# The Dangers of Over-Engineering a Safe System

## Building Functional Safety into Complex Software Systems, Part III

Chris Hobbs, Senior Developer, Safe Systems
QNX Software Systems Limited

## Abstract

Hasty attempts to deal with a specific safety issue without carefully considering the question of overall system dependability may lead to a great deal of work for little or no benefit, and the unwitting introduction of significant new problems.

For our discussion, we look at the hypothetical example of an in-cab train controller for an Automated Train Operations (ATO) system to be used in a Light Rapid Transit (LRT) system at a high-altitude location where the high neutron flux increases the threat of soft bit errors in DRAM. We examine the effect on dependability of adding software error detection to a 2-out-of-2 system, consider the benefits and adverse consequences of this additional check, and suggest some other approaches to improving dependability that might be effective.

**Elephants in the room**

In this discussion of safe software systems we are knowingly ignoring two large elephants in the room: security and in-use errors. For the sake of simplicity, we are assuming that the systems we are evaluating are absolutely secure, and that no person using them will ever use them incorrectly.

We know that, in practice, these assumptions are both incorrect and dangerous: a) a system cannot be deemed absolutely secure any more than it can be deemed absolutely safe; and b) people using a system will make mistakes, often seemingly random mistakes which fall outside all previously considered scenarios. In practice, then, we will need to incorporate security risks and human in-use errors in our evaluations of software system safety levels.

## Two tragic corrections

Two tragedies, one maritime, the other aviation, can illustrate how well-meaning but ill-thought solutions can precipitate the very tragedies they are meant to avoid. The first tragedy occurred in the Chicago River in 1915. The *SS Eastland* listed and rolled over, killing more than 840 passengers and crew. On its web site, the Eastland Memorial Society notes that, according to George W. Hilton, who wrote *Eastland: Legacy of the Titanic,*

> the Eastland's top-heaviness was largely due to the amount and weight of the lifeboats required on her. He explains that after the sinking of the *Titanic* in 1912, a general panic led to the irrational demand for more lifesaving lifeboat capacity for passengers of ships. Lawmakers unfamiliar with naval engineering did not realize that lifeboats cannot always save all lives, if they can save any at all.[1]

In conformance to new safety provisions of the 1915 Seaman's Act, the lifeboats had been added to a ship already known to list easily. As Hilton notes, lifeboats made the *Eastland* less not more safe. Other factors, such as the manner and speed at which a ship sinks may more significantly affect the number of deaths, as they determine whether the lifeboats can even be used.

The second tragedy occurred in Antarctica in 1979. Air New Zealand Flight 901 flew straight into a mountain, killing all 257 people on board. According to the New Zealand Transport Accident Investigation Commission's report, the computer flight plan for this flight "had been in error for 14 months.... This error was not corrected until the day before the flight ..." and "The crew was shown a copy of the erroneous flight plan with the incorrect co-ordinates ... but the flight plan issued on the day of the flight was correct".[2] Of the 10 contributing factors found by the Royal Commission charged with investigating the accident, only two were identified as "blameworthy acts or omissions": failure to supply the pilots with topographical maps of their intended flight path, and, especially failure to inform the pilots of the corrections made in the computer flight path. Not knowing of the corrections, the pilots thought that they were flying over McMurdo Sound, when in fact they were in fact headed into Mount Erebus. The Royal Commissioner, P. T. Mahon does not mince his words:

> In my opinion therefore, the single dominant and effective cause of the disaster was the mistake made by those airline officials who programmed the aircraft to fly directly at Mt. Erebus and omitted to tell the aircrew.[3]

As with the sinking of the *SS Eastland*, a correction of one threat (an incorrect flight path) inadvertently added stress to the larger system (correct navigation of the aircraft) and precipitated a tragedy.

## About safety claims

When we design a safe software system, one of our first tasks must be to determine its safety requirements. This means that we must determine:

- the system's required level of dependability; or, inversely, the acceptable level of system failure
- the limits of our safety claims; that is, the conditions and constraints within which we make our dependability claims

If we can demonstrate that within the limits of our safety claims our system achieves its required the level of dependability, then we can claim that the system is sufficiently safe.

### Reliability or availability?

For a software system, *dependability* is a combination of *availability* (how often the system responds to requests in a timely manner) and *reliability* (how often these responses are correct). Thus, a dependable software system is a system that responds with the correct answer, when it is required and in the time required. When we define sufficient dependability for a system, we must take care to understand both the required reliability and the required availability. The relative importance of availability versus reliability varies from system to system, depending on what the system is designed to do.

