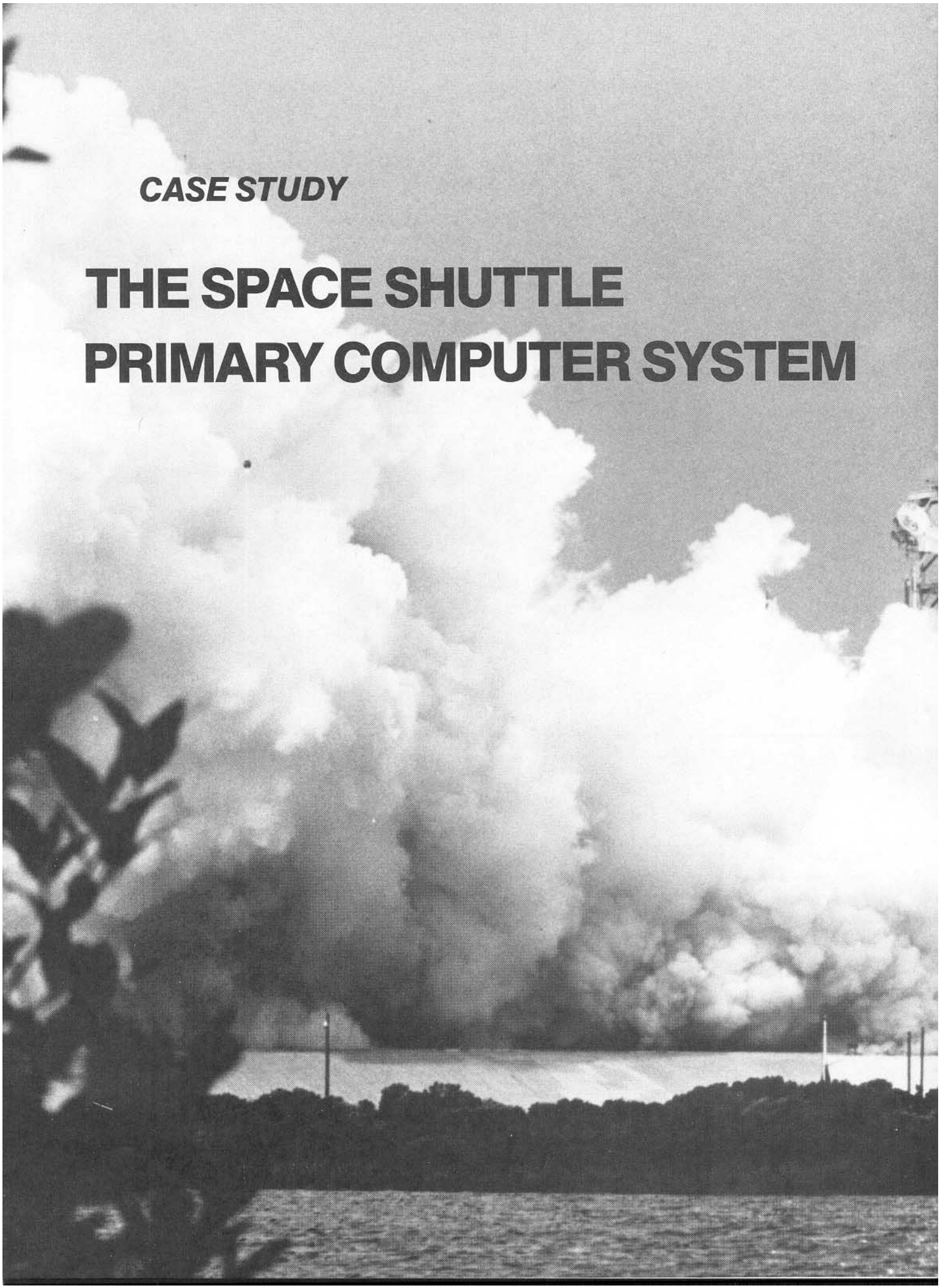
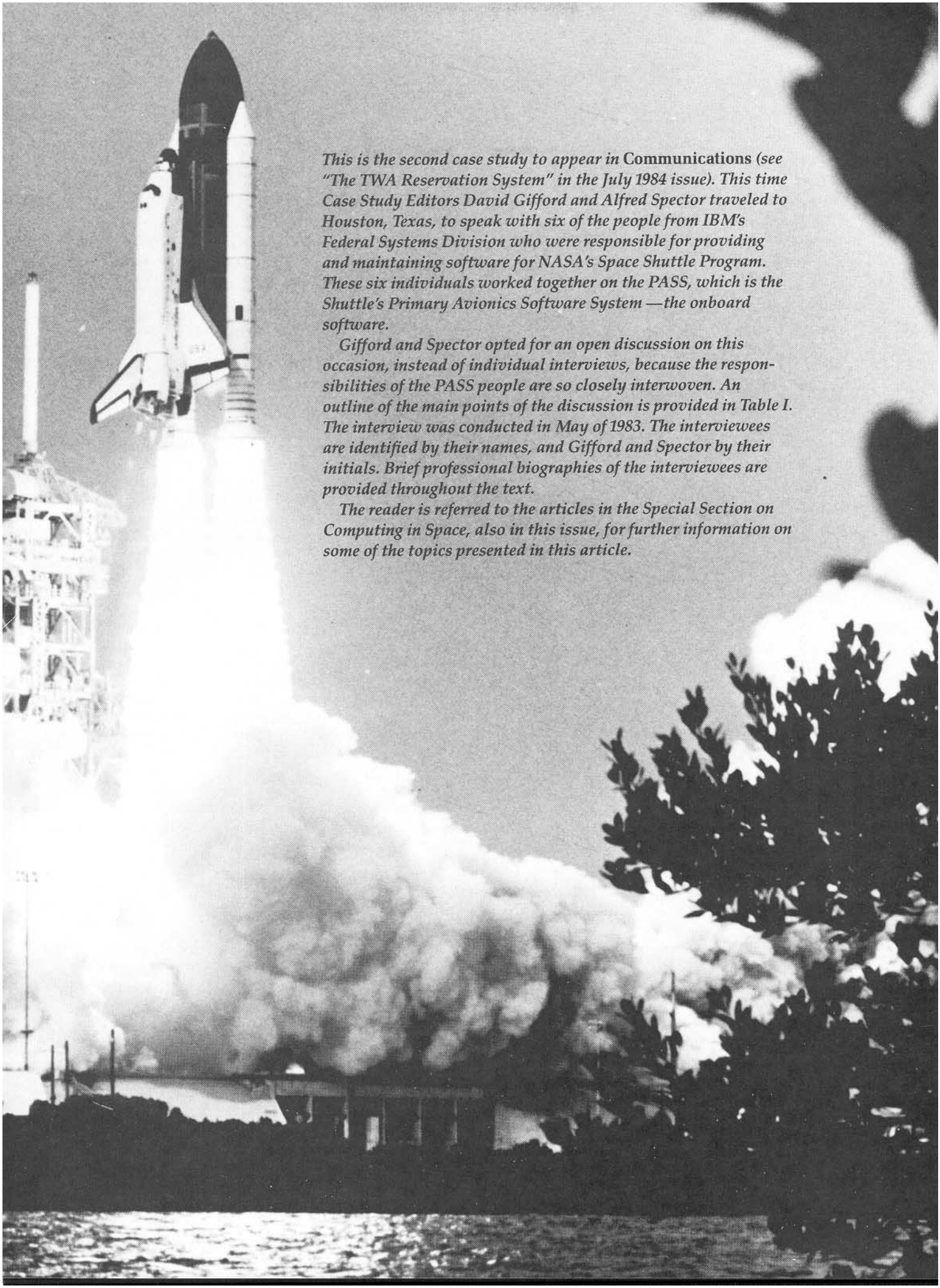


CASE STUDY

THE SPACE SHUTTLE PRIMARY COMPUTER SYSTEM



This is the second case study to appear in **Communications** *(see "The TWA Reservation System" in the July 1984 issue). This time Case Study Editors David Gifford and Alfred Spector traveled to Houston, Texas, to speak with six of the people from IBM's Federal Systems Division who were responsible for providing and maintaining software for NASA's Space Shuttle Program. These six individuals worked together on the PASS, which is the Shuttle's Primary Avionics Software System —the onboard software.*

Gifford and Spector opted for an open discussion on this occasion, instead of individual interviews, because the responsibilities of the PASS people are so closely interwoven. An outline of the main points of the discussion is provided in Table I. The interview was conducted in May of 1983. The interviewees are identified by their names, and Gifford and Spector by their initials. Brief professional biographies of the interviewees are provided throughout the text.

The reader is referred to the articles in the Special Section on Computing in Space, also in this issue, for further information on some of the topics presented in this article.

THE SPACE SHUTTLE PRIMARY COMPUTER SYSTEM

IBM's Federal Systems Division is responsible for supplying "error-free" software for NASA's Space Shuttle Program. Case Studies Editors David Gifford and Alfred Spector interview the people responsible for designing, building, and maintaining the Shuttle's Primary Avionics and Software System.

ALFRED SPECTOR and DAVID GIFFORD

PROJECT OVERVIEW

AS. **What is the extent of IBM's involvement with the Space Shuttle Program?**

Macina. IBM is involved in a number of different projects. The one that our group is involved with is development of the Primary Avionics Software System (PASS). The PASS is the highly fault-tolerant, on-board software that controls most aspects of Shuttle operation. IBM also has a contract to develop software and supply commercial hardware for the Mission Control Center in Houston. Other major activities include an integration and development contract for the Launch Processing System at the Kennedy Space Center in Florida, and a contract to supply the on-board computer hardware from Owego, New York.

AS. **What can you tell us about the historical background of IBM's involvement with the PASS?**

Macina. We won the initial software contract in 1974. Our first objective was to develop software for the Shuttle ALT (Approach and Landing Tests), which took place in 1975 and 1976. There were approximately 10 such flights, some where the Shuttle remained attached to its Boeing 747 carrier, and others where the vehicle was released at 20,000 feet and glided to a landing. Our initial effort involved the development of the operating system and a small part of the entry applications software. The operating system was probably the most difficult development task we had to tackle, since it involved our first attempt to form a synchronized set of redundant computers.

We also had the contract for the Software Development Lab (SDL). The SDL was both a six-degree-freedom flight simulator and a program management facility for performing builds, compiles, integration, and configuration management. (Eventually, the SDL evolved into the Software Production Facility (SPF).)

It was during this early phase that we defined our basic organizational structure: a development organization and a separate verification and validation (IVV) group controlled by different senior managers. At the time this was a somewhat unique idea in that the verification group was almost as large as the development group and did not begin producing a product for several months after its formation. They learned from and participated with the development people. Our key concept for the verification organization was that they should proceed with an assumption that the system was totally untested.

There was a transition period between 1976 and 1981 during which some of our people continued work on the ALT program, while others began work on the STS-1 software.[1] The Shuttle was evolving rapidly, and the requirements we received were constantly changing. Fortunately, the ALT program gave us a chance to learn how to deal with this constantly evolving system.

From a development and test standpoint, we came very early to the conclusion that as much time as possible should be spent designing and coding the system carefully, since it's more cost effective to prevent problems than to correct them. When you're pressed by schedules, you tend to develop software as fast as possible and to fix the problems later. This is always expen-

[1] STS-1 denotes the first Shuttle flight (for Space Transportation System No. 1). Subsequent flights continue this sequence (STS-2, STS-3, etc.).

TABLE I. Outline of the Interview

sive. The constant updating of the requirements early in the program contributed to our schedule problems. We would build software and then rush to incorporate late changes knowing all along that we would have to "test-out" any problems later in the process.

The next major change in the organization came after the STS-1 mission. We had expended all of our effort on the first flight and had to make certain adjustments to support multiple flights. We reduced our testing, increased our development activity, and began to automate. We built standardized test case libraries and improved our software production facility to make it more automated and more available to developers and verifiers.

AS. **Before we get to the on-board system, could you discuss some of the other computer systems involved in the Shuttle mission?**

Macina. The launch processing system located at the Kennedy Space Center provides the interface between the launch team and the on-board system. It uses an array of approximately 40 minicomputers, each having two color terminals to monitor and control major Shuttle subsystems. For example, one computer monitors the Shuttle main engine system while another interfaces with an on-board piece of hardware called the pulse code modulation master unit (PCMMU) that sends data to the ground. Many of the minicomputers receive their data from the on-board computer system. Two bidirectional channels called launch data buses (LDBs) tie the on-board computer system to the launch processing system.

AS. **What does the software at the Mission Control Center in Houston do?**

Macina. The software at the MCC performs a number of functions: It receives and processes telemetry from the vehicle through worldwide ground stations; it performs trajectory predictions, abort predictions, and other functions of that type; it provides the link from the ground controllers up to the vehicle, called uplink, which lets ground controllers perform most of the functions available to the crew via the 50–60 displays they have access to on board; and it controls the worldwide tracking network.

During a mission the MCC uses two large mainframe computers to maintain reliability. One is on-line—it processes data and interfaces with the tracking network, the flight controllers, and the vehicle in real time. The second is a dynamic standby computer, which can be brought on-line rapidly if the primary computer encounters problems.

DG. **Let's discuss the on-board system.**

Macina. The on-board system (called the DPS, for Data Processing System) utilizes five computers, which are known as GPCs (General-Purpose Computers). The PASS resides in a maximum of four of these at any one time. Since the software needed to support an entire mission would be too large to occupy the primary memory, it has been divided into eight overlays that make up the operational programs of PASS. The overlays are obtained from one of two mass storage devices (MMUs) and are only performed when the vehicle is in a quiescent flight phase, that is, prelaunch or on-orbit.

Flight-critical programs like those used for ascent and entry execute in a redundant set: Four computers simultaneously execute identical code and synchronize their I/O activities. The fifth computer is reserved for the backup flight system, or BFS, which was independently programmed by Rockwell. The BFS can perform critical flight functions if a catastrophic failure disables the PASS. The BFS can only be engaged by crew action.

Remember that the Shuttle cannot operate without the DPS. Most of the subsystems do not have manual backups. This is why NASA has tried to achieve fail-operational/fail-safe reliability: After a single failure, the Shuttle remains fully operational, and the mission continues; after a second similar failure, it can still return safely.

Let me describe the general interfaces that connect the on-board system to the rest of the Shuttle. Most crew commands go through the on-board system. It's a fly-by-wire vehicle. All sensors, effectors, and crew controls are connected to the on-board computers through multiplexors that are linked directly to the computers via a 1-MHz data bus network. When an astronaut throws a switch, that input is actually read by the computer via a multiplexor. An example is the rotational hand controller used by the crew to fly the vehicle. These are transducers whose deflections are measured by a multiplexor, which in turn provides the

TONY MACINA

A.J. (Tony) Macina became the manager of flight operations for IBM's On-board Space Shuttle Program in July 1983. His responsibilities include the preparation, integration, system test, and field maintenance of on-board software systems. Macina joined IBM in 1974, became manager of the Shuttle System Software Test Department in 1976, and manager of the Applications Performance Test Department in 1977. He has also held positions as technical staff to the manager of Shuttle software verification and to the manager of on-board space systems. Before coming to IBM, he was an aircraft design engineer for Lockheed Aircraft and a task manager for Apollo and Space Shuttle separation systems for TRW.

data to the computers for action. In addition to the cockpit switches and controls, there are 55–60 different display formats that provide the interface between the DPS and the hundreds of subsystems on board. They range from graphics and trajectory displays to pure monitoring displays.

