# Compiling an application for use in highly radioactive environments

Asked 8 years, 8 months ago Modified 1 month ago Viewed 531k times

1636

We are compiling an embedded C++ application that is deployed in a shielded device in an environment bombarded with ionizing radiation. We are using GCC and cross-compiling for ARM. When deployed, our application generates some erroneous data and crashes more often than we would like. The hardware is designed for this environment, and our application has run on this platform for several years.

Are there changes we can make to our code, or compile-time improvements that can be made to identify/correct soft errors and memory-corruption caused by single event upsets? Have any other developers had success in reducing the harmful effects of soft errors on a long-running application?

c++ c gcc embedded fault-tolerance

Share

Improve this question

Follow

edited Nov 24, 2020 at 14:07

Stargateur
26.6k ● 9 ● 71 ● 95

asked Apr 24, 2016 at 19:09

rook
**67k** ● 38 ● 166 ● 245

230 Are the values in memory changing or are values in the processor changing? If the hardware is *designed* for the environment, the software should run as if running on a non-radioactive environment. – Thomas Matthews Apr 24, 2016 at 19:19

4 If possible, you should set up a logging system that stores events in non-volatile memory that is resistant to radiation. Store enough information so that you can trace the event and easily find the root cause. – Thomas Matthews Apr 24, 2016 at 19:21

23 This is a combination hardware/software solution, but I know Texas Instruments (and probably others) makes embedded chips for safety critical applications that consist of two duplicate cores, running in lockstep, half a clock cycle out of phase. There are special interrupts and reset actions that get taken when the hardware detects something different between the cores, so you can recover from errors. I believe TI brands them as "Hercules" safety processors. – mbrig Oct 10, 2017 at 16:39

12 Redundant rugged motors, some gears, shafts and ratchets! Replace annually or more often as dose rates require. No really, my first question with these kinds of issues has always been, do you really need that much software in there? Be as analog as you can possibly get away with. – jwdonahue Mar 2, 2018 at 4:00

7 You may have already done this, but you should of course verify that the observed crashes can be attributed to solely the radiation environment, and not some other error. For example do the crashes never happen in a benign test environment, and in such an environment, is that the only change; i.e. is it an independent factor. – Clifford Feb 2, 2019 at 2:32

## 24 Answers

Sorted by: Highest score (default)

912

Working for about 4-5 years with software/firmware development and environment testing of miniaturized satellites*, I would like to share my experience here.

*(*miniaturized satellites are a lot more prone to single event upsets than bigger satellites due to its relatively small, limited sizes for its electronic components*)

> To be very concise and direct: there is no mechanism to recover from *detectable, erroneous situation* by the software/firmware itself *without*, at least, one *copy* of *minimum working version* of the software/firmware *somewhere* for *recovery* purpose - and with the *hardware supporting the recovery* (functional).

Now, this situation is normally handled both in the hardware and software level. Here, as you request, I will share what we can do in the software level.

1. **...recovery purpose...**. Provide ability to update/recompile/reflash your software/firmware in real environment. This is an *almost must-have* feature for any software/firmware in highly ionized environment. Without this, you *could* have redundant software/hardware as many as you want but at one point, they are all going to blow up. So, prepare this feature!

2. **...minimum working version...** Have responsive, multiple copies, minimum version of the software/firmware in your code. This is like Safe mode in Windows. Instead of having only one, fully functional version of your software, have multiple copies of the minimum version of your software/firmware. The minimum copy will usually having much less size than the full copy and almost always have *only* the following two or three features:

    1. capable of listening to command from external system,

    2. capable of updating the current software/firmware,

    3. capable of monitoring the basic operation's housekeeping data.

3. **...copy... somewhere...** Have redundant software/firmware somewhere.

    1. You could, with *or* without redundant hardware, try to have redundant software/firmware in your ARM uC. This is normally done by having two or

more identical software/firmware *in separate addresses* which sending heartbeat to each other - but only one will be active at a time. If one or more software/firmware is known to be unresponsive, switch to the other software/firmware. The benefit of using this approach is we can have functional replacement immediately after an error occurs - without any contact with whatever external system/party who is responsible to detect and to repair the error (in satellite case, it is usually the Mission Control Centre (MCC)).