#### The bicycle paradigm

The bicycle is an excellent paradigm for thinking about dependability, because its safe use so obviously requires both reliability and availability. The rider must make the correct decisions about steering, speed, balance, etc. or the bicycle will run into something. Unless the rider is a circus performer who can balance a stationary bicycle, she must also keep the bicycle moving or it will fall over.

Imagine a riderless bicycle run by a software controller. The controller must make the correct decisions (it must be reliable) in order to get the bicycle to its

destination without incident, and it must do so continuously (it must be available). Any failure that forces the controller into a design safe state (which usually means stopping) for too long would result in a catastrophic failure, just as surely as would a failure that caused the bicycle to turn incorrectly. In fact, because our riderless bicycle must keep moving in order to stay upright, our controller has no design safe state that does not place the larger system (the bicycle) in a dangerous state.

## A simple safe system

The system we will use for our discussion is a very simple, hypothetical in-cab controller (for an equally hypothetical) ATO system running a driverless Light Rapid Transit (LRT) system. Figure 1 below illustrates this system. For simplicity, we have assumed that the train runs on a circular track, and we have shown the system checking only four values:

- the state of the train (moving or not moving)
- the time in the station (more than 90 seconds or less than or equal to 90 seconds)
- the state of the doors (open or closed)
- whether the train has entered the station (entered or not entered)

When it is initialized, our train begins in a stopped state, opens its doors, waits 90 seconds, closes the doors and moves on to the next station. When the train enters a station, it stops and opens its doors, waits, then continues on in an endless loop until it finishes its day. Its design safe state is “stopped”.

**2oo2 or 1oo2?**

According IEC 61508, Part 6 our system is a 1-out-of-2 system (1oo2), because one subsystem on its own can decide to move the system into its design safe state.

The difference in nomenclatures arises because IEC 61508 uses definitions inherited from hardware design, describing the decision architecture based on the number of votes required to move the system *into* its design safe state, while we use a definition, now common with software design, which describes the decision architecture based on the number of votes required to keep the system *out of* its design safe state.

### The controller

The controller is a 2-out-of-2 (2oo2) system; it has two processing subsystems, which must agree that it is safe to keep the controller out of its design safe state. (See “About the 2oo2 system” on page 6 and Figure 2 on page 7.) If either subsystem indicates that the controller cannot run safely, then the controller cannot run. A 2oo2 design increases the system’s complexity, but we consider that the consequences of the controller not working correctly are sufficiently dire to warrant the time and effort duplication requires.

For the purpose of this discussion, the precise criteria which this system uses to determine if it must move the controller to a design safe state are less important than the fact that whatever the controller does, it must do unfailingly. Opening the doors while the train is moving, or starting to move while the doors are open and passengers are embarking or disembarking may injure or kill someone. We will therefore assume that the system must be certified to IEC 61508 Safety Integrity Level 3 (SIL3), which means that the “probability of a dangerous failure” is less

than one in $10^7$ per hour of continuous operation. We are assuming that this requirement has been met in the original controller design, but that a new risk, which was not considered when the original requirements were written, has just been identified.

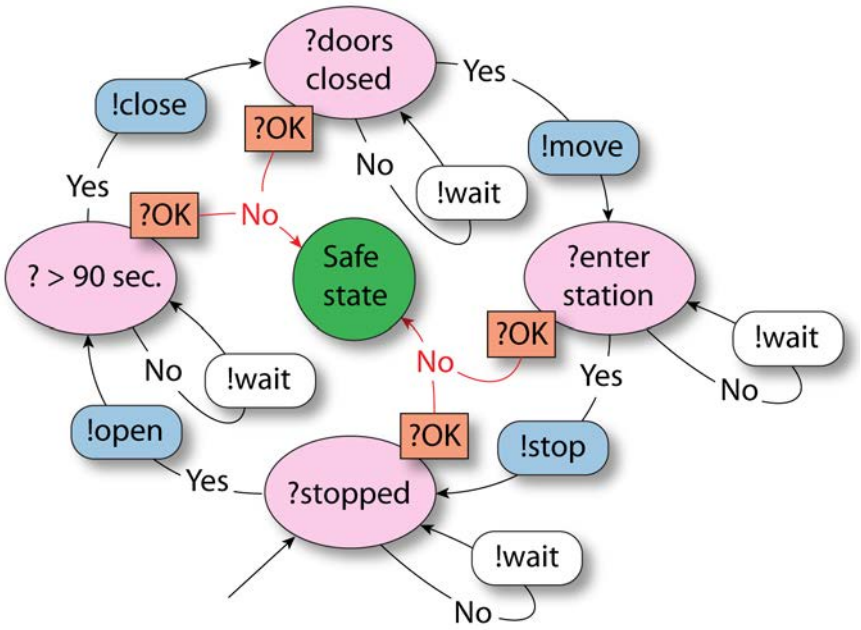


*Figure 1. A very simplified view of an ATO controller that moves to a safe state if it detects any dangers or anomalies. The system receives data from timers and sensors about the location of the train, its movement, the state of the train doors, and the time it has been in a station. It sends instructions to start and stop the train, and to open and close the doors. Before sending an instruction, it the checks with its 2oo2 system to determine if it is safe to continue.*

## A new vulnerability

The problem we face is that, though the effects of radiation on computer memory have long been known, when the original specifications for our system were written no one thought to include the threat of memory errors caused by cosmic rays.