AS. **How is the PASS coded and organized?**
Clemons. The software has an operating system written in assembler and applications written in HAL/S, a high-order language developed by Intermetrics, Inc., of Cambridge, Massachusetts. There are a number of stories about how the language was named—one is that it's from the computer in the movie *2001*, although that is probably apocryphal. HAL/S is a real-time, structured engineering language that is very readable. Theoretically, at least, it should limit the amount of structure-induced errors because it makes the programmer pay attention to structure as the software is being developed. It's somewhat similar to PL/1.

Macina. The operating system uses about 35K of the 106K 32-bit words available, and includes a priority-driven process management scheme and the on-board display system software. Our process management scheme lets important processes run at the expense of lower priority processes if necessary. It is essentially a self-correcting system that off-loads low-priority processing when CPU demand is high.

DG. **What function does the PASS perform?**
Macina. You can divide the applications into three sets. The first set contains guidance, navigation, and flight control. In this set there are separate groups of algorithms for ascent, on-orbit operations, and entry. Another set contains systems management—it's used during the on-orbit phase, primarily for opening and closing the payload doors, controlling the manipulator arm, and payload monitoring and control. The third set is used for vehicle checkout both prelaunch and while the vehicle is on-orbit.

AS. **How would you define guidance, navigation, and flight control?**
Macina. During all flight phases, all commands that are issued to the vehicle control systems on the boosters, Shuttle main engines, and aerodynamic surfaces originate in the DPS. The control laws, gains, filters, and so on, are imbedded in the flight software. These functions constitute the flight control software.

The guidance software issues the commands to the flight control software. Guidance obtains the current state (position, attitude, and velocity) from the navigation software. Guidance, knowing where the vehicle is from navigation as well as the desired state of the vehicle, determines what commands should be issued to get the vehicle to the desired state. All three pieces work hand in hand. These systems can be operated in a fully automatic mode, and during typical ascents, they actually do take the vehicle all the way to orbit without crew intervention.

In addition, there are various aids that provide data to the navigation system. The inertial measurement units (IMUs) assist the navigation software in maintain-

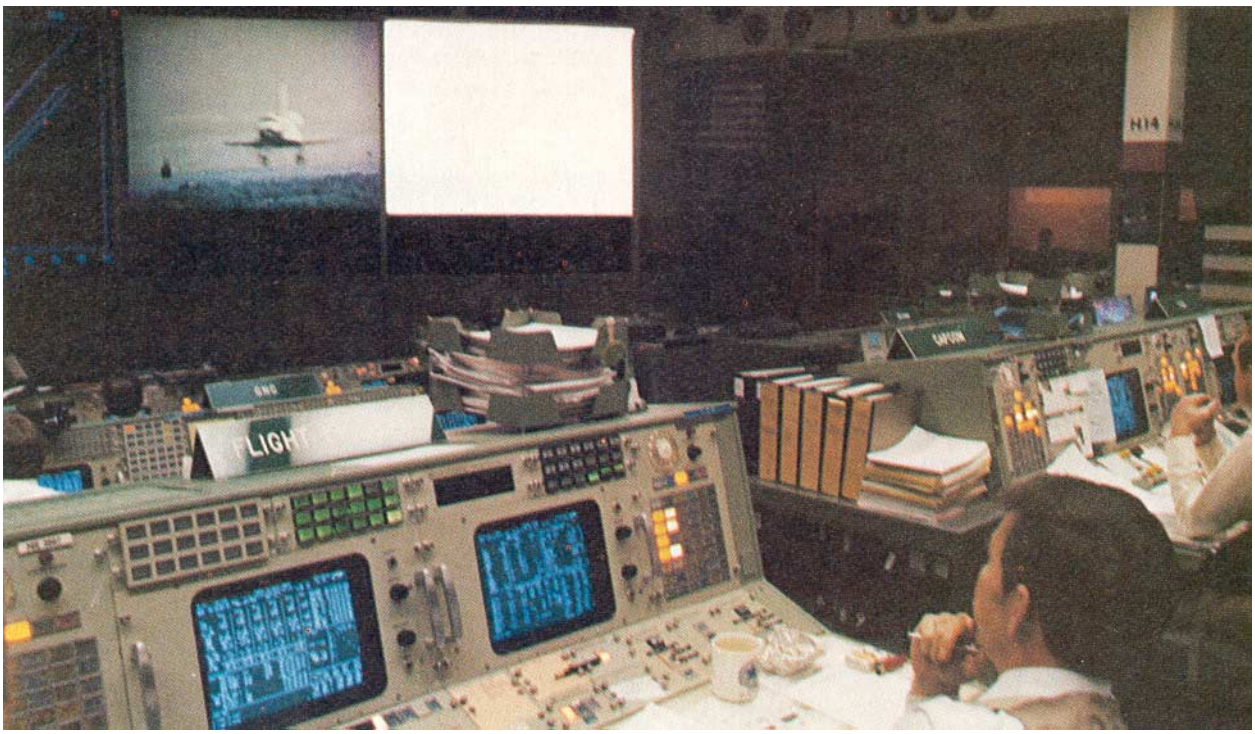


FIGURE 1. The Control Room at Mission Control in Houston.

ing vehicle position and attitude by measuring and integrating accelerations. They aren't perfect, however—for an entry where it's important to hit a point on the ground 300 feet wide and maybe several hundred feet long, additional navigational aids are necessary. The navigation software makes use of a standard tactical air navigation system, which provides range and bearing information from stations on the ground as well as a Microwave Scan Beam Landing System for glide slope and landing control.

AS. **Could the computers fly an entire mission—an unmanned mission?**
Macina. No, not the way the software is designed today. There are still a number of critical functions that require the crew. In a system as complex as the Shuttle, the human component provides a lot of flexibility for problem solving. For example, once during a flight test the radar altimeter locked on the nose gear at 2500 feet. The radar altimeter would be critical in an automatic flight, and this is the kind of problem that we'd want to eliminate before we would consider trying one.

DC. **Is the DPS involved with the Shuttle's payloads?**
Macina. Yes and no, depending on the payload. The deployments of the SBS (Satellite Business Systems) and ANIK (Canadian) geo-synchronous satellites on STS-5 were totally directed by the PASS software. The checkout, spin-up, and deployment are all controlled by the software via crew displays. The TDRS-IUS (NASA Communication Satellite) combination, on the other hand, was not. We performed only a monitoring function for that system.

AS. **From a global point of view, how do the on-board and ground systems fit together?**
Macina. About 72 hours prior to launch, an on-board computer is brought up, and the controllers begin checking out various on-board systems. About 15 hours prior to launch, the on-board system actually begins to run in the redundant set (four computers). The software allows controllers to begin checking out the systems that are going to be critical to launch guidance, navigation, and flight control. During this period the controllers also align the IMUs. At T-20 minutes, the ascent program is loaded from the mass storage devices into the redundant computer set. This program provides the functions that allow the Shuttle to achieve a stable orbit.

At this point the ground launch processing system (LPS) still has control and is configuring, checking, topping off fuel tanks, etc. The on-board launch sequencer is both passively monitoring the systems and fielding requests from the LPS to supply data. The interface becomes more dynamic as lift-off approaches. The two systems are no longer communicating through human controllers. The launch sequencer on the ground is talking directly to the on-board launch sequencer.

At about T-25 seconds a "go-for-auto-sequence-start" command is sent to the on-board system from the ground system. At that point, the ground system becomes passive and the on-board system takes over. The vehicle is still connected to the ground by umbilicals, but the LPS is now in a monitoring role. The on-board redundant set launch sequencer then takes the count from T-25 seconds through engine ignition, ascent, and into orbit, all of which takes about an hour.

AS. **When does the vehicle start communicating with Houston?**
Macina. Prior to lift-off the vehicle transmits data to the ground via two paths: an RF link called the downlink and the LDBs. Houston is in communication with the vehicle throughout the countdown but only in a monitoring role via the downlink path. As the vehicle leaves the launch pad, control of the flight is passed from the Launch Control Center at the Kennedy Space Center to the MCC in Houston.

About half of the data that are telemetered to the ground via the downlink originate from the on-board computers. On the basis of the requirements that are given to us prior to every flight, we precisely tailor the information that will be required to support the mission. Most of the changes are associated with the payloads. The guidance, navigation, and flight control parameters remain relatively stable from mission to mission. The uplink function is the exact reversal of the downlink. It relays commands from the ground controllers in the MCC to the on-board computers, thereby giving the ground access to many of the same displays and controls available to the crew.

DG. **How do you make the system reliable?**
Macina. During flight-critical phases like ascent and entry and during preflight fuel loading—periods where loss of the system might mean loss of the vehicle—the PASS executes in a redundant set of computers. In this set, the software is synchronized at the applications level. We are actually providing bit-for-bit identical data to each computer, and each computer is issuing bit-for-bit identical data commands to the various subsystems. We run our programs in these flight-critical phases in four of the computers, which are synchronized to within 150 microseconds of each other.

As I mentioned, there is a fifth computer that runs the Backup Flight System (BFS). Early on, NASA was concerned about the possibility of a generic software problem in the PASS. What if there were a "bug" in the PASS that brought the entire primary system down? The way they alleviated their fears was by developing independent ascent and entry software from a subset of the requirements they had given us. This independent software was written by Rockwell International and resides in the fifth computer.

During ascent and entry, the BFS is essentially in a listening mode: It monitors the PASS data buses as they collect data from the inertial measurement units, rate gyros, etc. Essentially, it's flying the vehicle, except

that all of its commands are disregarded. By depressing a single switch, the crew can disable the primary system and engage the backup.

The decision to engage the BFS is totally a crew function. Their procedures identify certain situations for which the switch should be made: for instance, loss of control, multiple consecutive failures of PASS computers, or the infamous two-on-two split where the computers split up into two pairs (we've never seen this occur). To date the crew has never had to use the BFS during a mission.

Once the vehicle is in orbit, one or two PASS computers run GNC programs; another executes the system management program, which runs the remote manipulator and interfaces with the payloads; and a third is loaded with an entry program and then turned off as a precaution. It provides the capability of flying back in the event all the other computers fail. The backup flight system in the fifth computer, which also contains entry software, is turned off as well. In this configuration there are two computers loaded with software that will allow a return to Earth.

AS. **If these two computers and the mass memory units failed, could you uplink programs to the computers?**
Macina. Yes, but only entry programs. That would be a third way to return. It's a very long, tedious process, since the data rates are low, and the vehicle is in and out of communication with ground stations. The procedure involves uplinking the programs and then downlinking the contents of the computer and verifying that the transmission was received correctly.

THE SHUTTLE COMPUTERS

AS. **Can you give us some more detailed information about the Shuttle computers?**
Macina. A single computer (GPC) is made up of two packages: a CPU unit and an I/O device unit (IOP), with a total of 106K 32-bit words of memory. The CPU, a System/4 Pi, Model AP-101 manufactured by IBM, is an off-the-shelf processor and has probably been around for 10 or 12 years. Our original contract specified that we use off-the-shelf hardware as much as possible. The 4 Pi design has been used in a number of other aerospace vehicles. For example, certain B-52 aircraft and the B-1 Bomber use the 4 Pi technology.

The IOP was specially built and designed for the Shuttle, using 4 Pi technology. It contains 24 "time-sliced" processors that handle the data buses on the Shuttle. The IOP obtains its instructions from main memory and is actually in contention with the CPU for memory access.

AS. **How much do the GPCs weigh?**
Macina. About 120 pounds each, for the combined CPU and IOP boxes.

AS. **How do the components fit together?**
Macina. An IOP and a CPU are interconnected by a parallel data channel and essentially perform as a single unit. Each pair of units interfaces with on-board systems through 19 or 20 prime interface devices known as MDMs, for multiplexor/demultiplexors. MDMs retrieve data from the various sensor devices, convert it to a Manchester code, and place it on a DPS data bus upon a request from the computers. MDMs make the system very flexible in that sensor devices can be added with only minor changes to the MDMs and the PASS software.

DG. **Can you describe the 4 Pi AP-101 in more detail?**
Macina. The AP-101 is a 450,000-operations-per-second machine, which isn't extremely fast by today's standards. The eight programs used during a typical mission average about 75 percent CPU utilization for most flight regimes, which leaves us well within the capability of this machine. Very early on in the development phase, we did have some trouble with excessive CPU utilization. We went through a very detailed scrub of the software requirements and the code to achieve the CPU utilization we have today.

AS. **When you mention 450,000 operations per second, I get the impression that you do a lot of floating-point operations.**
Macina. You would expect that to be true for a software system so heavily geared toward scientific applications like guidance and navigation. However, the reality is that only a small part of the operations are floating point. Most of the instructions generated by the compilers and assemblers tend to be loads, stores, branches, etc.—the data handling and bookkeeping instructions.

AS. **Is the memory core?**
Macina. Yes, it's ferrite core. By today's standards it seems outdated, but it does have certain advantages; for instance, it's inherently nonvolatile when power is removed.

AS. **Can you give us an idea of the failure rate of the GPCs?**
Macina. The computers that flew the first five Shuttle flights had a mean time between failure of approximately 6000 hours for the entire set.

DG. **Tell us about the I/O devices that connect to the Shuttle computers.**
Macina. There are four CRTs and three keyboards. Three of the CRTs reside forward in the cockpit of the Shuttle, with two keyboards just below them. There's one CRT and one keyboard in the aft cockpit area for payload operations. For each CRT display, there's a separate display processor called a display electronics unit (DEU). The GPCs generate dynamic data and the DEUs generate and format the background data for the CRTs. The crew has access to 55 or 60 different display for-

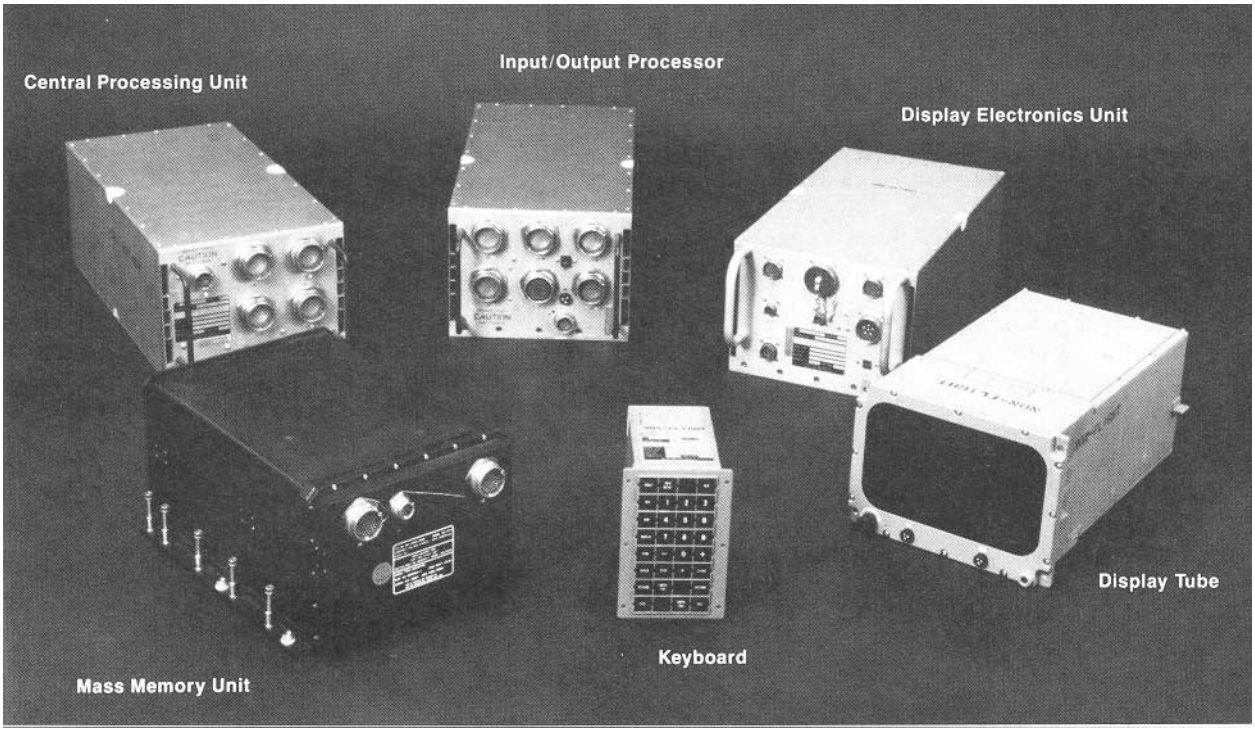


FIGURE 2. The on-board primary computer system. A central processing unit and an input/output processor make up a general purpose computer (*upper left*). There are five sets of these two boxes on board, all stored below the cockpit. The display system consists of display electronics units, display tubes, and keyboards (*right*). There are four DFUs (three forward in the cockpit, one aft), four display tubes (three forward, one aft), and three keyboards (two forward, one aft). There are two mass memory units on board (*lower left*). They are stored with the GPCs below the cockpit.

mats via the four display units. The units are interchangeable and can access any or all of the available formats.