Strictly speaking, without redundant hardware, the disadvantage of doing this is you actually *cannot* eliminate *all* single point of failures. At the very least, you will still have *one* single point of failure, which is *the switch itself* (or often the beginning of the code). Nevertheless, for a device limited by size in a highly ionized environment (such as pico/femto satellites), the reduction of the single point of failures to one point *without* additional hardware will still be worth considering. Somemore, the piece of code for the switching would certainly be much less than the code for the whole program - significantly reducing the risk of getting Single Event in it.

2. But if you are not doing this, you should have at least one copy in your external system which can come in contact with the device and update

the software/firmware (in the satellite case, it is again the mission control centre).

3. You could also have the copy in your permanent memory storage in your device which can be triggered to restore the running system's software/firmware

4. **...detectable erroneous situation..** The error must be *detectable*, usually by the hardware *error correction/detection circuit* or by a small piece of code for error correction/detection. It is best to put such code small, multiple, and *independent* from the main software/firmware. Its main task is *only* for checking/correcting. If the hardware circuit/firmware is *reliable* (such as it is more radiation hardened than the rests - or having multiple circuits/logics), then you might consider making error-correction with it. But if it is not, it is better to make it as error-detection. The correction can be by external system/device. For the error correction, you could consider making use of a basic error correction algorithm like Hamming/Golay23, because they can be implemented more easily both in the circuit/software. But it ultimately depends on your team's capability. For error detection, normally CRC is used.

5. **...hardware supporting the recovery** Now, comes to the most difficult aspect on this issue. Ultimately, the recovery requires the hardware which is responsible for the recovery to be *at least* functional. If the hardware is permanently broken (normally happen after its **Total ionizing dose** reaches certain

level), then there is (sadly) no way for the software to help in recovery. Thus, hardware is rightly the utmost importance concern for a device exposed to high radiation level (such as satellite).

In addition to the suggestion for above anticipating firmware's error due to single event upset, I would also like to suggest you to have:

1. Error detection and/or error correction algorithm in the inter-subsystem communication protocol. This is another almost must have in order to avoid incomplete/wrong signals received from other system
2. Filter in your ADC reading. Do *not* use the ADC reading directly. Filter it by median filter, mean filter, or any other filters - *never* trust single reading value. Sample more, not less - reasonably.

Share Improve this answer Follow

edited Nov 25, 2016 at 2:18
Robert Harvey
181k ● 48 ● 346 ● 512

answered Apr 25, 2016 at 2:58
Ian
30.8k ● 19 ● 76 ● 111

---

446

NASA has [a paper on radiation-hardened](a paper on radiation-hardened) software. It describes three main tasks:

1. Regular monitoring of memory for errors then scrubbing out those errors,

2. robust error recovery mechanisms, and
3. the ability to reconfigure if something no longer works.

Note that the memory scan rate should be frequent enough that multi-bit errors rarely occur, as most ECC memory can recover from single-bit errors, not multi-bit errors.

Robust error recovery includes control flow transfer (typically restarting a process at a point before the error), resource release, and data restoration.

Their main recommendation for data restoration is to avoid the need for it, through having intermediate data be treated as temporary, so that restarting before the error also rolls back the data to a reliable state. This sounds similar to the concept of "transactions" in databases.

They discuss techniques particularly suitable for object-oriented languages such as C++. For example

1. Software-based ECCs for contiguous memory objects
2. Programming by Contract: verifying preconditions and postconditions, then checking the object to verify it is still in a valid state.

And, it just so happens, NASA has used C++ for major projects such as the Mars Rover.

> C++ class abstraction and encapsulation enabled rapid development and testing among multiple projects and developers.

They avoided certain C++ features that could create problems:

1. Exceptions
2. Templates
3. Iostream (no console)
4. Multiple inheritance
5. Operator overloading (other than `new` and `delete`)
6. Dynamic allocation (used a dedicated memory pool and placement `new` to avoid the possibility of system heap corruption).

