acm

SOFTWARE ENGINEERING NOTES

An Informal Newsletter of the
SPECIAL INTEREST GROUP ON
SOFTWARE ENGINEERING

Volume 6, Number 5 October 1981

THE "BUG" HEARD 'ROUND THE WORLD

Discussion of the software problem which delayed the first Shuttle orbital flight

John R. Garman

On April 10, 1981, about 20 minutes prior to the scheduled launching of the first flight of America's Space Transportation System, astronauts and technicians attempted to initialize the software system which "backs-up" the quad-redundant primary software system.....and could not. In fact, there was no possible way, it turns out, that the BFS (Backup Flight Control System) in the fifth onboard computer could have been initialized properly with the PASS (Primary Avionics Software System) already executing in the other four computers. There was a "bug" - a very small, very improbable, very intricate, and very old mistake in the initialization logic of the PASS. It was the type of mistake that gives programmers and managers alike nightmares - and theoreticians and analysts endless challenge. It was the kind of mistake that "cannot happen" if one "follows all the rules" of good software design and implementation. It was the kind of mistake that can never be ruled out in the world of real systems development; a world involving hundreds of programmers and analysts, thousands of hours of testing and simulation, and millions of pages of design specifications, implementation schedules, and test plans and reports. Because in that world, software is in fact "soft" - in a large complex real time control system like the Shuttle's avionics system, software is pervasive and, in virtually every case, the last subsystem to stabilize. Software by its nature is the easiest place to correct problems - but by that very nature, it becomes a tyrant to its users and a tenuous and murky unknown to the analysts. Software is the easiest to change....but in change, it is the easiest to compromise.

The path to reliability in the Shuttle Orbiter spacecraft is through replication - replication of sensors, replication of effectors, replication of controls, computers, software, data buses, and power supplies. In fact, in order to satisfy a general Shuttle goal of "Fail Operational - Fail Safe" ("FO/FS") most components are replicated 4-deep - either literally (4 sets of hardware, etc.) or equivalently (alternate schemes substituting for one or more of the four). Four is the magic number for a very logical and intuitively obvious reason: FO/FS requires full operational capability after one failure, and a safe return capability after a second. It takes three to vote - so it initially takes four to still be able to vote after the first failure.

There are five onboard computers (called "GPC's" by everyone - with few remembering that they really were "general purpose") - four operate with identical software loads during critical phases. That approach is excellent for computer or related hardware failures - but it doesn't fit the bill if one admits to the possibility of catastrophic software bugs ("the bug" of this article certainly is not in that class). The thought of such a bug "bringing down" four otherwise perfect computer systems simultaneously and instantly converting the Orbiter to an inert mass of tiles, wires, and airframe in the middle of a highly dynamic flight phase was more than the project could bear. So, in 1976, the concept of placing an alternate software load in the fifth GPC, an otherwise identical component of the avionics system, was born.

That software system, plus an ingenious yet simple astronaut-managed control which permits only the fifth or the other four to have any control, is what makes up the BFS. Its development utilized the same requirements specifications, the same programming language and compiler ("HAL/S" - by Intermetrics, Inc., of Cambridge, Massachusetts, under contract to NASA) and the same target computer. But it was developed by an entirely different and remote organization (Rockwell International, Downey, California, vs. IBM, Federal Systems Division, Houston, Texas, for the PASS) and used an entirely different operating system (developed by Rockwell and IBM, respectively - both under contract to NASA). A very thorough and complete interface specification was drawn up and a relatively straightforward operating technique ensued. The BFS, when not in control, would "listen" to all the input and some output data traffic to and from the PASS-controlled GPC's. In that way the "BFS" could independently "keep up" with the goings-on of the Orbiter and mission and be ever-ready to take over control of the vehicle when switched in by the astronauts. Moreover, it could do so without any dependency on a potentially failing PASS except for its dependency on the PASS to fetch information from sensors and crew for it to listen to. This concept created several implicit but fundamental changes in the overall Orbiter design and operation.

First, the BFS had to "keep tabs" on the PASS and "stop listening" wherever it thought the PASS might be compromising data fetching. For in no way could any failure of the PASS be permitted to "pollute" the BFS (thus potentially bringing down all five computer systems simultaneously).

Second, the BFS could only possibly remain in a "ready" state for a short period after failure of the PASS. If there is no data to listen to then only extrapolation and certain reinitialization techniques following switch-over would permit the BFS to take successful control of the vehicle. As such, astronauts were trained to make their choices quickly.