There are two mass storage devices. Each contains a 600-foot continuous tape on which there are three areas. Each area has a complete copy of all eight operational programs. This means there are six copies of all the programs needed to fly the Shuttle on board.

The mass storage devices contain more than just the PASS software. They store software for the backup flight computer and the DEUs. They also hold software for the computers that monitor the health of the main engine. (One is mounted on each main engine—they communicate through an interface unit to the PASS GPCs.) The software and data in the mass storage devices are addressed in 512-word blocks, which provide a capacity of about eight million 16-bit words.

DG. **Is the mass storage ever written during flight?**
Macina. One of the eight PASS programs is a mass memory write capability. Though we have never needed to use it during a flight, this program is used during tests to modify the software to fit the needs of the various test facilities.

By the way, we also have a general main memory write capability available to the crew and to ground controllers. One of the PASS displays allows them to write hexadecimal numbers directly into a GPC's main memory. There are certain failure circumstances where this capability might be useful, but these are very rare situations. At first we were reluctant to incorporate such a capability, but NASA wanted the flexibility. They were worried that a potential generic software problem might go undetected until the vehicle was in orbit. Our first reaction was to want a sign that would come up and say "your warranty is void" when they used the capability.

AS. **Have you used this capability yet?**
Macina. As with the mass storage write capability, this capability is used mostly during testing. We have on some minor occasions used it during a mission. On one flight an overflow occurred in a counter that was keeping track of the number of Reaction Control Jet firings. NASA asked if they could have a general memory write procedure for zeroing that counter. Nothing very critical.

DG. **Where is all of this equipment located?**
Macina. All of the avionics hardware, including the computer system, is located in an equipment bay below the flight station in racks. The crew has complete access to the area. On the first flight, there was a lot of uncertainty about the new vehicle and the new computer system. NASA opted to take an extra preloaded computer along. They called it an "entry-in-a-suitcase." The thinking was that, if the entire system were to fail,

the crew could install this computer, attach the cooling units and pin connectors, and use it to fly the vehicle. They brought it along for two or three flights until they became confident enough to let the weight constraints convince them it was unnecessary.

JACK CLEMONS

J.F. (Jack) Clemons is manager of avionics flight software development and verification for the Shuttle on-board computers. He has also worked on the Apollo and Spacelab programs. Clemons joined IBM in 1974 and became manager of the Shuttle software independent verification organization in 1979 before taking over both the software development and verification organizations in 1982. Before coming to IBM, he worked for GE Valley Forge from 1967 to 1968 and TRW Systems in Houston from 1968 to 1974.

PROJECT ORGANIZATION

AS. **How many people from IBM FSD have been involved in the Shuttle?**

Clemons. Right now there are about 100 people developing code for the on-board system and about 80 working on IVV. These numbers exclude support activities like developing the software development and test bed and providing support to the various fields sites, which require about another hundred people.

There are two types of verification that we do on this project. One consists of verifying that our code meets the NASA-specified requirements, that is, that we meet the "letter of the law." Beyond that, however, we check our software in simulations of real flight situations—in what we call flight performance verification. We find out if the software can actually fly the vehicle and whether we can achieve orbit for different weights or under different kinds of hardware failures. We fly a series of "stress" cases and have our engineers look at the software response. Performance verification evaluates both the software requirements and the software design itself.

DG. **Have you tried to structure the software so that it can be changed easily?**

Clemons. By changing certain data constants, we can change relatively large portions of the software on a mission-to-mission basis. For example, we've designed the software so that characteristics like atmospheric conditions on launch day or different lift-off weights can be loaded as initial constants into the code. This is important when there are postponements or last-minute payload changes that invalidate the original inputs.

AS. **How do the payloads affect your software?**

Clemons. We anticipated from the beginning that we were going to have a lot of different payloads and payload interfaces, so we built modular software to interface to the payloads. We can plug in a large number of constants to restructure the way the software works in terms of its I/O, its interfaces, the monitoring and annunciation of payload conditions, the commands it sends and receives, and the displays it generates. All of these changes, relating both to launch conditions and to payloads, are known as reconfiguration changes. They are implemented with simple data changes, rather than with code and logic redesign.

We have also devised a tool that we call the systems management preprocessor. Systems management (SM) is our generic term for the software that does both Shuttle vehicle and payload monitoring and control. The SM preprocessor takes data relating to the orbital payload reconfiguration that we get from NASA, converts data formats where necessary, and places those data in the right places in the on-board software memory. SM is a fairly hefty piece of software: about 20,000–30,000 lines of code.

AS. **How do you verify the code produced that way?**

Clemons. We use a simulator to call up the reconfigured displays, simulate the payload activities, and fly portions of the mission. This can give us an idea if SM is working properly. This involves a considerable amount of testing—it becomes really the pacing item or limiting factor for this kind of reconfiguration. An automating device like the SM preprocessor is no better than a particular programmer's conception of what it can and should do.

We are now building still another tool that will help with the SM verification process. It will automatically "decompile" the memory images generated by this SM preprocessor and compare the results with the original inputs—it's sort of an "inverse preprocessor." It will be developed and programmed by a group that's independent from the one that built the original preprocessor. Differences will arise if either tool is in error or if there's some misinterpretation in input data. In any event this tool will flag every difference between the requirements that were loaded and what the code does. Once we satisfy ourselves that we haven't designed the same fault into both the SM preprocessor and the SM decompiler, we'll be able to supplement or replace a lot of the testing we do now and condense our test case analysis to a smaller set.

Even with these tools, however, the number of changes is growing as users become more numerous and flights more frequent. We're more or less going out of the business of verifying code and into the business

of verifying that new sets of constants will perform properly with our software. Our manpower requirements would be coming down because of all the sophisticated and efficient testing tools we've implemented, except that the quantity of missions we are now supporting and all the verification we have to do for these reconfiguration changes are keeping them up.

AS. **Are you responsible for checking the input data when a mission is postponed to ensure that everything will go as planned at a later date?**
Clemons. Yes, although in that particular case, there would only be certain classes of changes that we could accept and still be able to support the flight. NASA doesn't come to us two days before a mission to swap out one payload for another with totally different software needs.

AS. **Do you have to do a mass memory reload of the system if weather causes a postponement?**
Clemons. For the types of changes that we've been talking about, there's no need to recompile code for weather changes. We would just patch very specific data values on the mass memory devices. It's a little different when the Shuttle software code itself has to be changed. That could involve changing and recompiling some very specific pieces of code.

AS. **What's your cycle for major software changes?**
Macina. Software changes fall into two major categories: flight reconfigurations, which entail data changes to support specific missions, and capabilities changes, which entail the development of new code. We get an initial set of payload and initialization data six months before a typical flight. These data are based on predictions of the flight profile for the projected launch day and the payload. We deliver initial software about 20 weeks before the flight. About 10 weeks before the flight, NASA updates about 10 percent of the data based on refined information about the payload, the launch date, etc. If we were reconfiguring for only one mission at a time, this 20-week process would be much shorter, but with overlapping flights and manpower constraints, the 20-week period is realistic.

AS. **That would mean handling six flights at a time, working from the flight-a-month projection.**
Macina: By 1985 we may have as many as 10 going simultaneously.

AS. **How much data are you talking about for these reconfiguration changes?**
Clemons. We have access to about 8,000 to 10,000 parameters. Typically we change anywhere from 800 to 1,000 different 16-bit words for a given flight. In addition we reconfigure the SM and payload control software on each flight. This involves the regeneration of common data areas, which in turn facilitates the generation of new payload control displays and I/O within the software.

AS. **It doesn't sound as if there are many modules in your system that remain the same from one mission to another.**
Clemons. There are not a lot of modules for which code changes from one mission to the next. However, over the first dozen Space Shuttle flights, we'll probably change more than half the modules in response to NASA-requested capability enhancements.

Macina. The system was designed to allow us to be able to support a wide variety of missions without changing code. As a very simple example, consider that the flight control algorithms used during ascent are *n*th-order polynominals, the coefficients of which are changeable initialization values. We can satisfy ourselves that those polynominals meet the intended requirements by substituting a variety of values during testing. The result is a piece of operating code that we believe is correct and meets the requirements as intended. We can then change those coefficients on a flight-to-flight basis with very little need for additional testing.

If we were really confident in the process as a whole, we wouldn't perform any testing after reconfigurations. We test because we want to be absolutely sure that the combination of all these changes really can fly the vehicle, either from a requirements or an implementation point of view. Consequently on every flight we run a set of so-called performance verification cases where we simulate all mission phases: ascent, abort, entry, etc. We aren't the final verifiers of the avionics system either: NASA actually takes our software and does additional testing in the SAIL, in the SMS, and on the vehicle prior to each flight.

DG. **Does the crew ever make any suggestions?**
Clemons. Since this is an R & D project, the crew lets us know when they're not satisfied with the performance. We've been getting a steady stream of requirements changes that originate from the flight crews, and we expect this to be an ongoing process.

Macina. Most of these changes are enhancements to existing capabilities. In general they're meant to make the interfaces more user friendly, or to make it easier for a crewman to train or to operate the vehicle more safely.

AS. **What other kinds of changes do you see?**
Macina. There are very few major new capabilities. There is one new capability that will be introduced on STS-9 to provide more navigation code to permit a rendezvous capability.[2] This will be used for the first time on STS-11 to allow a rendezvous with the Solar Maximum Satellite in an attempt to repair it.

Clemons. The bulk of the code change requirements

[2] This capacity was actually used on flight 41C in early 1984 to rendezvous with the malfunctioning Solar Max satellite.

we get now relate to on-orbit enhancements for making the vehicle more flexible and operational for orbital use. The first five Shuttle flights had very few payload capabilities other than for launching and monitoring. We've had to develop further software modifications for deploying a payload with a remote manipulator system, maneuvering away from it, and retrieving it again to bring it back into the cargo bay.

Capabilities of this kind require significant precision in the guidance and navigational systems. We have to design code that can provide that kind of precision. Consider that the Shuttle is now heavily dependent on worldwide ground-based communication, monitoring, and control. Because of drift in the IMUs, the on-board navigation system is not precise enough to make an accurate entry and landing after five days in space without ground assistance. The risk would be unacceptable. We have to uplink a new state vector before the crew takes the craft out of orbit. There's an effort being made to upgrade the DPS system and the software to allow for more vehicle autonomy. We expect to see quite a few code changes as a result.

Once the Air Force starts launching out of Vandenberg Air Force Base, there will be another set of changes to make. The Air Force's requirements are somewhat different from NASA's.

AS. **I get the impression that you have countless versions of similar software. It must be a headache to keep track of them.**
Macina. It's the ultimate configuration management problem. Let me give you an idea of what we're looking at right now (as of May 1983): We're beginning to develop more on-orbit autonomy capabilities. We're finishing up STS-7, doing the 10 percent update. We're starting to reconfigure the software for the new payloads on STS-8. We've begun the six-month reconfiguration on STS-9 (Spacelab), and we're also reconfiguring the sixth-month set of software for STS-10 (Department of Defense). Altogether, we've got five activities going right now, all during what we consider a low-flight-frequency phase.

DG. **Are the same people involved on all the activities?**
Macina. Yes—the development, verification, and reconfiguration people work on many different things simultaneously.

Clemons. The independent verification group is verifying the final load for STS-7, doing the sixth-month update for STS-8, doing the new rendezvous capability for STS-9, and supporting the Department of Defense testing on STS-10.

We've had to use a rigorous configuration management control process. We're building a lot of tools that should ensure that a programmer can sit down at a terminal to make an update and can use panels that will ask which system is to be worked on and tell that programmer which kinds of entries would accidentally update the wrong version. (These would therefore be illegal.) As a management team, we've tried to keep the burden of configuration control off the individual worker as much as possible. We aren't asking programmers and test people to keep track of the configuration on a day-by-day basis.

DG. **How often do you get new environment and vehicle hardware models from NASA or other contractors?**
Macina. There's a set of model data that has to change every time we reconfigure software. We receive a new set of simulator initialization constants once per flight. Whenever new or updated hardware is added to the vehicle, there is a potential need for new simulator models.

AS. **Can you tell us more about the specifications that you get from NASA and Rockwell?**
Macina. There are three levels: Level A specifications consist of a general description of the operating system and the overall system architecture. These specifications were provided very early on (in 1975) and have remained relatively stable. Overall, the Level A specifications were a very good narrative description of the system: They presented the idea of the four computers in a redundant set, along with the I/O techniques we have used throughout the project, etc. They also specified the requirements for reliability, CPU utilization, transport lags for the flight control system, skews between machines, etc.

For the applications, we were given Level B specifications that provided a general description of all software functions and defined the rates at which each task is supposed to run. From the Level B specifications, we developed a functional design specification, which was reviewed with NASA.

The Level C requirements are the most detailed. They actually tell us how to code many areas. For example, they provide specific navigation and flight control algorithms, although specifications in other areas are somewhat more general. The Level C specs fill about 20 individual volumes, each addressing a different function (guidance, navigation, flight control, vehicle utility, SM, etc.). Overall, the Level C requirements are the base against which we test.

AS. **Are they mostly in English?**
Macina. Yes. The true requirements tend to be in prose, although in some cases (like navigation) the prose is supplemented with equations and flow diagrams.