### A change of context

Our hypothetical controller has already proven itself in Rome and several other locations. Now a new customer is considering it for an LRT ATO in the La Paz-El Alto metropolitan area in Bolivia. La Paz-El Alto has almost 2.5 million inhabitants living at an elevation that rises above 4,100 metres (13,600 ft.—higher than Mount Erebus). This is a significant change in context, because the threat of soft and hard memory errors caused by cosmic rays increases with elevation. The customer asks for proof that our system can still meet its safety requirements when the risk of soft memory errors caused by radiation is included in our dependability estimates. The increased altitude would also require us to revisit other design conditions, such as cooling, but in the interests of simplicity we will not consider these here.

We should start by congratulating our customer for recognizing that, whatever our system's previous successes, and however many millions of hours it ran without failure in Rome and elsewhere, changing the context of the system may have

invalidated some or all of the data on which its dependability claims were based. Specifically, cosmic rays have long been identified as a significant cause of soft memory errors, with neutrons the main culprit[4]. The greater the number of neutrons that pass through an area in a given time (relative neutron flux), the higher the risk that neutrons will cause a memory error. Neutron flux varies with location (chiefly but not exclusively latitude) and, especially, altitude. The Seutest[5] neutron flux calculator gives a relative neutron flux for La Paz-El Alto that is roughly 11 times that in Rome.[6]

### Errors and false positives

Our problem is twofold: memory errors may cause our controller to fail, and they may create false positives, prompting the controller to move to its design safe state unnecessarily. It is even possible that radiation will cause a soft memory error and affect the same bit in *both* processing subsystems at the same time causing the controller to fail and allow the train to function when it should be moved to a design safe state. It is far more likely, however, that a soft memory error in only one subsystem will cause a false positive in the 2oo2 system and provoke unnecessary controller and ATO shutdowns.

False positives in our controller may not compromise safety *directly*. The 2oo2 design that moves everything to a safe state in the event of any disagreement between subsystems creates a highly reliable system. However, dependability (and hence safety) also depends on system availability. Though availability appears to be less important than reliability in this system (a stopped train is usually less dangerous than a moving train) compromises to controller availability do in fact compromise overall safety:

- An LRT train that has stopped (due to a false positive in the controller or any other reason) will not be where it is expected to be. This may reduce the time and distance between trains, and increase the need for reliance on systems communicating and setting train locations, safe distances, etc., placing added stress on the wider system.
- If a system does not perform as required—if it is not available when needed—people find ways to make them work, often by circumventing the safety checks. For example, if a sensor intermittently reports one of twelve train doors open when it is shut, users may find ways to bypass the sensor in order to keep the train running, on the dangerous assumption that if 11 doors are shut the twelfth door must also be shut.

Given the increased threat of soft memory errors and their possible consequences in our system's proposed new context, we agree with the customer that we should look further into the effect of soft memory errors on our system's dependability.

## Software error detection

Since the problem is memory errors, it seems obvious that the solution is to add memory error detection to our system. Of course, before we do this we should be certain that this solution will

a) be effective

b) not compromise safety

What follows is a description of the controller, our assumptions about its 2oo2 subsystem and handling of memory errors, and our calculations of dangerous failure rates with and without the software error detection. The results of these

calculations will tell us if software error detection makes an appreciable improvement to our system's dependability, or if we should consider other methods to ensure that our controller is sufficiently dependable in its new context.

## About the 2oo2 system

The 2oo2 system that allows our ATO controller to move from its design safe state and perform its tasks running the LRT functions as follows:

1. Two independent processing subsystems receive the same stimuli (events) from the outside environment.
2. Each processing subsystems uses the events it receives from the outside environment to independently calculate whether the controller should move to its design safe state.
3. Each processing subsystem presents the result of its calculation ("Yes" or "No") to a gating subsystem (shown in Figure 2 below as an AND gate).
4. The gating subsystem compares the two outputs from the processing subsystems.
5. If both outputs agree that the controller may be kept out of its design safe state, the controller is allowed to continue running.
6. Under any other condition, the gating subsystem requires the controller to revert to its design safe state.

