THE SHUTTLE'S COMPUTERS

The computers in the Shuttle were candidates for change due to the rapid progress of technology coupled with the long life of each Shuttle vehicle. First to be replaced were the engine controllers. By the

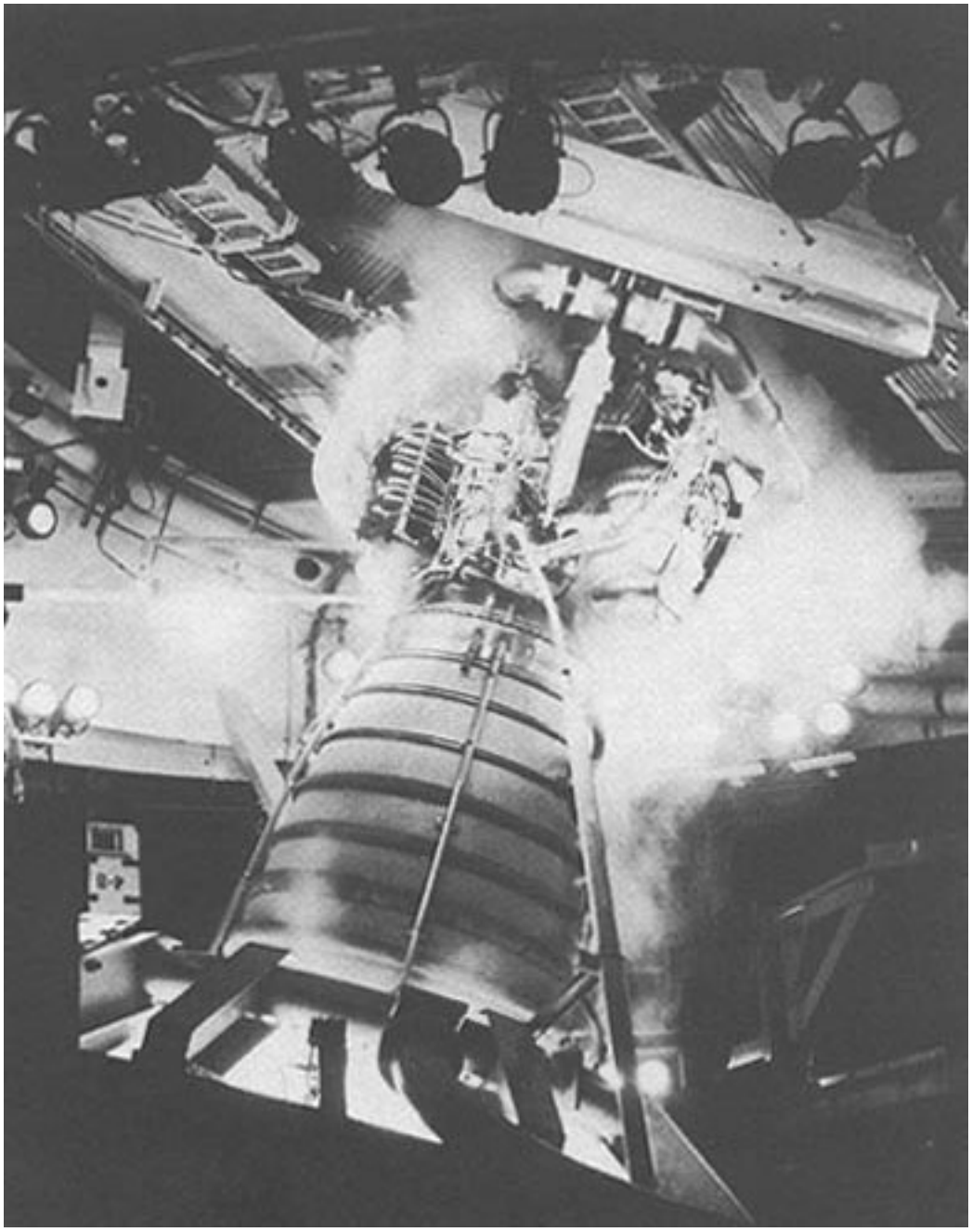

Figure 4–11. A Shuttle main engine in a ground test. The Controller can be seen mounted on the left side of the combustion chamber. (NASA photo 885338)