AS. **What kinds of tests do you run?**
Clemons. For STS-1, when virtually all the code was new, we built a set of test cases that explicitly verified all the requirements just as we received them. The independent verification group is expected to deliver error-free code, which is a goal we are asymptotically approaching. "Error-free" means 100 percent conform-

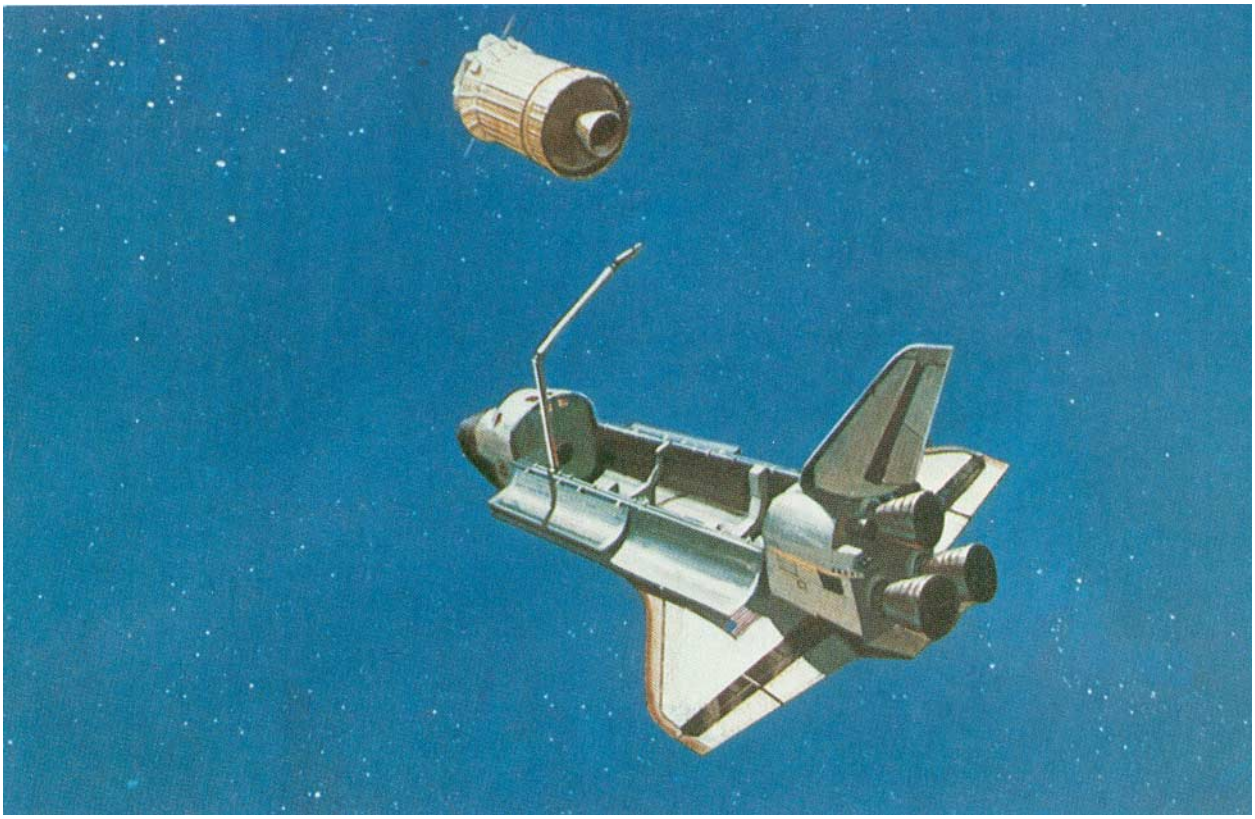


FIGURE 3. Deployment of a Satellite

ity to requirements, which is a massive verification job. For STS-1 we had about 50 people just testing guidance, navigation, and control (which excludes payload operations and the operating system). Those people worked about two years and built about 1000 test cases.

AS. **Could I point to any requirement on any page in those bookshelves and ask you to say how you determined that your code met the requirement?**
Clemons. Yes, and I could show you (1) a test specification that tells you generically how am I going to test the requirement; (2) maybe four or five test procedures written up and reviewed in advance, detailing how that particular requirement is to be tested; (3) a set of test cases; and (4) a set of test case reports.

DG. **Do these procedures hold for later flights?**
Clemons. Basically, yes. However, we can't run the entire STS-1 set—there are just too many flights and not enough people. Instead, we use what we call a "delta" verification approach. We take all of the changes that have been made to the system since we last verified it and explicitly verify each one.

First we assign every module to a specific verification analyst so that there's somebody to vouch for each module. We run a source compare program to define the changes (e.g., to compare STS-2 code with STS-1 code). The analyst accounts for each change in terms of an authorized change request, a discrepancy report, or some other authorizing document. The analyst must see that all of the changes, along with the code effected around them, are tested. This allows us to reduce the number of test cases that we rerun on each succeeding flight. We don't run cases over and over again for confidence but rather to accommodate new capabilities.

The test cases for STS-2 aren't a subset of the first 1000—they may consist, say, of 200 new tests and 200 repeats. By STS-3, there may be 300 total cases, some new and some significantly modified. The rendezvous capacity, for example, was a whole new set of code that we had to build 70 new cases for.

AS. **About how long does it take to build a typical case?**
Clemons. It takes one person about two weeks.

AS. **On the whole, are you satisfied with your software engineering?**
Clemons. The state of the art in software engineering has advanced significantly in the decade since we started this program. A more thorough approach to design reviews, code walk-throughs, and unit test inspections has been formulated. We've been able to start applying these techniques rigorously only in the last several years. We always did design and coding inspections, but not as meticulously as we do now. Though

these inspections are painful and expensive, we would apply that kind of rigor from the beginning if we had it all to do over again. It's easier for people to get used to this kind of rigor if it's in place from the beginning, and quite frankly, it's more cost effective to discover our errors early in the process.

DG. **How does your IVV team work?**
Clemons. Let me review some testing terminology. Classically, at the unit test level there is *white box testing:* The tester knows the design and tests the code against it. Theoretically, all requirements have been taken into account in the design, so unit testing does not have much of a requirements perspective. *Black box testing* is just the opposite: Software is tested strictly in terms of how well it meets the requirements. This is the traditional approach to IVV testing.

Where there are requirements that have been implemented incorrectly, or not at all, black box testing will find the discrepancies. White box testing, however, should identify instances of programmer "creativity," since this is code that goes beyond the letter of the requirements. Errors introduced into this code will not generally be found by comparing code performance to requirements. To find these errors with higher probability, we need a group of independent people who can look at the code, test it, and examine it from a black box and a white box point of view. This pushes our IVV effort toward *gray box testing.*

Our IVV team was originally a black box operation. Once the requirements were specified, we constructed procedures to test our code. Our initial philosophy was that, if we ran enough test cases, we would find all of the problems. However, we soon discovered that it's much better if the IVV organization is intimately familiar with the code, that is, if they're doing gray box testing. The argument against this is that verifiers are going to loose their objectivity if they're involved with the design. Our experience, however, shows that gray box testing lets us find many problems that can't be found by simply running test cases.

Our verification analysts first study and understand the requirements, then build the test specifications and the procedures to test the requirements, and only then perform a code inspection. They expand and supplement their test plan based on that inspection. This maintains a degree of objectivity. I was able to get a quantitative measure of the efficiency of this approach when I had responsibility for a department that was verifying support functions. This group had not previously done any code inspections as a routine part of their verification. I was able to make comparisons between them and three departments I already managed that were doing guidance, navigation, and control (GNC) verification. I tracked the number of discrepancy reports that each organization was discovering. On a per capita basis, the GNC people were finding many more discrepancies than the new people were. But beyond that, over one-half of the discrepancy reports found by the GNC people were by code inspection. This is a much less costly way of finding problems than running test cases. The approach taken by the GNC people required engineers who could look at software, use it as a tool, understand it, and yet maintain verification independence. The payoff is tremendous. If I had it to do over, I would insist on independent code inspection as a mandatory part of the verification process for the entire program.

AS. **As you know, the academic research community has devoted a lot of effort to analytic verification techniques for programs, by means of proof techniques of one form or another. Has any of this work been useful to you yet?**
Clemons. We've just started looking at these techniques. We haven't tried to use the analytic verification techniques as a standard on the project to date, either on the development side or on the verification side. My experience with them has been that they are useful for relatively small pieces of code that aren't complicated too much by real-time interrupts. Also, the proofs are exceedingly tedious without special tools.

I believe that if we asked our people to analyze Shuttle code using those techniques (1) their training might not be appropriate; (2) the complexity and size of this particular piece of software would be too large to experiment with; and (3) there wouldn't be a lot of data available that could indicate whether the techniques were valid, except in very small applications. There are pilot projects within the IBM/FSD to use these techniques, and I hope they bear fruit.

TESTING FACILITIES

DG. **Could you describe the different testing facilities and how they were used for the development of the Shuttle software?**
Clemons. The primary software development and test facility that IBM uses is called the Software Production Facility, or SPF. The SPF contains large IBM mainframes—3033 and 3083 machines. Those machines provide an interface to the terminal users and to the programmers and verifiers that allows them to do all their work from remote terminals. The SPF physically controls the allocation of resources among the programmers, and provides a simulation test bed of the Space Shuttle vehicle and the vehicle environment during flight. The SPF also contains math models that allow us to simulate all aspects of the Shuttle mission, from countdown through landing, with a very high degree of fidelity.

The mainframes are connected to Flight Equipment Interface Devices, or FEIDs. These are hardware components, specially built by IBM in Huntsville, Alabama, that interface the flight computers to the simulated environment. In a typical test, we load the flight code into target AP-101 flight computers connected to the mainframes. Math models with the environment data that

correspond to the flight that we're simulating are also loaded into the mainframes. In addition, the FEIDs allow us to interrupt and freeze the flight computers on any specified instruction fetch or data reference within the GPC memory. We can examine computer data, transfer the data into engineering units, make graphs and tables, and insert patches. We have similar capabilities with the simulated environment. For example, a verification analyst who is simulating an ascent and wants to see that the ascent guidance is working properly under an engine failure condition can build an ascent launch trajectory, stop the simulation at an appropriate time, cause the failure to be introduced into an engine, and then resume execution and monitor what happens. It's a very powerful development tool.

The programmer or verification analyst can sit at a terminal and build a test deck that is a step-by-step walk through the portion of the flight profile in question. After being submitted into the simulator's execution queue, the test deck is scheduled. It then executes and produces a step-by-step listing that contains all the desired information.

This simulator is the key to our ability to test flight software and is quite versatile: For example, if I have an analyst interested in examining the last few seconds of a flight prior to touchdown, it's obviously not practical to go through an entire launch, various orbital operations, and entry just to get to that point. We can execute a set of nominal loads, ascents, orbits, and entries, and generate checkpoints following their execution. We can then arrange the simulation to restart from any one of those points. A verifier interested in a certain point of the mission can start at the checkpoint nearest to that point.

AS. **Is this facility in use continuously?**
Clemons. Constantly. Three shifts a day.

DG. **What happens if you want to test something having to do with the display consoles?**
Clemons. There are two ways of doing that. We have a set of simulated display console input commands and the capability of providing graphic representations of what the display response is. If we need more dynamic information about the displays, there are display input hardware devices and CRTs available that make it possible to call the displays up physically and photograph them. We minimize this kind of hands-on testing because simulator time is very precious.

DG. **Can you use the mainframes for anything besides simulation?**
Clemons. Yes. The mainframes run MVS, which means that we can support many types of applications. Interactive terminal jobs get priority, and flight simulation jobs (FEID jobs) run in the background. At night, the simulations are run, so typically there is no more than a one-day turnaround on flight simulations.

Macina. The SPF is a massive facility: In terms of hardware, it has two IBM 3033s, a 3083, 95 gigabytes of disk storage, three AP-101 flight computers, and associated support devices. We had as many people developing software for the SPF at one time as we had developing the on-board software. The software effort was really twofold: software for the host computers (simulator and program management software) and software for the FEIDs. The most difficult development problem was interfacing the various unlike pieces of hardware that make up the simulator portion of the facility.

DG. **Aren't there other test beds that are actually more authentic than the SPF, and could you give us a description of them?**
Clemons. A point to make first is that none of the other facilities have software testing as their primary objective. They use software as a tool for testing other things. There's the Shuttle Avionics and Integration Laboratory (SAIL), which is owned by NASA but operated by Rockwell. SAIL integrates the software with the hardware. Remember that, except for the flight computers, our SPF does not contain the facilities you would find in a real vehicle (e.g., actuators). SAIL tests the integration of the hardware and software components for every flight. There's also a simulator at Rockwell's facility in Downey, California, called the FSL (Flight Systems Lab).

DG. **What does that do?**
Macina. It's similar to SAIL in that it does for the on-orbit and entry portions of the flight what SAIL does for the ascent portions.

Clemons. The other important simulator is the crew trainer, what is called the Shuttle Mission Simulator or SMS. The SMS gives us a test environment we can't get elsewhere—the crew's perspective on how they use the software. The people who run the SMS have rather devious minds and generate some crazy scenarios that nobody at IBM could ever have envisioned. They run through various ascents, orbits, and entries in order to see how the crew and software will react to unusual situations.

The crew also has another simulator that does not use the flight software. It uses a functional equivalent of the flight software, and video picture displays. It provides two-dimensional representations of different things the crew would see through the cockpit windows. As they work the instruments, they can look at this simulator and see just what they would be seeing if they were flying. It's very realistic.

Macina. The main training cockpit is identical to the Shuttle cockpit. It has a hydraulically driven moving base that gives some feeling for G forces. It vibrates on launch and bumps on landing. Suffice it to say that it provides a very faithful representation. NASA has been using this technology since the moon landing.

AS. **Could you describe a training scenario on the SMS that caused a problem for you?**
Clemons. Yes—it was a "bad-news–good-news" situation. In 1981, just before STS-2 was scheduled to take off, some fuel was spilled on the vehicle and a number of tiles fell off. The mission was therefore delayed for a month or so. There wasn't much to do at the Cape, so the crew came back to Houston to put in more time on the SMS.

One of the abort simulations they chose to test is called a "TransAtlantic abort," which supposes that the crew can neither return to the launch site nor go into orbit. The objective is to land in Spain after dumping some fuel. The crew was about to go into this dump sequence when all four of our flight computer machines locked up and went "catatonic." Had this been the real thing, the Shuttle would probably have had difficulty landing. This kind of scenario could only occur under a very specific and unlikely combination of physical and aerodynamic conditions; but there it was: Our machines all stopped. Our greatest fear had materialized—a generic software problem.

We went off to look at the problem. The crew was rather upset, and they went off to lunch.

AS. **And contemplated their future on the next mission?**
Clemons. We contemplated our future too. We analyzed the dump and determined what had happened. Some software in all four machines had simultaneously branched off into a place where there wasn't any code to branch off into. This resulted in a short loop in the operating system that was trying to field and to service repeated interrupts. No applications were being run. All the displays got a big X across them indicating that they were not being serviced.

AS. **What does that indicate?**
Macina. The display units are designed to display a large X whenever the I/O traffic between the PASS computers and the display is interrupted.

Clemons. We pulled four or five of our best people together, and they spent two days trying to understand what had happened. It was a very subtle problem.