### Assumptions about the gating subsystem

We assume that the gating subsystem is approved to IEC 61508 SIL4. For a system operating in a low-demand mode, this safety integrity level requires that the subsystem will correctly detect the difference between the two outputs from the processing subsystems 9,999 times out of 10,000. For a high-demand system or a system in continuous operation, the SIL4 rating means that the probability that it will fail to detect a difference between the two processor outputs is less than $10^{-8}$ per hour of operation.

Assuming that the role of the gating subsystem is to actively hold the ATO controller out of its design safe state, the continuous mode of operation may be more applicable. However, for the purposes of this calculation, the more conservative low-demand mode numbers are used, because they assume a less dependable gating subsystem.

#### Error detection

The gating subsystem must detect a disagreement between the processing subsystems in two different circumstances:

- A non-recoverable memory error (multi-bit) has occurred in one of the processing subsystems, causing that processor to shut down.
- An application error (possibly caused by an application bug, possibly by an undetectable memory error) has occurred on one processing subsystem but not on the other.

In both cases the output from one or both of the processing subsystems will be something other than "OK", in which case the gating subsystem requires the ATO controller to revert to its design safe state.

While it could be argued that the gating subsystem should be able to detect the shutdown of a processing subsystem more reliably than an application error, for the calculation below we assume that both types of detection are only accurate 9,999 times out of 10,000. This is probably a conservative view but, as with the assumption of SIL4 for a low-demand system, it is used because it assumes a less dependable system. If a less dependable gating subsystem is sufficiently dependable, then a more dependable gating subsystem (SIL4 for high demand, greater reliability detecting processing subsystem shutdowns) will also produce a system sufficiently dependable for our purposes.

## Assumptions about the processing subsystems

We assume that, in the absence of any memory failures, the two processing subsystems in our 2oo2 system have a dependability one order of magnitude lower than SIL1 (i.e., the probability of their delivering an incorrect response is less than $10^{-4}$ per hour of operation). In the calculations below we assumed that these failures occur with a negatively exponentially distributed arrival time with $\lambda = 10^{-4}$ per hour.

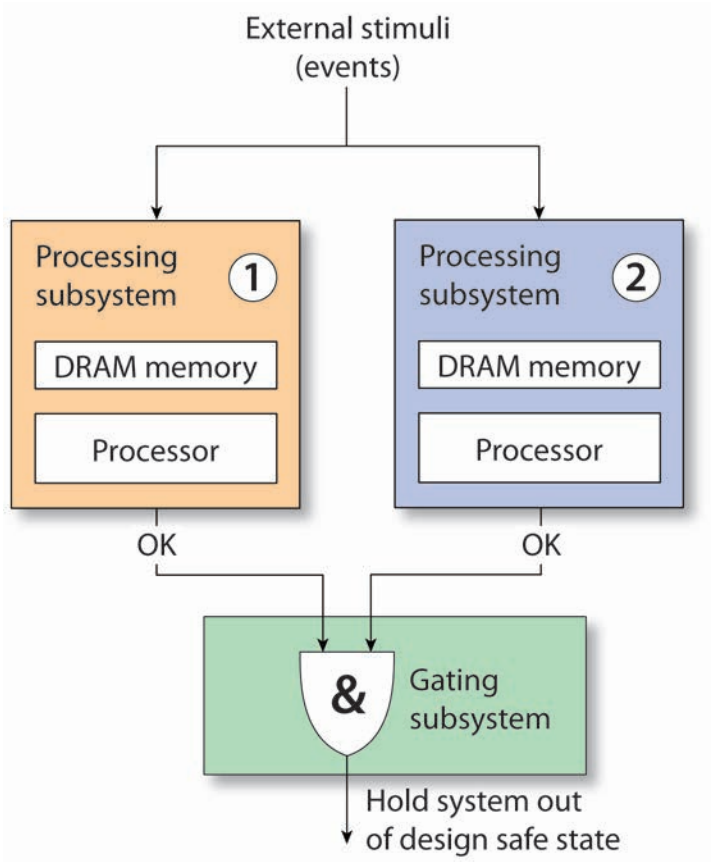


*Figure 2. A high-level view of the ATO controller's 2oo2 gating system. Both processing subsystems must agree that it is safe for the controller to run.*

### The probability of two simultaneous incorrect answers

It is essential for us to know the correlation between the failures of the two processor subsystems: What is the probability that both processors will calculate an incorrect answer simultaneously?

Given than most errors will occur in the application (rather than the operating system, which has millions of hours of dependable in-field use), and assuming that the application is common to both subsystems, based on our (hypothetical) fault injection testing we can assume in what follows that there is a 70% chance that an error in one processor subsystem will also appear in the other processor subsystem.