Third, the PASS itself had to make some design and operating technique modifications. The PASS operating system is asynchronous and priority driven (i.e., the most important task is always given immediate control of the computer, resulting in many interruptions and process "swaps"). Yet four different computers are able to keep in perfect process synchronization and maintain identical bit-for-bit data contents with no more than eight sync state codes, about 6% overhead, and some intermittent inter-computer data exchanges. The concept was difficult and the design was complex, but the result was a very "forgiving" software system - either to hardware failures ("FO/FS"!) or to many forms of intermittent software failures. An example of the latter is any condition which creates a temporary "CPU overload" (too much processing to do in too little time). This could be caused by software errors, simultaneous hardware failures (with the incumbent additional load of error processing to log what has happened, analyze it, and protect or fault-down the system where necessary), or even an unanticipated coincidence of normal processing. The priority-driven system "carries on" with whatever CPU time is available by "skipping" or delaying lower priority processing. By contrast, the BFS design is much more like a fully synchronous "time-slotted" system, where each process is given a fixed ration of time in which to execute each cycle. While perhaps more straightforward to validate, it carries less flexibility and "forgiveness". Unfortunately, asynchronous and synchronous systems don't "mix" well. For the BFS to literally listen to all critical external data fetches by the PASS, the PASS had to either become, or pretend to become, "synchronous" on those processes. This was accomplished with as little compromise to the design as possible. In fact, the most direct consequence initially was an improvement in efficiency of the PASS. Virtually all cyclic processing was "scheduled" with respect to a cycle counter in a high priority system process. The result was an "apparent" synchronous operation of all critical processing (though process interruption and swapping are yet rampant, and properly so) and a consolidation of much of the input/output processing on the 25-data bus network since the time phasing of most processes became constant and always predictable. Still, the PASS, under enough "stress", is prone to "misbehavior" - with asynchronism escaping to the forefront. This is quite proper behavior to the PASS (and usually successful), but it could leave the BFS with a nervous breakdown and the crew with a dilemma (which is normally to "stick with the PASS" since switchback from BFS to PASS is not possible). It is this dichotomy in design, an insolubility lurking between the PASS and BFS that was one of the roots of the launch delay. The changes to the PASS to accommodate BFS happened during the final and very difficult stages of development of the multi-computer software. The issue here was that not quite all the PASS cyclic process start-ups were converted to be done with respect to the cycle counter (i.e., the emulated synchronism). However, without "the bug" it would have done no harm, and except for consistency alone, initiating all processes with respect to the cycle counter was not a requirement.

Fourth, and the final fundamental effect of incorporating the BFS, was that the system became even more complex. Another subsystem, especially one as intricately woven into the fabric of the avionics as is the BFS, carries with it solutions to some problems, but the creation of others. While it certainly increased the reliability of the system with respect to generic software failures, it is still argued academically within the project whether the net reliability is any higher today than it would have been had the PASS evolved to maturity without the presence of its cousin - either as a complicating factor...or a crutch. On the other hand, almost everyone involved in the PASS-side "feels" a lot more comfortable!

Even "the bug" itself is complex and buried deep in the PASS. In fact, it wasn't until IBM really began examining "dumps" of PASS GPC's an hour or so after the problem initially occurred that anyone thought it could be anything but a BFS problem (after all, it was the BFS that refused to "sync" up - start listening - to some of the PASS data traffic; and the PASS had been operating perfectly for almost 30 hours during pre-launch countdown and checkout of the vehicle). When the very first GPC is turned on and the PASS is loaded, it attempts to synchronize the start-up of all its cyclic processing with the cyclic output of telemetry data from the vehicle. While not a mandatory requirement, having a known and fixed phasing between cyclic computer processes and telemetry from computers and hardware systems alike makes the post-flight analysis of data an easier task. This synchronization is accomplished by reading the value of time within the telemetry system (which in turn was sampled from the same central clock that the GPC's sample for time), and calculating from that the phasing of the telemetry system with respect to the central clock. A start time which is both in the future and would result in PASS synchronization with the telemetry is then calculated. Finally, a high priority system process is initiated at that calculated time and all (well, almost all) other processing then or later is started with respect to that process and the cycle counter kept within it. Even the BFS, using the cycle counter passed to it by this process, becomes time-synchronized so that it knows when to expect PASS data fetches on the data buses.