We started outside the module with the bad branch and worked our way backward until we found the code that was responsible. The module at fault was a multipurpose piece of code that could be used to dump fuel at several points of the trajectory. In this particular case, it had been invoked the first time during ascent, had gone through part of its process, and was then stopped by the crew. It had stopped properly. Later on, it was invoked again from a different point in the software, when it was supposed to open the tanks and dump some additional fuel. There were some counters in the code, however, that had not been reinitialized. The module restarted, thinking it was on its first pass. One variable that was not reinitialized was a counter that was being used as the basis for a GOTO. The code was expecting this counter to have a value between 1 and X, say, but because the counter was not reinitialized, it started out with a high value. Eventually the code encountered a value beyond the expected range, say $X + 1$, which caused it to branch out of its logic. It was an "uncomputed" GOTO. Until we realized that the code had been called a second time, we couldn't figure out how the counter could return a value so high.

We have always been careful to analyze our processes whenever we've done something that's let a discrepancy get out. We are, after all, supposed to deliver error-free code. We noticed that this discrepancy resembled three or four previous ones we had seen in more benign conditions in other code modules. In these earlier cases, the code had always involved a module that took more than one pass to finish processing. These modules had all been interrupted and didn't work correctly when they were restarted. An example is the opening of the Shuttle vent doors. A module initially executes commands to open these doors and then passes. A second pass checks to see if the doors actually did open. A third pass checks to see how long time has run or whether it has received a signal to close the doors again, etc. Important status is maintained in the module between passes.

AS. **Isn't flight control multipass?**
Clemons. Yes, in a broad sense. But every pass through flight control looks like every other. We go in and sample data, and based on that data, we make some decision and take action. We don't wait for any set number of passes through flight control to occur.

For the STS-2 problem, we took three of our people, all relatively fresh from school, gave them these discrepancy reports (DRs) from similar problems, and asked for help. We were looking for a way to analyze modules that had these multiple-pass characteristics systematically. After working for about a week and a half, they developed a list of seven questions that they felt would have a high probability of trapping these kinds of problems. To test the questions, we constructed a simple experiment: We asked a random group of analysts and programmers to analyze a handful of modules, some with these type of discrepancies, some without. They found every one of the problems and gave us several false alarms into the bargain. We were confident they had found everything.

We then called everybody in our organization together and presented these results. We asked them to use these seven questions to "debug" all of our modules, and ended up finding about 35 more potential problems, which we turned into potential DRs. In many instances, we had to go outside IBM to find out whether these discrepancies could really occur. The final result was a total of 17 real discrepancy reports. Of those, only one would have had a serious effect.

It turned out that this one problem originated during a sequence of events that occurred during countdown. A process was invoked that could be interrupted if there was a launch hold. The only way it would be reset to its correct initialization values was if a signal

was sent from the ground when the launch process was restarted. We incorrectly assumed that this signal was always sent. Had we not found this problem, we would have lost safety checking on the solid rocket boosters during ascent. We patched this one for STS-2 right away.

In retrospect, we took a very bad situation and turned it into something of a success story. We felt very good about it. This was the first time we'd been able to analyze this kind of error systematically. It's one thing to find logic errors, but in a system as complex as this, there are a lot of things that are difficult to test for. Despite a veritable ocean of test cases, the combination of requirements acting in concert under certain specific conditions is very difficult to identify, let alone test. There's a need for more appropriate kinds of analysis.

DG. **How confident were you beforehand that this kind of thing couldn't happen?**
Clemons. We were confident, but we understood that there are always risks when humans do software development and testing. Prior to that time, we had done everything possible to test the software; we had used all of the techniques available to us, even going so far as to explicitly ask people to look for initialization and cleanup problems. The nature of the errors that we discovered up to that time did not point us to anything more generic to search for in the design. If you're suggesting that there could be other classes of things like this multipass problem that we haven't addressed yet, though, it's possible.

Despite all our analysis and testing, the remote and convoluted combination of events that can finally break a module concerns me. Of the few errors our process has missed, the preponderance have been of that kind. We've collected enough of them now so that I've been able to put a task team on them to do exactly what we did for multipass. The discrepancy reports are not linked by anything other than the fact that they occurred in scenarios where there were a lot of unlikely conditions that occurred in combination before the software got to the error. We haven't encountered anything as bad as the STS-2 crew training incident, but we have gotten some discrepancy reports. Hopefully, this task team will come up with another breakthrough.

DG. **Everybody in the computer business has been surprised at the frailty of the human mind at times. We create immense systems with complex internal interactions that we have little hope of understanding. As you add function to the Shuttle system and its follow-ons, will you really be able to understand enough about that software and its interactions to verify its performance adequately? Would you ever consider saying, "Yes, I can add that function, but I don't feel confident enought to verify that it would function correctly"?**
Clemons. We have, on occasion, urged NASA to forgo new capability enhancements for just those reasons. There are some nonmandatory changes that would be awfully nice but are just too complex. We have steadfastly resisted making any changes to the operating system. As we get further away from our base set of test cases, there are certain classes of changes that become just too risky.

Macina. Consider this example: When the on-board system is running as a redundant set, it's possible for the crew to have the same display up on two CRT units. Two crewmen could be accessing the same set of software in an unprotected way. Because this could result in intercomputer communications problems that might cause the redundant set to break apart, NASA would like us to come up with some way to protect against such concurrent usage. To provide this added protection would require a major redesign of the user interface software, a function that has remained relatively stable over the past five or six flights. We believe, and NASA agrees, that the risk of the crew not synchronizing their use of the displays is lower than the risk associated with redesigning the user interface portion of the operating system.

DETAILED SYSTEM OPERATION—NO REDUNDANCY

DG. **We thought it would be a good idea to talk about the system first without considering redundancy—as though the whole system were operating on a uniprocessor. We can go over the redundancy later in greater detail. For something like ascent, then, let's take some of the highest priority tasks and discuss inputs, outputs, displays, and so on.**
Macina. The highest priority application task in all flight phases is the flight control high frequency executive (HFE). It runs once every 40 milliseconds. It's scheduled cyclically and executes to completion within 8 to 12 milliseconds, depending on the cycle. When it's scheduled, it identifies the cycle and performs a table lookup to determine which processes to execute. Before the HFE begins its computations, it initiates I/O to collect the sensor information it needs. Typical sensors are rate gyros, accelerometers, etc. This information is combined with commands from the guidance software. The HFE then goes through typical flight control algorithms, and a command output is issued to the appropriate vehicle effectors via the data bus network.

AS. **Is the on-board system doing active control during ascent?**
Macina. Yes. It takes a very active high-frequency control system to keep the Shuttle vehicle stable during ascent. The engines are gimballed frequently since the vehicle is inherently unstable. The typical ascent is completely under computer control, and the crew is essentially along for the ride if no failures are encountered.

AS. **Do you control the flight surfaces much on ascent?**

Macina. Yes. As the vehicle goes through high dynamic pressure regions, loads begin to build up on the orbiter wings. The aerodynamic surfaces are deflected to minimize this effect.

DG. **Is flight control also reading information from the crew?**
Macina. What I've been describing so far is essentially the automatic flight control system. The crew, however, has the capability to fly the Shuttle in all flight regimes. Because of the inherent instability of the vehicle during the ascent phase and its rapid acceleration, however, it would be very difficult for them to maintain control. During on-orbit operations and entry, the crew can throw a switch to go from auto to manual mode. Commands are still filtered through the flight control software but are initiated by a crew member. Typically, the crew flies the vehicle manually in the lower portions of the entry trajectory.

DG. **After flight control, what task has the highest priority?**
Macina. For the uniprocessor model, guidance is the next highest priority task, running every 160 milliseconds. Guidance uses iterative algorithms to steer for a particular point in space. It collects data from sensors, receives state information from the navigation software, and issues command information to the flight control software when in the auto mode. In the manual mode, guidance commands are provided to the crew on CRT displays or through the full array of aircraft instrumentation.

AS. **What's the next highest priority?**
Macina. After guidance comes navigation. Navigation collects vehicle acceleration and rate data from the inertial measurement units and navigation aids and computes the position of the vehicle in inertial space using a Kalman filtering technique. Navigation runs every two to four seconds, depending on the flight phase.

What I have described are the major processes. There are a number of other processes interspersed at various priorities. One such process is the software that controls the crew CRT displays. One task polls the keyboard and another outputs to the display. These tasks run every 480 milliseconds—a little more frequently than navigation and a little less frequently than guidance.

Other cyclic processes put data together for the downlist and monitor the uplink. There are also on-orbit processes for the payload bay doors and the remote manipulator arm.

AS. **Are there any processes that run on demand?**
Thomas. Things can be scheduled by means of keyboard input: cyclically or just for one shot. There are as many as 70 or 80 total individual processes that can be scheduled this way. For example, a crewman can request the system to incorporate TACAN data, thereby initiating a process. Altogether, there are typically about 18 active processes (either waiting for something or running) during the ascent and the entry phase.

B.J. THOMAS
B.J. Thomas is currently the manager of hardware engineering for the Space Shuttle Programs in Houston. He has been with IBM since 1965, working on the Apollo/Saturn Program and the Army Safeguard Program before joining the Shuttle team in 1974. He was formerly manager of software test and operations.

AS. **The Shuttle has to maintain an environment for the fragile human beings inside. Is the PASS involved in this process?**
Macina. The PASS performs a fault detection function that monitors the environment and alerts both the crew and the ground if it detects anything outside safe parameters. These are part of the reconfiguration parameters that can be changed from mission to mission. It does not, however, control the environment.

AS. **How many sensors do you monitor?**
Killingbeck. Most of the critical sensors. We pick up the data by reading the PCMMU telemetry stream. The sensors dump their data into its memory for telemetry,

LYNN KILLINGBECK
L.C. (Lynn) Killingbeck is a senior systems analyst at IBM in Houston. He began work on the multiple computer system of the Space Shuttle Program in 1969, with emphasis on redundancy management, fault detection and isolation, computer synchronization, and techniques to assure identical inputs to and outputs from all computers. He is currently working on the definition of the flight data system for the Space Station Program.

and the computer picks them up at that point.

AS. **How do the uplink and the downlink work?**
Macina. The PCMMU collects data from subsystems and sends them through a network signal processor, which converts them into radio signals to be sent down. About half of these data come directly from various subsystems that are independent of the data processing system, and the other half originate in the flight computers. A process in the flight computers running cyclically collects parameters from the various software modules and writes them into a memory in the PCMMU. There are thousands of parameters that can be monitored—they vary from mission to mission.

Spotz. From the standpoint of the PCMMU, the on-board computers are just another source of telemetry data. The PCMMU monitors data directly from the vehicle, formats them, and sends them to the ground. During that process there's an area in the unit's memory called the OI RAM (Operational Instrumentation RAM) where the GPC processor can read data that we can't get directly from this bus network.

AS. **Are there times when the Shuttle is not in contact with the ground?**
Macina. Yes. On earlier flights, data would be recorded on-board whenever the Shuttle was out of range of the network of ground stations. The tapes were played back to the ground stations during sleep periods. Once the TDRS satellites are deployed, we'll have much better coverage (we expect to be in touch 80–90 percent of the time).

AS. **Do you know what the data rate is?**
Macina. There is a high data rate (128 Kbits/sec) and a low data rate (64 Kbits/sec).

DG. **Is the uplink encrypted?**
Macina. An operating system-type program in each running computer takes the uplink in and examines the header information to determine the appropriate software destination. To protect ourselves from any malicious ground senders, the system will not accept anything when the Shuttle is out of sight of NASA ground stations. The link, however, isn't encrypted.

Killingbeck. There are ways to protect against transmission errors. We have end-to-end checking, since many commands are in two stages. A command will not be acted on until it's been sent back to the ground and confirmed. Critical things, like the loading of state vectors, are always done in two stages.

Eiland. There's also range testing for some of the variables, depending on which application receives the uplink message. That's a second level of protection.

Spotz. Keep in mind that the uplink does not usually relay much information for controlling the vehicle. The most important thing is the updated state vector, which is sent up about once every two orbits and is only about 100 bytes.

DG. **What's the channel capacity of the uplink?**
Killingbeck. The software can accept commands about once every 160 milliseconds. The buffer is 32 words.

AS. **I noticed that you don't have an alphanumeric keyboard—the system doesn't seem to be particularly user friendly. Can you tell us a little bit about the user interface?**
Thomas. The software is structured into mission phases—ground checkout, GNC, and payload and system management are the three functions. Assuming the astronaut has been in the ground checkout phase at T−20 minutes, he or she would type OPS 101 PRO (OPS and PRO are function keys). This would cause the ascent profile to be loaded off mass memory and enable the computers to handle the ascent.

We present a display that shows the ascent trajectory. This is mode 101 of the ascent software. With solid rocket booster ignition, the software goes into mode 102. A lot of it is automatic moding, whereby software recognizes that an event has occurred. At booster staging, for instance, a function will get rid of the rocket boosters when it senses a drop in the chamber pressure. The software then modes to the second stage guidance, mode 103.

On-orbit, there are a lot of displays available for call up. For instance, the software announces faults by means of an audible tone in the cockpit and a message at the bottom of all of the CRT screens. A message might say "Fuel cell problem, see Display 201." The astronaut would then type in "SPEC 201 PRO," which would call up a display with more detailed information.

AS. **Is it true that you've got an overlay that lets the crew play an outer space video game with real vehicle dynamics?**
Thomas. No, that's not true. As a matter of fact, IBM is particularly interested in asset protection, and we made sure the computers were being used for business purposes only. I don't really think the AP 101 lends itself to games, anyway.

DG. **Have you made software changes to make the interface more user friendly?**
Macina. In fact, a lot of the changes that are coming up in the near future are of this type. This is because, as the missions come more frequently, the crews will have less time for training.

DG. **How forgiving is the current user interface?**
Macina. The crew would have to do something really nonsensical to get into trouble. We don't know of any subtle things that could give them any problems.

Thomas. If they do something wrong, the display tells them to try again.