Since most errors that go undetected during system validation are likely to be Heisenbugs rather than Bohrbugs,[7] we consider our estimate to be conservative. Like all of the other values we use in this paper, its validity would need to be assessed during testing and field trials.

## Assumptions about the memory devices

We assume that the memory devices (DIMMs) in our 2oo2 system have single-bit error correction and multiple-bit error detection (SECDED) ECC algorithms built in based on a Hamming code with a minimum distance of 4. We also assume that the memory devices do *not* have Chipkill algorithms.

It is difficult to get firm figures for the failure of memory devices. Until Schroeder, Pinheiro and Weber's 2009 publication of a large study of Google servers, "DRAM Errors in the Wild: A Large-Scale Field Study"[8] the anticipated errors rates were in the order of 200 to 5000 FITs[9] per Mbit. With these estimates, a 2 Gigabyte DRAM could expect an error every 12 to 313 hours of operation. The Google study found something quite different. It discovered that DRAMs experienced either no errors or a large number of errors: their figures were 25,000 to 75,000 FITs per Mbit. The 75,000 figure corresponds to an error every 50 minutes of operation for a 2 Gigabyte DRAM.

In order to have a justifiable error rate for calculation, we used the raw information extracted from Table 2 (page 4) of the Google study (DRAMs for platforms A, B, C, D and F) converted to FITs per Mbit. From this information we can assume a value of 36,707 FITs per Mbit as a reasonable value for correctable errors (i.e., single bit errors corrected by the ECC hardware in the memory device).

The Google study indicates that between 0.08% and 0.3% (mean 0.22%) of DIMMs experience an uncorrectable error per year, and that there is a significant correlation between correctable and uncorrectable errors: if a correctable error occurs on a DIMM, then the probability of seeing an uncorrectable error on that DIMM within the same month is between 9 and 47 times higher than on a DIMM where no correctable error was observed.[10]

For the sake of our modeling, we assume that detected but uncorrectable errors occur at a rate of 1 every $1/0.22 = 4.5$ years per 2 Gigabyte DIMM. This value corresponds to approximately 3 FITs per Mbit, a value supported by the 2004 paper "Soft Errors in Electronic Memory—A White Paper" from Tezzaron Semiconductor, which says that uncorrectable errors occur one to two decimal orders of magnitude less often than correctable ones.[11]

One value that is impossible to deduce from the statistics given in the Google paper is the rate of *undetected* memory errors: by definition these errors were not detected, so no figures are available—or can ever be available—for them.[12] To be cautious in our estimations, in the following calculation we assume that these undetected bugs occur with a frequency one decimal order of magnitude lower than uncorrectable errors; that is, 0.3 FITs per Mbit. Thus, in summary, for our calculations we assume the following error rates:

- Detected and corrected errors: 36,707 FITS per Mbit
- Detected but uncorrectable errors: 3 FITS per Mbit
- Undetected errors: 0.3 FITS per Mbit

## Assumptions about handling memory errors

Three types of memory failure are possible. In our calculations we make the following assumptions about how these three error types are handled.

### Detected and correctable memory errors

Detected and correctable memory errors are counted, but otherwise ignored because they are corrected by the hardware and are invisible to the application. In light of the evidence presented in the Schroeder *et al.* study of Google servers, however, in a real product it would be wise to monitor these errors: correctable errors today appear to be strongly correlated with uncorrectable errors tomorrow.

### Detected but uncorrectable memory errors

Detected but uncorrectable memory errors are assumed to cause the associated processor to shut down. This shutdown will almost certainly be detected by the gating subsystem and cause the system to take appropriate action to recover or move to a design safe state (see "Assumptions about the gating subsystem" above). However, if the processor shutdown passes undetected by the gating subsystem, it will lead to a dangerous failure of the entire system.

### Undetected memory errors

Undetected memory errors are clearly the most dangerous type of error. In practice, many undetected errors will affect unused or uninitialized memory and will be, therefore, harmless. However, we must expect that some of these errors will affect active memory. The result may be relatively benign, say an incorrect character in a string for display to a user (benign if we assume that the affected character does not cause the user to take an incorrect action). Unfortunately, though, in some cases an error in active memory will cause the two processing subsystems to give different answers, and the gating subsystem must detect the discrepancy.

For the calculations in this document, we assume that *all* undetected memory errors cause the affected processor to produce wrong values. This is a very conservative estimate; many undetected memory errors may have no effect on a processor. We use this estimate, however, with the view that it is more prudent to assume the worst.