Early on Friday morning, even before the launch was scrubbed, it became apparent that a few processes in the PASS were occurring out of phase from (one cycle early with respect to) all other PASS and BFS cyclic processing. One of these was the PASS periodic fetching of data ("polling") from the vehicle's two uplink processors (the devices from which the GPC's obtain command information from Mission Control). Since the BFS, to prevent "pollution", stops listening to all data on any buses on which it hears unanticipated PASS data fetches, it soon became apparent why the BFS couldn't "sync" up. To the BFS, the data fetches one cycle early were simply unanticipated "noise" from the PASS (with silence following where data fetches were

expected). Since the two uplink devices are located on "strings 1 and 3" (the four pairs of critical buses which hold most of the guidance, navigation, and flight control sensors and effectors are called "strings" 1 through 4), the BFS stopped listening. In fact, it never really started listening to anything on those two strings 20 minutes prior to the scheduled launch when the BFS was first turned on.

By early afternoon, and primarily by forcing all the experts into the same room and talking out the symptoms observed in computer dumps, most of "the bug" was understood. Whenever the operating system is asked to initiate a cyclic process with a start time in the past, it will "slip" the start time the number of process cycles required to put it in the future. The result is as if the process did actually start "on time" (much as an alarm clock - if set to ring in the past, it slips a 12-hour cycle so it can ring in the future). Since the start time is a calculated value, and since polling of the uplink device is one of the few processes started independently of this calculation, an error model involving a late calculation of start time was quickly constructed. And in fact, it was then verified (the vehicle GPC's had been left "untouched" to assist the effort) that in fact all PASS and BFS processing was late by one cycle with respect to telemetry. Uplink polling was on time and therefore simply appeared early (it was right and the rest of the world was wrong) on each of its cycles.

Nevertheless, we were not able to ascertain how the startup time calculation could possibly take so long that the calculated start time would be in the past when the calculation completed - that appeared to be impossible! However, we did know:

1) That it was low probability - never seen before on the vehicle or in labs (though it could often be masked in labs through the use of "reset" or restart points - avoiding the costly encumberance of "IPL" and initialization on each test).

2) That it was "latching" - if the phasing error were present, the system would work just as hard to maintain it across all possible GPC reconfigurations as it would to maintain the correct phasing if it weren't present. Thus if we would get the GPC's initialized with the proper phasing, there was no concern whatsoever that the problem could "pop up" during the flight.

3) That we could test for it- either by examing the PASS process and telemetry phasing, or simply turning on the BFS.

4) That we could correct the phasing - just like an old television set: if it doesn't work turn it off and back on again. This is something that couldn't have been done with the fully-fueled and armed vehicle on Friday morning - the GPC's play a critical role in processing information from and to the KSC Launch Processing System, and therefore in managing the vehicle on the launch pad.

All that information, plus the fact that recycling the GPC's Friday night did clear the problem on the vehicle, gave NASA management more than enough information to commit to a launch on Sunday morning.

But we still didn't know exactly how the problem had happened - and didn't until Sunday afternoon, about 8 hours after launch. Actually we felt quite fortunate in getting the final entomology of "the bug" that soon. This problem had all the characteristics of bugs that can take weeks of analysis to really "nail down". And what was the answer? Time! That nemesis of real-time systems and concurrent processing, that concept which though pervasive in our lives is so difficult to conceptualize in many contexts. The "obvious" occurred to one of the IBM analysts - since the calculation of start time couldn't possibly take so long that the start time would be in the past when the calculation completed, maybe it didn't! Maybe the GPC "thought" it did - "past" is a concept to humans, it's an equation in computers: the difference of start time and current time being negative. Multiple computers with identical software and data can't look at "clocks" to see what time it is. If they did, they would each get slightly, ever so slightly, different values. With slightly different values used in such operations as "fire pyrotechnics on the first cycle after 3:00", the computers might easily "sliver": some GPC's firing on one cycle, and some on the next (sampling 3:00.001 versus 2:59.999). To solve this dilemma, PASS GPC's utilize the operating system's "timer queue" as a clock. The top entry is the desired start time of the next cyclic process - and with hundreds of cyclic process executions scattered across any second, the top entry is always a fairly accurate reading of current time (always a little in the future, but always fairly accurate). Moreover, it is always bit-for-bit identical across redundant PASS GPC's. Of course, when the very first GPC is turned on there is no active processing. Since that's the only time the queue is empty, and since the queue is initialized with a known fill-pattern, a test for that pattern is the same as testing for "first GPC on". In that case, since no other GPC's are around, the GPC is allowed to use its clock for time.