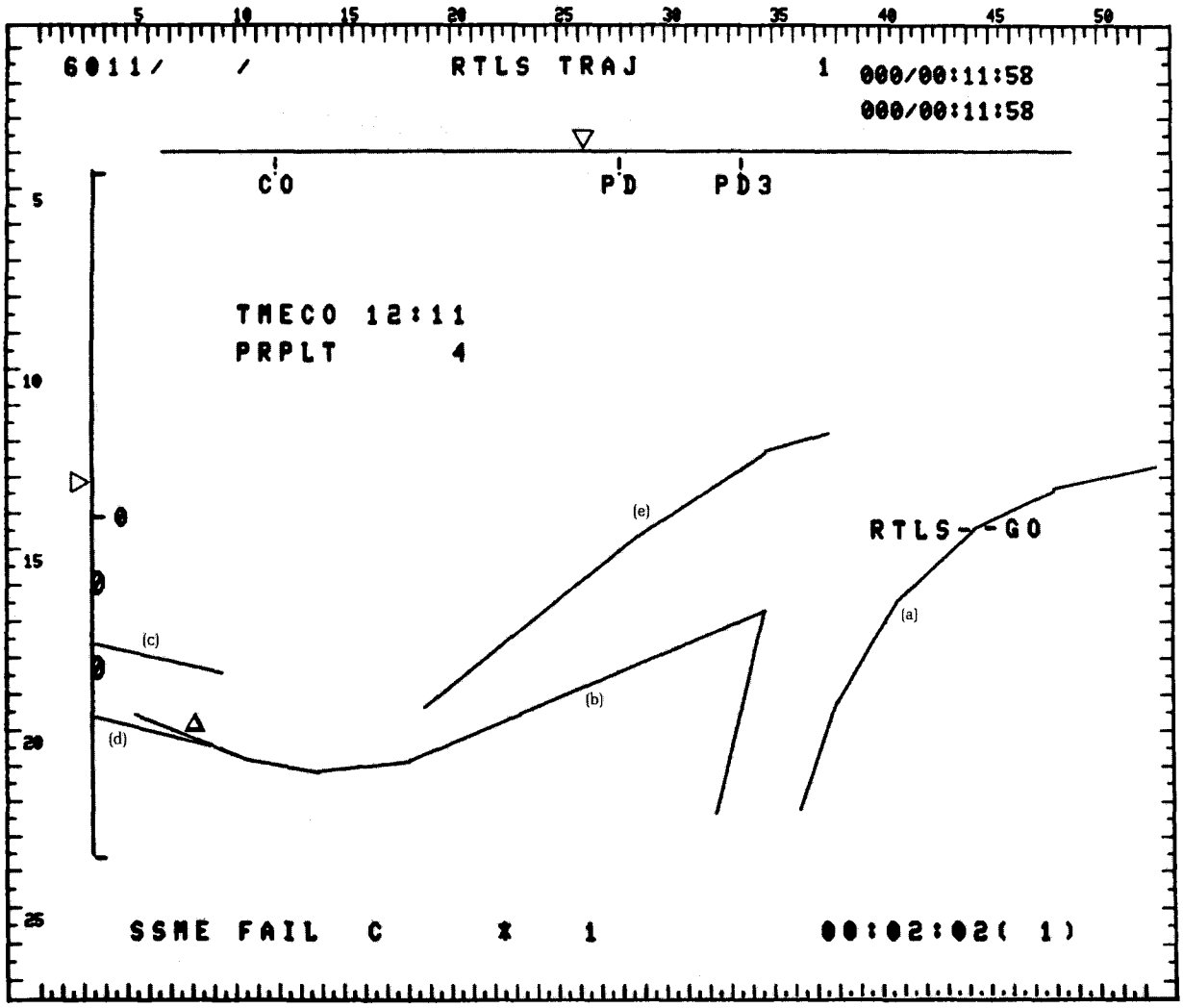


FIGURE 4. Abort Simulation Display

This display was generated by an abort simulation on the SPF. It shows the actual information that the PASS would provide to the crew midway through an abort.

The number on the upper left (6011) and the label RTLS TRAJ indicate that this is a display used during a return to launch site. The date and time in the upper right (000/00:11:58) indicate the day of the mission and the number of hours, minutes, and seconds that have elapsed since lift-off. The data and time below are a timer that the crew can use as they wish.

TMECO 12:11 indicates the predicted time of main engine cutoff, in minutes and seconds from launch. PRPLT 4 indicates the percentage of propellant remaining in the external fuel tank.

The horizontal axis shows the horizontal velocity of the Shuttle. The inverted triangle above it moves leftward as the current horizontal velocity increases. The tick mark labeled CO indicates the horizontal velocity at which engine cutoff should occur. The tick mark labeled PD indicates the horizontal velocity at which the vehicle must be pitched down to permit external fuel tank separation during a two-engine powered return to launch site. The tick mark labeled PD3 is similar but is used for a three-engine powered return to launch site.

The right-pointing triangle to the left of the vertical axis shows the vertical velocity with respect to the nominal vertical velocity for this point in the abort, which is indicated by the zero. Generally, the right-pointing triangle should be very close to the zero mark.

The curved lines in the center of the screen are not plotted against these two axes—essentially, this is two displays in one.

The right-most curve (a) is the statically drawn plot of the nominal ascent profile, showing altitude versus horizontal velocity. The angular line in the central-bottom portion of the screen (b) shows the minimum altitude and horizontal velocity required to achieve a safe landing after a single-engine failure. The two short curved lines above and below this ((c) and (d)) represent limits on dynamic pressure for external fuel tank separation to be successful. The central curved line (e) shows the last opportunity for a return to launch site. Data from the navigation system of the PASS are used to dynamically update the triangle located in the lower right-hand portion of the screen; this shows the current vehicle altitude and horizontal velocity. The two nearby circles predict the state 30 and 60 seconds into the future.

Macina. Typically, astronauts don't want too much protection, since it would cut down on flexibility and make the system very rigid.

DG. **From your experience in working with crews in the simulator, do you find that they learn to trust the system more as they work with it more?**
Thomas. Definitely. The STS-6 crew had three extra months to train after a postponement, and seemed to have an easier time with the system as a result.

AS. **Has human error or mistraining led to any strange situations?**
Thomas. There was an odd incident on one of the recent flights. We usually expect the crew to keep two displays on the primary system; during ascent, however, we expect them to have one display on the primary and one on the backup. This particular crew had two displays on the backup, one showing fault detection, the other the ascent trajectory. They had swapped displays in the middle of the ascent, which made us think we had CRT errors in the primary. This was a matter of the crew's preference: There was no reason not to switch, but we were concerned.

AS. **What errors did you see?**
Macina. When displays are transferred from the PASS to the BFS, transient I/O errors are produced for a while. These are reported on the telemetry downlink. I/O error messages are produced whenever a communication path is severed by crew action or a failure.

DG. **Considering guidance, navigation, and flight control, would you say the applications were written using data abstraction techniques?**
Killingbeck. I don't think that modern notions of data abstractions were applied. A lot of this design dates back to the Mercury, Gemini, and Apollo days. We've stayed with what we consider a classical structure for flight control systems. I don't think we've got anything remotely close to data abstractions in the advanced sense of the term.

AS. **What would happen if flight control didn't run at the appropriate times?**
Macina. It's a typical digital flight control problem. Say you're collecting sensor data and you want to issue a command based on that data. If your processing takes too long, the data get stale and the command you send out isn't going to reflect the current state of the vehicle. This is what is known as "transport lag." Excessive transport lag can cause vehicle control instabilities. For the Shuttle, the delay between data collection and command issuance can be no longer than 18 milliseconds.

We have a secondary flight control requirement called jitter. It's a requirement mandating the variation in the interval between initiations of the flight control task (or more directly, between the output commands). The requirement is plus or minus 800 microseconds.

AS. **Are there any processes that have a higher priority than the flight control process on an emergency basis?**
Macina. No. Our applications priorities are numbered 0 to 255. Flight control is 255. The only thing having a higher priority is the operating system, and it will not normally interfere with either the transport lag or the jitter requirements of the flight control application unless there are massive I/O errors.

BILL SPOTZ
W.H. (Bill) Spotz is an Advisory Programmer at IBM Federal Systems Division in Houston. He joined IBM, and the Space Shuttle Program, in 1977. He has participated in development, subsystem testing, and flight support of the on-board software. He is currently investigating advanced software technology applications for the Space Station Program.

REDUNDANT SET OPERATION

Spotz. As you know, we have four computers all in a synchronized redundant set for ascent and entry applications (during the on-orbit phase, different computers are running different applications). In the redundant set, all of the computers get identical inputs and do whatever synchronization is necessary to ensure identical processing. There are four sets of critical buses, and each GPC is connected to each bus via the IOP. Typically, a GPC *listens* to all the buses but *commands* only one. Connected to these buses are the MDMs (Multiplexor/Demultiplexors), which interface to the Shuttle's sensors and effectors. In the typical mode of operation, a single GPC sends a command to an MDM and then all of the GPCs receive the data in response to that command. Because there are never two masters of the same bus, two GPCs never try to do I/O on the same bus at the same time.

There's a special feature in the IOP itself that handles the differences between the commanders and the listeners. If the transmitter is turned on when a GPC issues a receive data command, the IOP assumes that the processor has already sent the command and immediately starts looking for the data. If the transmitter is not on, it assumes that another computer is going to send the command, looks for the command on the bus,

and then starts its time-out waiting for the data. We initially synchronize on the command going out and then later on the data coming back, which are received in all of the computers simultaneously. Thus, all of the computers get all of the data at the same time.

DG. **Are the display keyboards one kind of sensor?**
Spotz. Yes. As one computer polls the on-board display unit for keyboard entries, all of the computers get the data that come back, so all see the keystrokes from the crew members.

DG. **How do you determine which GPC is the commander of a particular bus?**
Spotz. That's controlled by a table in the software that the crew can update. Normally, though, when there are four GPCs functioning, the first commands string one, the second commands string two, etc.

AS. **What about synchronizing the computers between I/Os?**
Killingbeck. I/O requests have to be synchronized closely enough for each listener to issue its receive command early enough to hear the commander issue its command. We synchronize the computers with software first and then start the I/O. We have also separated the commander and listener IOP code. The commanding processor's IOP has a channel program that starts with a fairly sizable delay to ensure that the listeners issue their receives first. So, even if we start the listener's channel program a little bit late, it will have time to reach the receive before the commander ever sends out its command.

AS. **You must have some complicated techniques for dealing with I/O errors.**
Spotz. Just consider flight control. It has an input that occurs 25 times a second. That input is simultaneously sampling sensors on eight data buses. We have to be very careful to read the three inertial measurement units (IMUs) at the same time so we can make a meaningful comparison of their values. Each of these transactions is really an I/O (channel) program that the IOP is executing.

Potentially, we can have I/O errors on any or all of the eight buses. Each computer has to know what the other computers' perceptions of errors are—if any one computer rejects data because of an error, the rest of them have to reject those same data.

AS. **How can you know that all of the computers have picked up an error that occurred in only one of them?**
Spotz. First of all, when the I/O completion occurs, we have to synchronize by exchanging completion codes on discrete intercomputer lines that are separate from the data buses. There are two codes that are used to synchronize at I/O completion. One indicates a normal completion, the other an error. If no computer indicates an error, there's no problem. If any computer indicates an error, then more detailed information is exchanged. In any event, the erroneous information must be ignored by all of the machines if they are to stay in sync. On a second consecutive error, the element causing the problem is bypassed if at least two computers see the same error. If only one computer sees it, that computer is removed from the redundant set.

Macina. Since I/O errors use processing time that might otherwise be used for applications execution, cyclic processing delays can result in extreme cases with intermittent I/O errors. This brings about an interesting situation. Say there were I/O errors that took up so much processing time that flight control couldn't finish a particular cycle before it was supposed to start a new one. This is what you were suggesting a minute ago when you asked what would happen if flight control didn't run on time. In this situation flight control would finish its current cycle and then skip the next cycle. This would temporarily off-load the system until the operating system could clear the I/O error condition.

AS. **Does this happen operationally?**
Spotz. It's a graceful degradation, but it doesn't really happen in the current system. The ALT program had a lot of problems with loading, because our CPU utilization was over 90 percent. Almost anything caused cycles to skip. The only problems we see now are in the simulator, and those are due to differences in the simulator displays. Instead of updating a display every half a second, it only updates every second.

DG. **How do you use redundant sensors?**
Macina. Redundant sensors are connected to separate MDMs for input data. For example, each rotational hand controller (stick) is connected to three buses. Each unit has three transducers and three wires that measure deflection. Each of those wires is connected to a separate MDM. Each MDM is tied to all GPCs via a separate bus of the data bus network. Therefore, each individual sensor box has its own data path to every GPC.

The redundancy management software in each GPC contains algorithms for selecting the appropriate measurement from redundant sensors. The selection is based on the number of available sensors and various comparison algorithms like mid-value selection.

AS. **How are actuators controlled?**
Killingbeck. For the aerosurface actuators, each of the four computers sends out an independent command on an independent bus. With no failures, the commands should be identical. The voting is done at the actuator using a hydraulic voting mechanism, called a force-fight voter. In it, there are four hydraulic ports called secondary ports, each commanded by one of the four GPCs. The secondary ports go into the primary ports, which are heavy-duty actuators that connect to what's called a "summing bar," which is no more than a mas-

sive steel rod. If there are three good computers and one bad one, the three good commands physically outmuscle the fourth. This limits the control authority a little bit—we don't get the total force we'd like to get, but there's still enough power to control the vehicle. If you have a large enough pressure differential for a large enough time, the port is hydraulically bypassed, which relieves the pressure in that one port. The remaining three ports then regain their full authority.

Macina. This voting is important, since a computer is never allowed to turn itself off or turn another computer off. The summing and bypass occur at the actuator; the bad computer continues to operate as if it were still controlling the vehicle. The communication and listen mode synchronization may be broken between the bad computer and the other three, but the bad computer still has control of its port and still issues commands on that data channel.

DG. **Which actuators vote, and when?**
Spotz. The thrust vector controllers vote during ascent—these control the pitch and yaw of all the engines. The aerosurface actuators (the elevons and rudder) use force-fight voting during entry. The OMS (Orbital Maneuvering System) engines, which are the on-orbit maneuvering system, also vote, although they use a somewhat different technique.

Macina. The master events controller is the other voter. It's the device that handles pyrotechnical functions like firing and separating the boosters. The GPCs issue commands to this device, which in turn ignites the appropriate initiators. Voting in this case is an electrical process—important functions will not take place without two or three concurring votes. Booster ignition at lift-off, for example, won't take place without three concurring votes. After lift-off, however, the booster can be separated with only two concurring votes.

AS. **We'd like to know why you used the voting strategy you did. Why didn't you use hardware voting at a much finer granularity, for instance?**
Killingbeck. We didn't use hardware voting because it would have required a very high bandwidth data bus. We were worried about physical damage with the processors close together, especially after that explosion back on Apollo 13. That's why the computers, the sensors, and the data buses are all physically separated on the vehicle. That's one reason why we don't have a single, highly redundant, highly reliable processor.

To understand our design, you have to look back to around 1970, when it was first devised. This was before distributed systems started getting a lot of attention from the universities. NASA had come to the conclusion that too much money had been spent on the Apollo, Mercury, and Gemini programs for analyzing reliability when in practice almost nothing ever failed. They wanted a system that could tolerate three failures in any subsystem. This is called the fail-operational/fail-operational/fail-safe approach. It enables us to withstand two failures in the same type of system and still survive a third failure in an emergency. For pragmatic reasons, this was relaxed to fail-operational/fail-safe. In essence, we have three or four versions of all of our vital systems. Only nonvital functions are duplexed or simplexed.

Now, with four computers, we had trouble with the fail-operational/fail-operational/fail-safe approach because of our reliance upon majority voting. The fifth computer was brought in to ensure that voting would still be possible after two failures. When the requirements were later changed to fail-operational/fail-safe, the fifth computer was used for the BFS.

DG. **Does anybody in your organization ever look at the backup system's software, or is that specifically prohibited?**
Macina. It isn't specifically prohibited. We need to understand the interface between the PASS and the BFS. About 20 minutes prior to launch, we actually transfer data to the BFS. This is called a one-shot transfer. During the flight we periodically send it data that would not otherwise be available to it. There's constant communication, which means that we need to know something about their software. Their requirements are nearly the same, but their implementation is totally different. For example, they have a synchronous, time-slice type operating system, whereas ours is priority driven.

DG. **How did you choose this particular kind of system operation?**
Killingbeck. We originally looked at three redundancy management schemes. First, we considered running as a number of totally independent sensor, computer, and actuator strings. This is a classic operating system for aircraft—the Boeing 767, for example, uses this basic approach. We also looked at the master/slave concept, where one computer is in charge of reading all the sensors and the other computers are in a listening mode, gathering information. One of the backups takes over only if the master fails. The third approach we considered is the one we decided to use, the distributed command approach, where all the computers get the same inputs and generate the same outputs.