## Calculation with no software error detection

To estimate the dangerous failure rate, we ran a simulation of $10^9$ years (about $88 \times 10^{12}$ hours) 100 times, enough to obtain sufficient results for us to calculate a confidence interval.[13] The results of our simulation are shown in Table 1, which should be read as follows: "It can be said with 99% confidence that, given the assumptions listed above, the dangerous failure rate lies between 7.96157 and 8.01067 FITs."

| Confidence Level | Dangerous Failure Probability | |
|---|---|---|
| | Lower Bound | Upper Bound |
| 95.0% | 7.96884 FITs | 8.00340 FITs |
| 97.5% | 7.96547 FITs | 8.00677 FITs |
| 99.0% | 7.96157 FITs | 8.01067 FITs |

*Table 1. Results of calculation for estimated dangerous failure rate, with no software error detection.*

Thus, combining two SIL0 processing subsystems with a SIL4 (low-demand mode) gate and without using software memory error detection, the resulting system is a SIL4 system.

### Limits of our calculations

Note, that we have not performed the second essential calculation as described in "Assumptions about the processing subsystems" above. We have shown that, subject to the assumptions given, the system provides the safety level required, but we have not shown that it can meet its availability targets. A system that never moves from its design safe state (a train that is always stationary, a traffic light that is always red, etc.) is safe but useless or worse.[14] For a real system, we would have to make the availability calculation to show that our system is not only reliable, but also available and useful (see the bicycle paradigm "Reliability or availability?" above).

## Calculation with software error detection

Given the relatively slow speed at which application-level software error detection operates (about 23 hours to test two Gigabytes of memory),[15] it is likely that ECC hardware will find both correctable and detectable but uncorrectable memory errors well before the software finds them.

Software could, however, complement the ECC hardware and be used to find hard memory errors that have slipped by the ECC circuitry undetected. Software error detection could be useful, therefore, in the following circumstance:

**Hard and Soft Memory Errors**

Memory errors are often referred to as "hard" or "soft". A hard memory error is a memory error that is permanent: a section of memory (one bit or several bits) is no longer able to accept and keep its setting. A soft memory error is transient: a bit fails to keep its setting, but may behave correctly the next time it is used—perhaps every time it is used following the single failure.

While the difference between hard and soft errors may be immaterial to the software designer creating a software that must recover in the event of any memory errors, it is important to the designers of the overall system.

Schroeder *et al.* note a correlation between soft memory errors and the appearance in the future of hard memory errors. Further, as electronic components decrease in size, the speed at which they wear out and fail is increasing. Soft errors may be the canary in the coal mine announcing the imminent hard failures. Safety may depend not just on software that can (usually) recover from memory errors, but on maintenance programs that replace suspect boards and reduces the stress on this software.

1. An undetected memory error occurs.
2. This error affects the operation of one processing subsystem, causing it to give an incorrect answer, and
3. This error is not caught by the gating subsystem.

For our calculation, we have assumed that, if the conditions described above occur, then the software check has an 80% chance of finding the error and forcing a halt on the affected processor. This halt may, or may not, be caught by the gate subsystem.

When we re-ran the simulation with these assumptions, we found that there is effectively no change in the probabilities of dangerous failure given in Table 1 for the same system without software error correction. Our system remains a SIL4 (low-demand mode) system.

## Summary of findings concerning software error correction

The 2oo2 model provides an excellent controller design for providing system safety. Even with very low levels of dependability in the processing subsystems, the probability of dangerous failure is *very* low. Given the assumptions we have worked with, adding software detection of memory errors makes no appreciable difference in our system's dependability, and, in any case, detection is far too slow to be useful for detecting soft errors[16]. In short, software detection of soft memory errors does not appear to be a terribly useful solution for dealing with soft memory errors in our hypothetical ATO controller.

## Rethinking the problem

Soft memory errors are a real threat, and the incidence of these errors caused by cosmic radiation is very likely to increase with altitude. For our LRT controller, software error detection does not appear to be a good solution, for two reasons: a) it is too slow, and b) it does not appreciably reduce the probability of our controller failing. Our hypothetical controller's 2oo2 design appears to ensure our system's safety.

For our system, we should remember that just because the system is sufficiently safe, it is not necessarily sufficiently available—or, for that matter, even useful. We have noted that with our 2oo2 design, false positives can significantly compromise our controller's availability,[17] which may not only make our controller perform poorly, but also compromise the safe operation of the LRT by putting added stress on the correct and safe operation of other ATO components.