Share Improve this answer Follow

edited Sep 8, 2018 at 17:25

answered Apr 24, 2016 at 19:32

rsjaffe
5,720 ● 7 ● 30 ● 39

33 This actually sounds like something that a pure language would be good at. Since values never change, if they are damaged you can just go back to the original definition (which is what it is supposed to be), and you won't accidentally do the same thing twice (because of lack of side effects). – Christopher King Apr 25, 2016 at 2:26

24 RAII is a bad idea, because you can't depend on it performing correctly or even at all. It could randomly damage your data etc. You really want as much immutability as you can get, and error correction mechanisms on top of that. It's much easier to just throw away broken things than it is to try and repair them somehow (how exactly do you know enough to go back to the correct old state?). You probably want to use a rather stupid language for this, though - optimizations might hurt more than they help. – Luaan Apr 25, 2016 at 8:22

78 @PyRulez: Pure languages are an abstraction, hardware isn't pure. Compilers are quite good at hiding the difference. If your program has a value it logically shouldn't use anymore after step X, the compiler may overwrite it with a value that's calculated in step X+1. But this means you can't go back. More formally, the possible states of a program in a pure language form an acyclic graph, which means two states are equivalent and can be merged when the states reachable from both are equivalent. This merger destroys the difference in paths leading to those states. – MSalters Apr 26, 2016 at 14:23

3 @Vorac - According to the presentation the concern with C++ templates is code bloat. – jww Jun 12, 2019 at 9:18

3 @DeerSpotter The exact problem is much more bigger than that. Ionization can damage bits of your running watcher program. Then you will need a watcher of a watcher, then - watcher of a watcher of a watcher and so on ... – Agnius Vasiliauskas Jun 12, 2019 at 11:59

133

Here are some thoughts and ideas:

**Use ROM more creatively.**

Store anything you can in ROM. Instead of calculating things, store look-up tables in ROM. (Make sure your compiler is outputting your look-up tables to the read-only section! Print out memory addresses at runtime to check!) Store your interrupt vector table in ROM. Of course, run some tests to see how reliable your ROM is compared to your RAM.

**Use your best RAM for the stack.**

SEUs in the stack are probably the most likely source of crashes, because it is where things like index variables, status variables, return addresses, and pointers of various sorts typically live.

**Implement timer-tick and watchdog timer routines.**

You can run a "sanity check" routine every timer tick, as well as a watchdog routine to handle the system locking up. Your main code could also periodically increment a counter to indicate progress, and the sanity-check routine could ensure this has occurred.

**Implement error-correcting-codes in software.**

You can add redundancy to your data to be able to detect and/or correct errors. This will add processing time, potentially leaving the processor exposed to radiation for a longer time, thus increasing the chance of errors, so you must consider the trade-off.

**Remember the caches.**

Check the sizes of your CPU caches. Data that you have accessed or modified recently will probably be within a cache. I believe you can disable at least some of the caches (at a big performance cost); you should try this to see how susceptible the caches are to SEUs. If the caches are hardier than RAM then you could regularly read and re-write critical data to make sure it stays in cache and bring RAM back into line.

**Use page-fault handlers cleverly.**

If you mark a memory page as not-present, the CPU will issue a page fault when you try to access it. You can create a page-fault handler that does some checking before servicing the read request. (PC operating systems use this to transparently load pages that have been swapped to disk.)

**Use assembly language for critical things (which could be everything).**

With assembly language, you *know* what is in registers and what is in RAM; you *know* what special RAM tables the CPU is using, and you can design things in a roundabout way to keep your risk down.

Use `objdump` to actually look at the generated assembly language, and work out how much code each of your routines takes up.

If you are using a big OS like Linux then you are asking for trouble; there is just so much complexity and so many things to go wrong.

**Remember it is a game of probabilities.**

A commenter said

> Every routine you write to catch errors will be subject to failing itself from the same cause.

While this is true, the chances of errors in the (say) 100 bytes of code and data required for a check routine to function correctly is much smaller than the chance of errors elsewhere. If your ROM is pretty reliable and almost all the code/data is actually in ROM then your odds are even better.

**Use redundant hardware.**

Use 2 or more identical hardware setups with identical code. If the results differ, a reset should be triggered. With 3 or more devices you can use a "voting" system to try to identify which one has been compromised.