early 1980s, Marshall Space Flight Center began studying a Block II controller design because it was becoming impossible to find parts and programmers for the late 1960s components of the Block I[183]. The revised computer uses a Motorola 68000 32-bit microprocessor. When selected, it was clearly the state of the art. Instead of plated wire, a CMOS-type semiconductor random-access memory is used. Finally, the software is written in the high-level programming language, C. Such a computer reflects the current design and components of a ground-based, powerful digital control system. The C language is also known as an excellent tool for software systems development. In fact, the UNIX operating system is coded in it.

Aside from the processor change, the Block II's memory was increased to 64K words. Therefore, the entire controller software, including preflight routines, can be loaded at one time. Semiconductor memories have the advantages of high speed, lower power consumption, and higher density than core, but lack core memory's ability to retain data when power is shut off. Reliability of the memory in the Block II computer was assured by replicating the 64K and providing a three-tier power supply[184]. Both Channel A and Channel B have two sets of 64K memories, each loaded with identical software. Failure in one causes a switch-over to the other. This protects against hardware failures in the memory chips. The three tiers of power protect against losing memory. The first level of power is the standard 115-volt primary supply. If it fails, a pair of 28-volt backup supplies, one for each channel, is available from other components of the system. Last, a battery backup, standard on most earth-based computer systems, can preserve memory but not run the processor.

The significance of the evolution to Block II engine controllers is that they represent the first use of semiconductor memories and microprocessors in a life-critical component of a manned spacecraft. Honeywell scheduled delivery of a breadboard version suitable for testing in mid-1985. The new controller is physically the same length and width, so it fits the old mounting. The depth is expected to be somewhat less. When the first of these computers flies on a Shuttle, NASA will have skipped from 1968 computer technology to 1982 technology in one leap.

IBM's new version of the AP-101 (the F) incorporates some of the same advantages gained by the new technology of the engine controllers. Increasing the memory to 256K words means that the ascent, on-orbit, and descent software can be fitted into the memory all at once. (This is not likely to happen, however, because of the pressing need to improve the crew interfaces and expand existing functions.) Higher component density allows the CPU and IOP to be fitted into one box roughly the size and weight of either of their predecessors. Execution speed is now accelerated to nearly 1 million operations per second, twice the original value. In essence, NASA has finally acquired the power and capability it wanted in 1972, before the software requirements showed the inadequacy of the original AP-101.

As in the engine controllers, the memory in the AP-101F is made of semiconductors. Power can be applied to the memory even when the central processor is shut down so as to keep the stored programs from disappearing. A commercially available error detection and correcting chip is included to constantly scan the memory and correct single bit errors. These precautions help eliminate the disadvantage of volatility while still preserving the size, power, and weight advantages of using semiconductors over core memories[185].

CONCLUSION

The DPS on the Shuttle orbiter reflects the state of software engineering in the 1970s. Even though the software was admittedly the key component of the spacecraft, NASA chose the hardware before the first software requirement was written. This is typical of practice in 1972, but less so now. NASA managers knew that time and money spent on detailed software requirements specification and the corresponding development of a test and verification program would save millions of dollars and much effort later. The establishment of a dedicated facility for development was an innovative idea and helped keep costs down by centralization and standardization. A combination of complete requirements, an aggressive test plan, a decent development facility, and the experience of NASA, Rockwell, Draper, and IBM engineers in real-time systems was enough to create a successful Shuttle DPS.

Even as the system took shape, NASA managers looked to the future of manned spacecraft software. Increased automation of code and test case generation, automated change insertion and verification, and perhaps automated requirements development are all considered future necessities if development costs are to be kept down and reliability increased. In the 1980s, a new opportunity for software development and hardware selection presents itself with NASA's long-awaited Space Station. NASA has another chance to adopt updated software engineering techniques and, perhaps, to develop others. Success in space is increasingly tied to success in the software factory.