### Did we try to solve the right problem?

In his article about the 2005 grounding of the *Queen of Oak Bay* in Horseshoe Bay, British Columbia, Terry Hardy notes that on this ship redundancy was implemented in such a way that it was perfectly useless when both engines were shut down:

> While redundancy can theoretically improve reliability, redundancy can also increase system complexity and lead to unforeseen failures. In addition, redundancy can introduce unforeseen dependencies that can decrease safety.[18]

In other words, the redundant engines only improved system reliability if nothing compromised the engines' availability. As it turned out, both engines were unavailable and the entire system failed. Fortunately, no one was hurt in this accident.

Keeping in mind that our ATO system appears to be sufficiently safe but that we may not have given adequate consideration to availability, it may be worth our while to step back and try to rethink the problem. We could start by asking ourselves if the problem we are trying to solve is indeed soft memory errors.

As with so many questions in the real world, the answer is both “Yes” and “No”. It is “Yes” in that soft memory errors may create false positives in our 2oo2 subsystem, forcing our controller into a design safe state and compromising its availability. It is “No”, in that we only care about soft memory errors insofar as they might cause the controller to actually fail. What we really care about is that the controller is sufficiently dependable, which means both sufficiently reliable and sufficiently available.

Our 2oo2 design ensures our system is sufficiently safe. As we saw above, the probability that our system will deliver an incorrect response is within the requirements of IEC 61508 SIL4. As we also saw above, this carries a cost of possibly reduced availability. We could, therefore, rephrase our problem as follows:

1. We are confident that our system is sufficiently safe.
2. We are not confident that our system is sufficiently available.
3. Therefore, how can we improve our system’s availability without compromising its safety?

## Alternate strategies

The following are a few suggestions that may help improve controller availability. We are assuming that we will not alter the 2oo2 design, as it is fundamental to our reliability claims, and that we would perform the appropriate calculations to evaluate the effect of each change we implement.

### Second opinions

If there is time to ask for an information refresh when gating doesn’t agree, then asking for this refresh may be an excellent solution. Since soft memory errors are by their nature transient, if the disagreement in the two gating subsystem processors is caused by a soft error, a refresh will likely provide correct and matching answers, and avert a false positive. If the second opinion confirms the first disagreement, then we can be confident that something is amiss and have the controller take appropriate action, such as move to a design safe state.

This second opinion strategy may be triggered dynamically. A controller on a train stopped in a metro station may be able to a two second delay required for a reset and retry. A controller on the same train approaching a station at 50 km/h may not. The decision to request a second opinion could therefore be triggered by the train’s speed, as well as other factors, such as its location and the weather (which might affect stopping distance).

### ECC

ECC, including Chipkill (repair up to 4  bits), can be used to handle soft memory errors. In many cases a SECDED algorithm on the memory can provide the necessary level of resilience in the system. Chipkill memory would provide even more resilience, as it would handle many memory errors before they were noticed by the software. This memory is costlier, but its cost may be acceptable on low-volume products, such as LRT systems (compared to automobiles) and worth the decreased risk of memory errors.

**Board replacement**

Based on evidence such as that in the Google study cited earlier, there appears to be a correlation between soft memory errors now and hard memory errors showing up in the future. We may be able to improve dependability by logging errors and replacing those boards where soft errors are most frequent, by replacing boards more frequently, say every 20 months instead of waiting five years.

## What did we miss?

Our focus on soft memory errors was legitimate, considering the change in context where our system is to be deployed and what we know about neutron flux at altitude. However, now that we have examined this problem, we should also consider what we might have missed by focusing on solutions to soft memory errors caused by cosmic radiation.

For example, is our system susceptible to soft memory errors caused by electro-magnetic interference from, say, mobile phones or other radios or power sources? Are the physical location of the system and cabin design such that no source of radiation can be placed close to the system and inadvertently interfere with its correct functioning? Has the change in altitude and the correspondingly thinner atmosphere significantly changed our hardware's cooling requirements?

Since our ATO system has already been in use in another environment, we can probably assume that we have designed the system with the understanding that components and the system itself may indeed fail. That is, we have designed our system to:

- isolate safety-critical components from other components and each other, as required by the relevant safety standards
- detect errors while the system is running and correct them
- detect failures and contain them, or perform controlled shutdowns and restarts, or move to a design safe state as required and possible

Finally, we need to ask if the solutions we propose decrease or increase the risk of a failure, both of the specific system we are designing (in our case the in-cab controller for the ATO system), and of the larger system in which it will be implemented. Even if, for example, false positives causing a decline in availability do not significantly compromise the dependability of the controller, what are the consequences for the entire LRT system? If moving to a design safe state and even stopping the LRT train is acceptable, how does this solution affect the rest of the system? How does having an unplanned stop of one train increase the stress on other elements in the system?