Share Improve this answer Follow

answered Apr 24, 2016 at 23:11

Artelius

49k ● 13 ● 92 ● 106

14 Nowadays, there is ECC available through hardware, which saves the processing time. Step one would be to pick a microcontroller with built-in ECC. – Lundin Apr 25, 2016 at 8:01

30 Somewhere in the back of my mind is a reference to avionics (perhaps space shuttle?) flight hardware where the redundant architecture was explicitly designed not to be identical (and by different teams). Doing so mitigates the possibility of a systemic error in the hardware/software design, reducing the possibility of all of the voting systems crashing at the same time when confronted with the same inputs. – Peter M Apr 26, 2016 at 12:21

11 @PeterM: AFAIK that's also claimed for the flight software for the Boeing 777: Three versions by three teams in three programming languages. – Martin Schröder Apr 26, 2016 at 21:12

7 @DanEsparza RAM typically has either a capacitor (DRAM) or a few transistors in feedback (SRAM) storing data. A radiation event can spuriously charge/discharge the capacitor, or change the signal in the feedback loop. ROM does not typically need the ability to be written (at least without special circumstances and/or higher voltages) and hence may be inherently more stable at the physical level. – nanofarad Apr 27, 2016 at 14:48

7 @DanEsparza: There are multiple types of ROM memories. If the "ROM" is emulated by i.e. eeprom or flash readonly-at-5v but-programmable-at-10v, then indeed that "ROM" is still prone to ionization. Maybe just less than others. However, there are good ol' hardcore things like Mask ROM or fuse-based PROM which I think would need a really serious amount of radiation to start failing. I don't know however if there are still manufactured. – quetzalcoatl Apr 28, 2016 at 8:19

You may also be interested in the rich literature on the subject of algorithmic fault tolerance. This includes the old assignment: Write a sort that correctly sorts its input when

122

+500

a constant number of comparisons will fail (or, the slightly more evil version, when the asymptotic number of failed comparisons scales as `log(n)` for `n` comparisons).

A place to start reading is Huang and Abraham's 1984 paper "Algorithm-Based Fault Tolerance for Matrix Operations". Their idea is vaguely similar to homomorphic encrypted computation (but it is not really the same, since they are attempting error detection/correction at the operation level).

A more recent descendant of that paper is Bosilca, Delmas, Dongarra, and Langou's "Algorithm-based fault tolerance applied to high performance computing".

Share Improve this answer Follow

edited Apr 27, 2016 at 18:04
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Apr 24, 2016 at 21:13
Eric Towers
4,215 ● 1 ● 17 ● 17

52

Writing code for radioactive environments is not really any different than writing code for any mission-critical application.

In addition to what has already been mentioned, here are some miscellaneous tips:

- Use everyday "bread & butter" safety measures that should be present on any semi-professional embedded system: internal watchdog, internal low-voltage detect, internal clock monitor. These things shouldn't even need to be mentioned in the year 2016 and they are standard on pretty much every modern microcontroller.

- If you have a safety and/or automotive-oriented MCU, it will have certain watchdog features, such as a given time window, inside which you need to refresh the watchdog. This is preferred if you have a mission-critical real-time system.
- In general, use a MCU suitable for these kind of systems, and not some generic mainstream fluff you received in a packet of corn flakes. Almost every MCU manufacturer nowadays have specialized MCUs designed for safety applications (TI, Freescale, Renesas, ST, Infineon etc etc). These have lots of built-in safety features, including lock-step cores: meaning that there are 2 CPU cores executing the same code, and they must agree with each other.
- IMPORTANT: You must ensure the integrity of internal MCU registers. All control & status registers of hardware peripherals that are writeable may be located in RAM memory, and are therefore vulnerable.

  To protect yourself against register corruptions, preferably pick a microcontroller with built-in "write-once" features of registers. In addition, you need to store default values of all hardware registers in NVM and copy-down those values to your registers at regular intervals. You can ensure the integrity of important variables in the same manner.

Note: always use defensive programming. Meaning that you have to setup *all* registers in the MCU and not just the ones used by the application. You don't want some random hardware peripheral to suddenly wake up.