AS. **How did you make your decision?**
Killingbeck. With the independent strings, there are problems with flight control. Suppose we were approaching one of the critical points in the mission: orbit insertion or landing. Error tolerances are getting very small as these points approach. Suppose we were running two separate strings, one that thinks the altitude is 100.5 miles and one that thinks it's 99.5 miles. That difference doesn't seem significant, but if one of the commands that's going to the vehicle is commanding a pitch-up and the other is commanding a pitch-down, the voting actuators are going to have problems.

The two commands are going to cancel each other,

and as engine cutoff time approaches, one string will be calling for full pitch-up and the other for full pitch-down. The vehicle will just keep going right through the middle. The same kind of thing can happen on the approach to the touchdown point. If one string thinks the vehicle is a few feet high and the other thinks it's a few feet low, the same kind of pitch-up, pitch-down situation exists; only this time the vehicle is probably going to crash.

Macina. Another problem with this kind of system is that it's not very fault tolerant. If we lose a computer, we lose all the sensors and effectors connected to it.

Killingbeck. There are approaches to the instability problem that involve equalization and periodic exchanges of data—some kind of averaging, middle select, or whatever, to keep things from getting too far apart. The problem is that, for every sensor, an analysis has to be made of what values are reasonable and how an average should be picked. The extra computation consumes a lot of manpower and time, and creates a lot of accuracy problems. It's very hard to set a tolerance level that throws away bad data and doesn't somehow throw away some good data that happen to be extreme. It wasn't so much that we felt that this scheme couldn't be made to work, it's just that we believed there had to be a better way. Of course, we recognized the advantages of this approach in terms of recovery time. If the computer fails, one whole string goes out and the rest of the system keeps going.

The next most obvious approach is the master/slave approach. The difficulty with this approach arises after the master fails. Assuming we could detect the failure instantly, how would we get control to another computer? There's about four-tenths of a second of reaction time in certain critical phases from when the master fails until the vehicle starts misbehaving and losing control. One especially delicate phase is the final flare at touchdown when it's necessary to make an extremely rapid excursion of the elevator and to touch down at the right rate. If there's an elevator hard over at that point, the vehicle's either going to pitch way up and stall or pitch down and dive. There's no time for manually appointing a new master under these conditions.

The second critical point is during the ascent phase. At about 60 seconds into the flight, there is a region where air pressure on the vehicle peaks. If a failing master commands all of the actuators to swing over to their stops, there would only be four-tenths of a second to get the actuators back to the center. If we couldn't recover in time, the vehicle would literally be blown over and would break up. Again, there would be no time to manually appoint a new master.

We would have to arrange for automatic switchover, which would be complex. How do you know a bad computer won't jump into the automatic switchover code and preempt command of the vehicle by mistake?

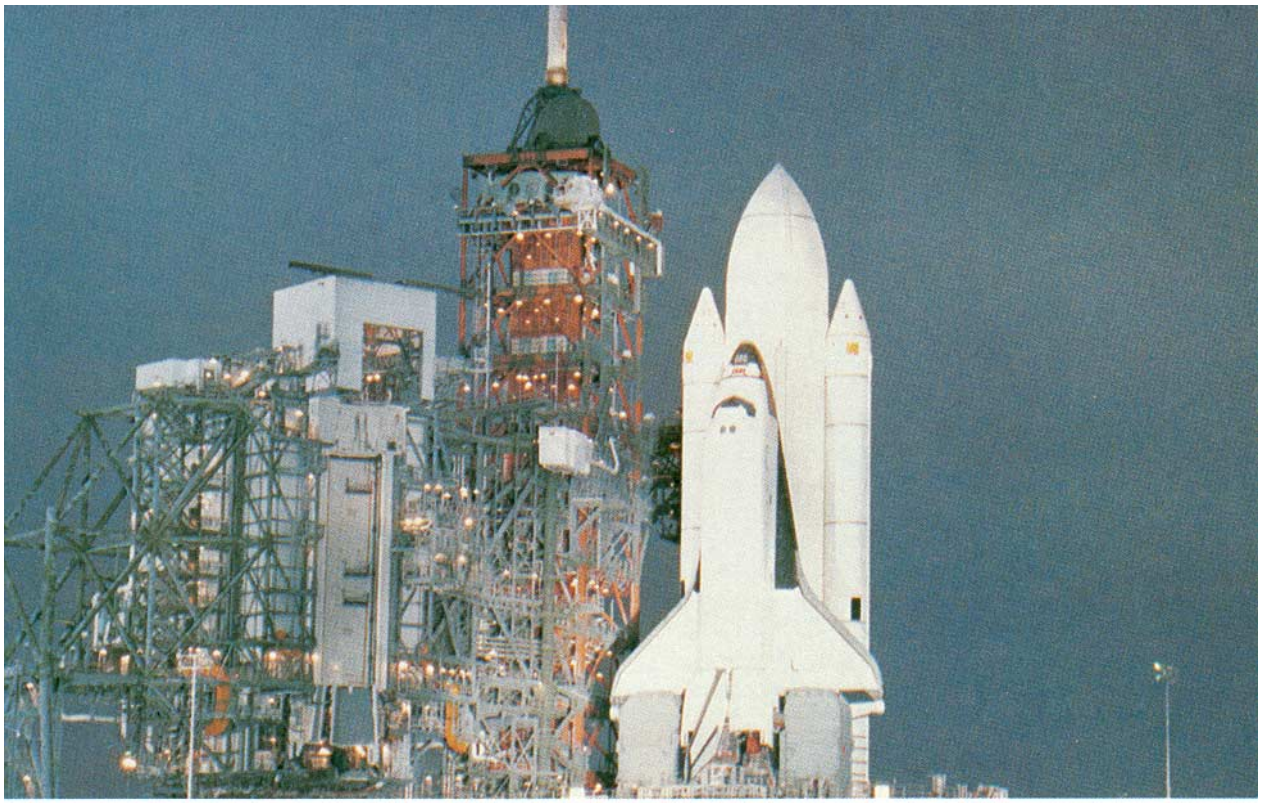


FIGURE 5. The Shuttle on the Pad Prior to Launch

You may not believe that this is a reasonable type of failure, but there's a lot of concern about this type of thing happening. We were positive that we didn't want any automatic switchover code. Therefore, a master/slave system was out.

Spotz. What we were left with is our current system—distributed command. This system gives us real flexibility: We can reassign data buses, if we need to, through keyboard entries. First we modify the data bus assignment tables in memory, and then we dynamically reassign those data buses between I/O operations. The next I/O transaction picks up with all of the good sensors so there's more recovery possible. We're somewhat reluctant to use this feature because it causes one computer to command two outputs and gives that computer 50 percent control over the vehicle (instead of the usual 25 percent).

There aren't the divergence problems of the independent strings or the unacceptable recovery time of the master/slave approach. When a computer fails, there's virtually no perturbation to the control of the vehicle. The remaining set does take four to five milliseconds to decide that the other computer is not there any more, but that's not a problem. We can also reassign data (via keyboard entries) to permit continuing use of sensors following a GPC or IOP failure.

AS. **Can you explain what happens in a "fail to sync"?**
Macina. A fail to sync is essentially a break up of the redundant set into two or more parts: The "bad" computer, which may just be a slow computer, considers itself the only good machine. It disables all communication with the other three computers but continues processing sensor data and issuing outputs on the buses it commands. The other three computers stop communicating with the bad computer and also continue using the sensor and output paths assigned to them. At this point there are essentially two computer systems attempting to fly the vehicle. Since the three-computer set is in the majority, the various actuators ignore the one machine and respond to the other three. Crew procedures don't allow this condition to persist for more than a few moments—the failed computer is powered off.

DG. **Have there been any unusual fail to syncs?**
Killingbeck. We did have a slow failure during an ALT flight, the first time we dropped the shuttle from a 747. You'd like to think that when a computer quits it just quits. In this particular case, though, a computer interspersed 12 I/O errors among some good I/O before it failed. The whole process took about four-tenths of a second—ten flight control cycles. The computer had a cracked solder joint that was opening and closing because of the high acceleration rate, and good data were intermittently interspersed with the noise.

AS. **Because of that it was too slow and missed a sync point?**
Killingbeck. Well, no. It was getting to the sync points but saying it had I/O errors. In fact, because it was commanding certain sensors and the commands weren't going out, all of the computers were having to deal with I/O errors of various types. We got a couple of cycle overruns, and finally, after about four-tenths of a second, the bad computer was isolated and removed from the set and everything recovered. The crew then powered it off and flew to a successful landing.

We now have a test case called "Free-Flight One," which we've used throughout the OFT (Orbital Flight Test) development. It uses massive I/O errors to determine whether the remaining computers can recover.

DG. **How quickly would the astronauts have to switchover from the PASS to the BFS?**
Spotz. In the worst case, there would be a 400-millisecond window where, if the whole primary went down and had previously commanded hard overs on the thrust vector controllers, we would lose the vehicle if they couldn't switchover in time. This would be at maximum dynamic pressure, shortly after lift-off.

The general rule, based on observations from the crew trainer, is that, if they can't switchover within about 10 seconds, they needn't bother. There are physical reasons for this: One is that the integrating accelerometer registers overflow in about 10 seconds at three gravities. After 10 seconds, they would have lost 1000 feet per second of velocity.

As long as the BFS is able to listen to all of the data the primary is getting, there's no real constraint unless the primary issues a hard over command improperly. If the primary stops issuing I/O commands, the crew has about 10 seconds from that point.

AS. **How will they know what to do?**
Macina. There are any number of ways: anomalous vehicle dynamics, Xs on their screens, multiple failures to syncs, etc. The crew is trained extensively on these things.

AS. **How would they engage the backup?**
Spotz. By pushing the red button on their hand controls.

DG. **Is there any single point of failure in the hardware system that could affect all five computers? A clock, maybe, for synchronizing all of the computers for data bus access, or anything of that sort?**
Killingbeck. There's not supposed to be. The system does not run synchronously at the hardware level. The closest we have to a central clock is the master timing unit. It's an atomic clock that we read about once a second in order to calibrate the computers' oscillators. This prevents long-term drift. In between those one-second points, the computers are totally on their own as far as time reference. We've got a complex strategy

for recovery should the master timing unit fail. This system is not centrally synchronized. We're only synchronized at the software level.

DG. **Are we correct in assuming that both the primary system and the backup are written in HAL and that they use the same compiler?**
Clemons. Yes. Both use HAL/S, although they might be using different versions of the compiler at any given time. If you're wondering about a generic compiler error in both the PASS and the BFS, we did have failures like that way back when we were developing the ALT program. We wrote about a half dozen or so discrepancy reports against the compiler.

Macina. I don't remember any that were catastrophic—they were mostly just nuisance problems.

Clemons. There hasn't been much to worry about on that account. I think it's because we test the compiler indirectly by testing every change in our simulator to see that it conforms to the requirements and the design. We also try to stay on the same version of the compiler all the way through the development, verification, and flight cycle. We won't switch to a new version of the compiler in midstream.

Occasionally, we get nuisance problems in the compiler. Once, we saw some matrix operations that didn't work under certain sets of conditions, and we had to go back and do a full audit of all uses of that operation in the compiler to make sure that there weren't any other exposures. We've never found any other problems. The compiler has not been a source of errors. It's also to our advantage that the source code in the backup flight system is different. It's highly unlikely that there would be compiler problems in two sets of separately developed code at the same time.

DG. **Could the Shuttle be flown without its displays, if absolutely necessary?**
Macina. It would be very difficult to fly a complete mission but not impossible to land the vehicle. We learned this during an ALT simulation (in the FSL) when we ran a test to see how our priority-driven operating system reacted to high CPU utilization. We put a pilot at 20,000 feet and began to artificially steal processing resources from the computer so the lower priority tasks couldn't operate. One of these was the display process. We ran the CPU up to an effective 120 percent, which left only flight control operational. With just this task running, the vehicle continued to respond to the pilot's commands, although the displays were dead. With the array of aircraft instruments, some of which were operating at reduced efficiency, the pilot was able to take the vehicle from 20,000 feet and land it manually, although he had a little difficulty and had to rely on visual clues. We can presume, then, that even if there were a problem that disabled the displays, the Shuttle could return if the failure occurred after the software for entry had been loaded into the computer and if the vehicle was low enough in the entry trajectory.

Clemons. One other point: All the data that are driven on the displays are also put on the downlist. The ground system uses that information to construct similar displays, and there are equivalent commands that can be issued to their displays and control systems. They could issue commands through the uplink in an emergency. It's possible to go through all of the ascent and on-orbit operations from the ground. The entry could be flown automatically, since there is a set of displays that are driven totally independently of the flight software itself. Once the pilot gets low enough, he can fly the Shuttle manually the rest of the way.

There's another kind of generic failure that hasn't been mentioned: a generic flight computer problem. After all, the backup flight computer is the same as the primary. We haven't ever encountered this kind of problem, though, in our 10 years of experience with these machines.

We could have put the backup computer's software in a different manufacturer's computer so that all five machines wouldn't be identical, but we decided that this reduced our flexibility and that it just wasn't necessary.

SYSTEM PROBLEMS

DG. **I understand there was a problem with the ascent trajectory on STS-1.**
Macina. That was a vehicle performance issue: We were given a set of initialization loads for shaping the ascent profile to minimize loads while still getting the vehicle into the correct orbit. These data are based on knowledge of the boosters, the main engines, etc. The initialization data given to us in this case didn't provide adequate capability to perform a return-to-launch-site abort. This was determined by an engineering analysis of the data after the flight. That has nothing to do with a software deficiency. The problem was corrected on subsequent flights with a different initial load.

DG. **It must be difficult to find errors like that.**
Macina. The difficulty is that you can't simulate the vehicle perfectly prior to flight.

AS. **We've heard a lot about a synchronization problem you had on the first Shuttle flight. Could you describe that for us?**
Spotz. The symptom of the STS-1 problem was the inability of the BFS to receive cyclic data from the PASS GPCs. This isn't strictly related to PASS synchronization, since the BFS monitors PASS I/O transactions at predetermined times (called BFS listen windows). At approximately T−17 minutes in the countdown, the BFS was moded to its ascent program and started listening to the PASS I/O transactions. All flight-critical sensor data were received properly, but the uplink transactions had errors, so the corresponding BFS buses

for the transactions were downmoded (marked as not being "tracked"). Since the BFS establishes its listen windows at the transition into the ascent program, it was moded back to its idle state and back again to the ascent program in an attempt to clear up the problem. The problem persisted.

Dumps were taken of both the BFS and a PASS GPC, and from the PASS and BFS error logs, we quickly determined that the PASS was initializing the uplink I/O on the wrong cycle relative to other transactions monitored by the BFS. (This was determined before the launch was scrubbed, even though the information didn't get disseminated until later.) Since all "BFS listen" transactions are also reestablished in the PASS upon memory overlays, the primary system was moded back to the prelaunch checkout program and back again to the ascent progam. The problem persisted here, too. Someone suggested we completely reinitialize the PASS, which as it turned out would probably have cleared up the problem, but this wasn't considered a safe procedure with the vehicle fully fueled. Also, a full explanation of the problem, not just its convenient disappearance, was required for a launch commit.