To make our bicycle paradigm more accurately describe the questions we face with our ATO controller, we need to adjust it somewhat. We need to consider, not just one bicycle, but many bicycles packed together in a race. A solitary bicycle may be able to get away with suddenly slowing down while its controller resets, but a bicycle that does this in a race is a danger to the other bicycles. It may put them in a situation they cannot handle, and send them all crashing into the pavement.

## Notes

[1] Eastland Memorial Society web site. <www.eastlandmemorial.org/eastland7.shtml>

[2] Office of Air Accidents Investigation, *Aircraft Accident Report No. 79-139, Air New Zealand McDonnell-Douglas DC10-30 ZK-NZP,Ross Island, Antarctica, 28 November 1979*, Wellington, 12 June 1980. para. 1.17.7.
< www.nzalpa.org.nz/Portals/4/Documents/Reports/Chippindale/79-139_section1.pdf>

[3] P.T. Mahon, Report of the Royal Commission to inquire into The Crash on Mount Erebus, Antarctica of a DC10 Aircraft operated by Air New Zealand Limited, Wellington, pp. 157-59.
<archives.govt.nz/exhibitions/currentexhibitions/chch/downloads/AntarcticReport.pdf>

[4] See, for example, Ray Heald, *How Cosmic Rays Cause Computer Downtime* (IEEE Rel. Soc. SCV Meeting: 3/23/05) who states that "Cosmic events are the dominate cause of soft errors in ICs manufactured with very low alpha materials", p. 22, and "Neutrons are the primary problem", p. 23. See also Tom Simonite, "Should every computer chip have a cosmic ray detector?" *New Scientist Technology Blog*. 7 March 2008.
<www.newscientist.com/blog/technology/2008/03/do-we-need-cosmic-ray-alerts-for.html>

[5] Seutest describes itself as "a cooperatively managed website providing links and support for soft-error testing compatible with the JEDEC standard JESD89 – 'Measurement and Reporting of Alpha Particles and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices'". Solar modulation: 50% for both locations.

[6] We have deliberately used an extreme case. However, the effect of altitude is also significant in, say, Denver Colorado, which is often used (in the U.S. at least) to illustrate these types of scenarios. Of course, systems in aircraft are even more vulnerable; for example, at 10,000 metres (well below the service ceiling of many airliners) above Rome, the relative neutron flux is 147.36! "Understanding Soft and Firm Errors in Semiconductor Devices", an FAQ published by Actel Corporation notes that "In a commercial airplane, the effect [of neutrons] can be 100-800 times worse than at sea-level", 2002, p. 1.

[7] A Heisenbug is an irreproducible failure that is unlikely to occur on both subsystems simultaneously. A Bohrbug is a solid, reproducible bug that would arise on both systems.

[8] Bianca Schroeder, Eduardo Pinheiro and Wolf-Dietrich Weber, "DRAM Errors in the Wild: A Large-Scale Field Study", Seattle: SIGMETRICS/Performance '09, June 15-19, 2009.

[9] A FIT is one failure per $10^9$ hours.

[10] Schoeder *et al.*, p. 6

[11] Tezzaron Semiconductor, "Soft Errors in Electronic Memory—A White Paper", 2004.
<www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf>

[12] This is the problem of silent evidence. We can say that we have found no evidence of bugs in the system, but we cannot say that we have found evidence of no bugs.

[13] We can make this program available on request, but it is a *very* simple program. It could be argued that, given the simplicity of the assumptions, in particular the negative exponential distributions, a simulation is overkill. We assumed, though, that the program might need to be extended in the future.

[14] The traffic light that is always red is only always safe if we assume that drivers will always obey the light. In practice drivers will eventually lose patience (the timing dependent on individual personality and local driving culture) and risk crossing the intersection.

[15] Assuming 8 Kbytes are tested every 330 milliseconds.

[16] Whether detection after the system has operated incorrectly for several hours would actually be useful is a question that needs to be considered at the system level.

[17] Moving the controller unnecessarily to its design safe state is also a reliability problem: the decision mechanism gave the wrong answer.

[18] Terry Hardy, "Grounding of the Ship *Queen of Oak Bay",* Journal of Safety*, Sept-Oct., 2012, p. 7.*

## About QNX Software Systems

QNX Software Systems Limited, a subsidiary of BlackBerry, is a leading vendor of operating systems, development tools, and professional services for connected embedded systems. Global leaders such as Audi, Cisco, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle infotainment units, network routers, medical devices, industrial automation systems, security and defense systems, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems Limited is headquartered in Ottawa, Canada; its products are distributed in more than 100 countries worldwide. Visit www.qnx.com and facebook.com/QNXSoftwareSystems, and follow @QNX_News on Twitter. For more information on the company's automotive work, visit qnxauto.blogspot.com and follow @QNX_Auto.

**www.qnx.com**