- There are all kinds of methods to check for errors in RAM or NVM: checksums, "walking patterns", software ECC etc etc. The best solution nowadays is to not use any of these, but to use a MCU with built-in ECC and similar checks. Because doing this in software is complex, and the error check in itself could therefore introduce bugs and unexpected problems.
- Use redundancy. You could store both volatile and non-volatile memory in two identical "mirror" segments, that must always be equivalent. Each segment could have a CRC checksum attached.
- Avoid using external memories outside the MCU.
- Implement a default interrupt service routine / default exception handler for all possible interrupts/exceptions. Even the ones you are not using. The default routine should do nothing except shutting off its own interrupt source.
- Understand and embrace the concept of defensive programming. This means that your program needs

to handle all possible cases, even those that cannot occur in theory. Examples.

High quality mission-critical firmware detects as many errors as possible, and then handles or ignores them in a safe manner.

- Never write programs that rely on poorly-specified behavior. It is likely that such behavior might change drastically with unexpected hardware changes caused by radiation or EMI. The best way to ensure that your program is free from such crap is to use a coding standard like MISRA, together with a static analyser tool. This will also help with defensive programming and with weeding out bugs (why would you not want to detect bugs in any kind of application?).
- IMPORTANT: Don't implement any reliance of the default values of static storage duration variables. That is, don't trust the default contents of the `.data` or `.bss`. There could be any amount of time between the point of initialization to the point where the variable is actually used, there could have been plenty of time for the RAM to get corrupted. Instead, write the program so that all such variables are set from NVM in run-time, just before the time when such a variable is used for the first time.

  In practice this means that if a variable is declared at file scope or as `static`, you should never use `=` to initialize it (or you could, but it is pointless, because you cannot rely on the value anyhow). Always set it in run-time, just before use. If it is possible to repeatedly update such variables from NVM, then do so.

Similarly in C++, don't rely on constructors for static storage duration variables. Have the constructor(s) call a public "set-up" routine, which you can also call later on in run-time, straight from the caller application.

If possible, remove the "copy-down" start-up code that initializes `.data` and `.bss` (and calls C++ constructors) entirely, so that you get linker errors if you write code relying on such. Many compilers have the option to skip this, usually called "minimal/fast start-up" or similar.

This means that any external libraries have to be checked so that they don't contain any such reliance.

- Implement and define a safe state for the program, to where you will revert in case of critical errors.
- Implementing an error report/error log system is always helpful.

Share Improve this answer Follow

edited Jul 7, 2020 at 6:43

answered Apr 27, 2016 at 14:11

Lundin
**212k** ● 45 ● 270 ● 426

One way of dealing with booleans being corrupted (as in your example link) could be to make `TRUE` equal to `0xffffffff` then use [`POPCNT`]() with a threshold.
– wizzwizz4 Dec 23, 2017 at 18:46

@wizzwizz4 Given that the value 0xff is the default value of non-programmed flash cell, that sounds like a bad idea.
– Lundin Jan 16, 2018 at 7:41

1 @wizzwizz4 Or just the value 0x1, as required by the C standard. – Lundin Jan 16, 2018 at 7:47

1 But then a lucky cosmic ray could flip it to false! – wizzwizz4 Jan 16, 2018 at 7:49

1 @wizzwizz4 Why you use some or all of the above mentioned methods (ECC, CRC etc). Otherwise the cosmic ray may as well flip a single bit in your `.text` section, changing an op code or similar. – Lundin Jan 16, 2018 at 7:53

37

It may be possible to use C to write programs that behave robustly in such environments, but only if most forms of compiler optimization are disabled. Optimizing compilers are designed to replace many seemingly-redundant coding patterns with "more efficient" ones, and may have no clue that the reason the programmer is testing `x==42` when the compiler knows there's no way `x` could possibly hold anything else is because the programmer wants to prevent the execution of certain code with `x` holding some other value--even in cases where the only way it could hold that value would be if the system received some kind of electrical glitch.