As it turned out, the problem had occurred at the initialization of the first primary GPC about 30 hours earlier, and had no major symptom other than the BFS tracking problem for the entire countdown. Actually, there was one other subtle symptom that turned out to be quite useful in proving that the problem really occurred over 30 hours prior to when the dumps were taken. This symptom was the loss of "pseudo-sync," or the expected timing between the GPC downlink data and other telemetry data. It wasn't until we suspected a process and phasing shift at initialization that we discovered that this relationship had not even been established for a single downlink cycle. The ground systems hadn't noticed the shift either, since this phasing isn't critical to their processing.

After the launch was scrubbed, we started back through the dumps. The operating system time queue (which is used to initiate cyclic processes) showed that not only the uplink process, but every process that was initiated by a mechanism known as phase-scheduling, was occurring on the wrong 40-millisecond cycle. Other processes, primarily the guidance, navigation, and flight control processes, used a different scheduling algorithm that guaranteed execution on their proper cycles, and these processes determined the profile for all I/O except the uplink. The two scheduling algorithms should have been consistent, and in tens of thousands of hours of testing, we had never seen an inconsistency. Several theories about possible causes were proposed, but finally Lynn Killingbeck realized that the apparent shift in all phase-scheduled processes could actually have been a shift in the initial scheduling of the system interface process—the process that phase-scheduled processes are phased from. The failure to establish downlink pseudo sync confirmed that the problem had occurred at initialization, and that the system had not somehow " slipped a cog" during countdown. This was important since a failure later, during redundant set operation, would have implied some single-point failure or a generic software problem. Even though the exact mechanism of the initial shift was not deduced and proved until Sunday morning, after Columbia was in orbit, enough data were available to determine the nature and probability of the problem, and to assure NASA that a safe mission could be flown.

The specific software problem had its roots in the assumption that the system interface process was always the first process to be scheduled—the timer queue should have been empty, except for the system interface, during computer initialization. As we later realized, this wasn't always the case.

The assumption was true when the scheduling algorithm was first implemented for ALT, but subsequent changes to the bus reconfiguration and initialization software invalidated it. After ALT, the bus reconfiguration software was changed to allow more crew control of the DPS configuration and to provide intercomputer cooperation for the display and launch data buses. These changes involved a delay, which required a time queue entry. This violated the assumption of an empty timer queue when system initialization was scheduled, and a potential process phasing anomaly was introduced.

Fortunately, this first change didn't expose us to the problem, even though it violated the "empty queue" assumption. It was a second later change made to expand the delay from 50 to 80 milliseconds that created the timing that eventually exposed us to the problem. After the second change, we had essentially a 1 in 67 chance of it happening each time we turned on the first GPC of the set. We had opened a 15-millisecond window within each second when there could be another process on the timer queue when the uplink process was being phase-scheduled. It was this 1 in 67 probability and the knowledge that the problem only occurred during initialization that allowed NASA to turn the computers off and then back on, and to fly the mission two days later without making a software change to correct the problem.

Macina. The potential for the problem—the first change—was introduced in 1978. The delay was changed from 50 to 80 milliseconds in 1979, which opened the window. It took us until the first flight, in 1981, to find it, or rather, for it to find us. We tested all through this area, but no one ever got lucky.

AS. **Which of your test systems might have found the problem?**

Macina. The SAIL was the most likely since it has a real PCMMU, and, as we determined later, the problem only occurred when you were using real PCMMU hardware. Simulator models don't have enough fidelity to produce the right timing. Even the SAIL simulations were limited since they're usually initiated from "checkpoints." This saves the time involved in having to initialize the computers each time and fly to the

point in the trajectory where the testing is to be performed.

DG. **Did you reproduce the problem after the fact?**
Eiland. Yes, the SAIL recreated it. They dedicated a full eight hours to just turning computers on and off. The system failed on the 75th run.

THE INTERPROCESS VARIABLE PROBLEM

AS. **What is your perspective on the structure of your system now that you've been intimately involved with it for a long time? Are there things that you would like to do differently?**
Macina. Certain design decisions we made early in the program have cost us a lot in maintenance. Things are under control now, and we can live with what we have. So the answer would have to be "yes," there are some things we would like to do differently.

Clemons. The fundamental design decision that's worried us the most and caused literally man-years of analysis is in the area of variables accessed by multiple processes within the software—so-called interprocess variables. Recall that we have a priority-driven executive with different priority processes passing thousands of parameters back and forth among each other. Remember also that each machine is slightly out of sync in the redundant set, although by no more than three or four milliseconds. Consider two machines in a redundant set where two processes are redundantly computing some variable. On one machine, a process computes a new value for the variable but is then interrupted for a higher priority process that uses that value. On the second machine, the process also gets interrupted, but before it has computed the new value. The two higher priority processes in the two computers that began to run after the interrupt have different values of that variable and may perform different functions. If their execution paths are sufficiently different, they may fail to synchronize.

Macina. Remember: To provide identical outputs, processes have to receive identical inputs and execute instructions in the same order. In this case, the second process is not getting identical inputs in both machines, and their outputs may begin to diverge.

Eiland. I don't think the original design team realized how many variables would be shared by separate processes. The HAL language actually had a capability to protect those variables with what we call an update block. In the system's original form, we incorporated these update blocks. But, as design and development continued, we realized that multiple processes required many more variables than could conveniently and economically be passed within update blocks.

DG. **What do these protected blocks do?**
Eiland. Any update block is a lock group. In the implementation, the update block notifies the operating system that a particular process is using an element of a lock group, in order to prevent concurrent access to that group by any other process.

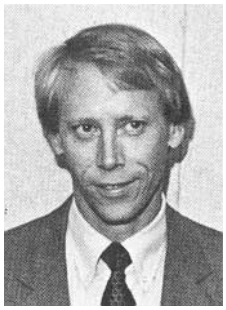
Once we realized we needed a cheaper protection mechanism, we implemented what we called a disable block to disable interrupts in the lower priority process while that process is accessing the interprocess variables. This was more efficient but still too expensive.

Clemons. Early on, our estimates of CPU utilization and core size showed there would be no room for update or disable blocks for every variable. We decided to protect only those variables that had to be protected and to analyze the remainder. We could provide alibis in the code to show why protection wasn't necessary. We have thousands of alibis.

Macina. But in light of what has happened since, the program might have been better off developing a faster and larger machine.

Clemons. Every time we make a design change, now, we have to consider these alibis, one at a time. Maybe we've changed the way a module works or the frequency in which it operates. Maybe protection wasn't needed before on a variable because a certain module never invoked another under a particular circumstance but would now.

We almost never have synchronization problems any more, so we should probably leave well enough alone. Our protection is good, but we're always trying to anticipate the effects of the next change. Local changes made by applications programmers may have system-wide implications, and the programmers may not be able to understand these implications.



BARRY EILAND
As manager of SOPC verification, Barry Eiland is responsible for planning the system test and integration activities for the Shuttle Operations and Planning Center, which will enable the Air Force to plan and control Department of Defense Shuttle missions out of Colorado Springs. Eiland joined IBM in 1974, and has held various technical and management positions on the on-board Shuttle software project. He was formerly with TRW, where he worked on the Apollo Program.

TABLE II. Explanations of Acronyms and Special Terminology

ALT. Approach and landing test programs. These were the tests that involved dropping the Orbiter from the 747 carrier aircraft from an altitude of 20,000 feet. They constituted the first operational use of the PASS system.

AP-101. There are five System/4 Pi Model AP-101 computers on board the Shuttle. Four are used by the PASS, and one by the BFS. The computer has 106K 32-bit words and executes about 450,000 operations per second.

BFS. Backup flight system. The BFS executes in the fifth computer. It can perform critical functions during ascent and entry if something catastrophic happens to the PASS. The BFS is independently programmed by Rockwell.

DEU. Display electronics unit. The DEU processes commands from the on-board computers before they go to the video displays.

Downlink. The downlink is the communication channel from the Shuttle to Mission Control.

Downlist. The downlist is the data sent by the DPS to Mission Control via the downlink.

DPS. Data processing system. This term designates the entire on-board computer system.

FEID. Flight equipment interface device. FEIDs are hardware components that interface the flight computers to a simulated environment within the SPF flight simulator.

GNC. Guidance, navigation, and control. These are a set of applications within the DPS.

GPC. General purpose computer. Any one of the five on-board computers.

HFE. High-frequency executive. This is a high-priority process that runs synchronously in the PASS at 25 Hz and controls the flight control application programs.

IMU. Inertial measurement unit. IMUs are navigational aids that measure accelerations on the vehicle.

IOP. Input/output processor. An IOP is one of two boxes that make up an on-board computer. It is used to interconnect with various buses.

IVV. Independent verification and validation. The IVV group for the Shuttle projects was completely separate from the development group, so that testing would be as objective and as thorough as possible.

LDB. Launch data bus. The LDB connects the data processing system to the LPS.

LPS. Launch processing system. The LPS is the computer system at the Kennedy Space Center in Cape Canaveral.

MDM. Multiplexor/demultiplexor. The MDMs interface the GPCs to the Shuttle's sensors and actuators.

MVS. MVS is an IBM standard operating system used on large IBM mainframes.

OFT. Orbital flight test. This term designates any of the first four scheduled developmental space flights of the Space Shuttle.

On-orbit. The orbital phase of a Shuttle mission; the part between ascent and entry.

PASS. Primary avionics software system. The PASS is the software that runs in up to four of the five GPCs.

PCMMU. Pulse code modulation master unit. The PCMMU is an on-board hardware component that collects information to be transferred from the DPS to Mission Control.

Redundant Set. The four computers used by the PASS to achieve reliability during flight-critical phases of a mission are the redundant set. The computers are synchronized at the applications level and provide bit-for-bit identical output.

SAIL. Shuttle avionics and integration laboratory. The SAIL is a facility for integrating the Shuttle's hardware and software.

SDL. Software development laboratory. The SDL was the forerunner of the SPF.

SM. Systems management. SM is the software that does vehicle and payload monitoring and control while the vehicle is in orbit.

SMS. Shuttle mission simulator. The SMS is the simulator, located at the Johnson Space Center in Houston, that the Shuttle crews train on.

SPF. Software production facility. The SPF is the primary facility for developing, integrating, and testing software for the PASS and BFS. It comprises IBM mainframes, special interfaces devices, and actual AP-101 flight computers.

STS. Space transportation system. This is the formal name for the Space Shuttle. The first Shuttle flight was designated STS-1, and each succeeding flight has been numbered sequentially.

Uplink. The uplink is the communication channel from Mission Control to the Shuttle.

Eiland. Analyzing interprocess variables requires a good working knowledge of the entire system. As more and more of our most knowledgeable people move on to other things, it will get harder and harder to keep track of the implications of various kinds of changes.

DG. **Do you have any tools that work on source code that help you ascertain when your assertions are correct?**

Eiland. We have a static analyzer tool that we run for every software release. It identifies all of the interprocess variables, the processes that reference them, and whether or not they are explicitly protected. If it is determined that a variable is exposed, then it's necessary to identify why the design of the system protects the variable.

This tool determines the priority of all the processes and takes account of certain assertions. There may be two processes that can never run at the same time, for instance. With the disable blocks in the code, the tool can determine which variables are protected. This still leaves a set of variables that are not explicitly protected. We have to analyze those and put in manual cards indicating that these references are protected. If we change that software and invalidate an alibi, we get an exposed situation. The reanalysis that we have to do after every change becomes very costly.

Killingbeck. Some of the most difficult changes are to pointer variables, which HAL calls name variables. If there's a pointer that's indexing through a whole array of variables, our tool is going to lose track of those

names. It can see the primary name at the beginning of the chain, but it can't keep track of anything beyond that point.

AS. **Do you have any idea of the actual cost involved in trying to perform this extra validation every time through?**
Eiland. It was in terms of many man-years for the first five flights. The tool was absolutely necessary for identifying these variables and in many cases pointed out problems that required explicit protection. It was absolutely the only way to manage all the variables that we have.

Spotz. There was one instance where a major design change caused a problem that we didn't detect, which caused a fail to sync about 13 hours before the launch of the STS-5. Overnight, we were able to determine the circumstances that led up to this. The problem could only have occurred during reconfigurations, so it wasn't an ascent or an entry issue. We were able to determine that this problem was not going to affect the upcoming flight.

AS. **Where are the alibis physically stored?**
Eiland. We have a separate database for alibis.

DG. **Another alternative to the overall system design would have been processes that could send messages to each other to communicate information. Would you consider this kind of approach as an alternative to shared variables, if you were going to redo the system?**
Spotz. We would make every effort to minimize the number of shared variables. When the system was designed in the early 1970s, we expected only about 50–100 interprocess variables. This estimate was based on the guidance, navigation, and control systems. Navigation computes seven things for a state—three positions, three velocities, and one time—and passes them on to guidance, which converts them to the pitch, roll, and yaw information. Those three things go on to flight control. Today there are thousands of interprocess variables. The problem is with the interactive crew interface code, not flight control. There's an awful lot of on-demand asynchronous processing involved there.

CONCLUDING REMARKS

AS. **What could the computer science research and development communities do to make efforts like this simpler in the future?**
Macina. The most important area is software engineering. We need more structured ways of designing software and coding, and more automated methods for validating the correctness of software early in the life cycle. Ultimately, there should be less of a need for testing, which is a costly way of assuring software reliability. We're reasonably satisfied with the languages we have today; it's the rigor with which we apply them that's the issue.

AS. **What kinds of on-board systems do you see in more advanced Space Shuttles and Space Stations?**
Macina. I think the major goal for the designers of the Space Station will be more extensive distribution of both hardware and software functions. The Shuttle's DPS isn't really a distributed system—there are five centralized computers each performing all of the on-board functions. Although we were able to develop a reliable system, the centralization of hardware and software reduced the system's adaptability. The Space Station designers will be working toward a data processing system that is not only reliable, but that can be upgraded with only a local effect on hardware and software.

Acknowledgments. The editors would like to thank Betty Branick for transcribing the document, Robin Brewster for helping with the display plot, and the interviewees for their gracious cooperation.

BIBLIOGRAPHY

Carlow, G.D. Architecture of the Space Shuttle primary avionics software system. *Commun. ACM 27*, 9 (Sept. 1984), 926–936.
Garman, J.R. The "bug" heard 'round the world. *Softw. Eng. Notes 6*, 5 (Oct. 1981), 3–10.
Madden, W.A., and Rone, K.Y. Design, development, integration: Space Shuttle primary flight software system. *Commun. ACM 27*, 9 (Sept. 1984), 914–925.
Sheridan, C.T. Space Shuttle software, *Datamation* 23 (July 1978).
Trevathan, C.E., Taylor, T.D., Hartenstein, R.G., and Stewart, W.N. Development and application of NASA's first standard spacecraft computer. *Commun. ACM 27*, 9 (Sept. 1984), 902–913.

CR Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]—*real-time systems*; C.4 [**Performance of Systems**]—*reliability, availability, and serviceability*; D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, tracing*; D.2.9 [**Software Engineering**]: Management—*software quality assurance*; D.4.1 [**Operating Systems**]: Process Management—*synchronization, multiprocessing/multiprogramming*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance, verification;* D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems*; J.2 [**Physical Science and Engineering**]—*aerospace*; J.7 [**Computers in Other Systems**]—*real time*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software maintenance*
General Terms: Management, Reliability, Verification
Additional Key Words and Phrases: PASS, space shuttle, avionics system