.....But the queue wasn't empty! (Determined via lab simulation.)

About 2 years earlier, a change to use of a common subroutine to do some data bus initialization prior to the "start time" calculation was made. That the subroutine had a "delay" in it which placed a time in the timer queue went totally unnoticed....after all, what does a bus initialization routine have to do with a calculation to determine telemetry phasing? Moreover, at that time the delay was small enough that the top entry in the queue was close to current time - in the future, but close. That same subroutine is used elsewhere for bus initialization after certain system reconfigurations. The time

delay in it was such that a temporary CPU overload during critical flight control processing could occur. A "fix" to that problem was made about a year later - about a year before the flight. The delay time, "just a constant in the code", was increased slightly in order to place the subroutine's post-delay processing away from the flight control processing. That increase was enough to open a 1 in 67 probability window for a problem in the first GPC turned on. In its start time calculation, using that "slightly more"-in-the-future queue time as if it were current time, it could cross the "cliff" and "think" that the start time was a little in the past. The operating system would then slip that single system process by one cycle, which in turn would cause all (almost all) processing in all GPC's to be one cycle late!

But again, to testers and analysts, what does a constant in a bus initialization routine have to do with the calculation of start time in the first GPC? No "mapping" analyzer built today could have found that linkage. Testing might have. But the window wasn't opened until late in the test program (relative to this code), and even then, most simulations didn't go through the expense of initializing "from scratch". And even where they did, it would have to have been in a lab with a reasonably accurate model of the telemetry system plus a simulation or test involving both PASS and BFS, and it would still be fighting the low probability. Even then, the temptation would be to try again....and never be able to repeat it; and never be sure it wasn't a "funny" in the lab set-up....or a similar problem fixed by another software change. That, in fact, apparently did happen in one of the labs....about 4 months prior to the flight.

And then, on the day that the first GPC was turned on, 30 hours before scheduled launch, we hit the problem.....

The development of avionics software for the Space Shuttle is one of the largest, if not the largest, flight software implementation efforts ever undertaken in this nation. It has been very expensive, and yet it has saved money, saved schedule, and increased design margins time and time again during the evolution of the Orbiter and its ground test, flight tests, and finally the STS-1 mission. Since computers are programmed by humans, and since "the bug" was in a program, it must surely follow that the fault lies with some human programmer or designer somewhere--maybe! But I think that's a naive and shortsighted view, certainly held by very few within the project. It is complexity of design and process that got us (and Murphy's Law!). Complexity in the sense that we, the "software industry" are still naive and forge into large systems such as this with too little computer, budget, schedule, and definition of the software role. We do it because these systems won't work, can't work, without computers and software.

If there are lessons to be learned from "the bug", they must be in how we view ourselves and our task. Building software in a large system against fixed schedules is not conducive to "bug-free" products. We can minimize the errors, and we can minimize the flight criticality of the ones that remain (and "the bug" certainly wasn't flight critical) - but we can't treat it like a problem with a methodological solution. If it were, we would simply start following the rules of the many and respected software engineering theoreticians in this country and around the world. But even today, their approaches assume a relatively pristine environment for software development - an environment where requirements can be completed correctly prior to design, design prior to code and test, and code and test prior to verification, validation, and integration into the hardware system. In fact, an environment where the project is assumed to know what it wants out of the software in the beginning! Software development from my perspective is almost never that way. It should be, and we should try hard to make it so - but it is always iterative and incremental instead. Were we to force the assumed environment, it would be like the "tail wagging the dog" and I can assure you that STS-1 would be on the ground yet.

The lesson from "the bug" that I plea is directed to the academic and software engineering community: help us to find ways to reliably modify software with minimum impact in time and cost. Not perfect reliability, because projects will always back off to trade for time and cost. Maintaining software systems in the field, absorbing large changes or additions in the middle of development cycles, and reconfiguring software systems to "fit" never-quite-identical vehicles or missions are our real problems today. It's easy to say "don't break the rules". It's impossible not to without inverting the relative position of software in embedded systems - and that's wrong! Software may be the "soul" in most complex systems, but it is still just part of the supporting cast....a very flexible part.

John R. Garman

Deputy Chief,
Spacecraft Software Division
NASA, Johnson Space Center
Houston, Texas
August 24, 1981