Declaring variables as `volatile` is often helpful, but may not be a panacea. Of particular importance, note that safe coding often requires that dangerous operations have hardware interlocks that require multiple steps to activate, and that code be written using the pattern:

```
... code that checks system state
if (system_state_favors_activation)
{
  prepare_for_activation();
  ... code that checks system state again
  if (system_state_is_valid)
  {
    if (system_state_favors_activation)
      trigger_activation();
  }
  else
    perform_safety_shutdown_and_restart();
}
cancel_preparations();
```

If a compiler translates the code in relatively literal fashion, and if all the checks for system state are repeated after the `prepare_for_activation()`, the system may be robust against almost any plausible single glitch event, even those which would arbitrarily corrupt the program counter and stack. If a glitch occurs just after a call to `prepare_for_activation()`, that would imply that activation would have been appropriate (since there's no other reason `prepare_for_activation()` would have been called before the glitch). If the glitch causes code to reach `prepare_for_activation()` inappropriately, but there are no subsequent glitch events, there would be no way for code to subsequently reach `trigger_activation()` without having passed through the validation check or calling cancel_preparations first [if the stack glitches, execution might proceed to a spot just before `trigger_activation()` after the context that called `prepare_for_activation()` returns, but the call to `cancel_preparations()` would have occurred between the calls to `prepare_for_activation()` and `trigger_activation()`, thus rendering the latter call harmless.

Such code may be safe in traditional C, but not with modern C compilers. Such compilers can be very dangerous in that sort of environment because aggressive they strive to only include code which will be relevant in situations that could come about via some well-defined mechanism and whose resulting consequences would also be well defined. Code whose purpose would be to detect and clean up after failures may, in some cases, end up making things worse. If the compiler determines that the attempted recovery would in some cases invoke undefined behavior, it may infer that the conditions that would necessitate such recovery in such cases cannot possibly occur, thus eliminating the code that would have checked for them.

Share Improve this answer Follow

edited Aug 8, 2016 at 18:36

answered Apr 25, 2016 at 16:14

supercat
**80.8k** ● 9 ● 174 ● 220

7 Realistically speaking, how many modern compilers are there that don't offer `-O0` or an equivalent switch? GCC will do a lot of strange things *if you give it permission*, but if you ask it not to do them, it's generally able to be fairly literal too. – Alex Celeste Apr 25, 2016 at 23:35

28 Sorry, but this idea is fundamentally dangerous. Disabling optimizations produces a slower program. Or, in other words, you need a faster CPU. As it happens, faster CPU's are faster because the charges on their transistor gates are smaller. This makes them far more susceptible to radiation. The better strategy is to use a slow, big chip where a single photon is far less likely to knock over a bit, and gain back the speed with `-O2`. – MSalters Apr 26, 2016 at 14:28

30 A secondary reason why `-O0` is a bad idea is because it emits far more useless instructions. Example: a non-inlined call contains instructions to save registers, make the call, restore registers. All of these can fail. An instruction that's not there cannot fail. – MSalters Apr 26, 2016 at 14:30

17 Yet another reason why `-O0` is a bad idea: it tends to store variables in memory instead of in a register. Now it's not certain that memory is more susceptible to SEU's, but data in flight is more susceptible than data at rest. Useless data movement should be avoided, and `-O2` helps there. – MSalters Apr 26, 2016 at 14:31

10 @MSalters: What's important is not that data be immune to disruption, but rather that the system be able to handle

disruptions in a manner meeting requirements. On many compilers disabling all optimizations yields code that performs an excessive number of register-to-register moves, which is bad, but storing variables in memory is safer from a recovery standpoint than keeping them in registers. If one has two variables in memory which are supposed to obey some condition (e.g. `v1=v2+0xCAFEBABE` and all updates to the two variables are done... – supercat Apr 26, 2016 at 14:39

---

31

This is an extremely broad subject. Basically, you can't really recover from memory corruption, but you can at least try to **fail promptly**. Here are a few techniques you could use:

- **checksum constant data**. If you have any configuration data which stays constant for a long time (including hardware registers you have configured), compute its checksum on initialization and verify it periodically. When you see a mismatch, it's time to re-initialize or reset.
- **store variables with redundancy**. If you have an important variable `x`, write its value in `x1`, `x2` and `x3` and read it as `(x1 == x2) ? x2 : x3`.
- implement **program flow monitoring**. XOR a global flag with a unique value in important functions/branches called from the main loop. Running the program in a radiation-free environment with near-100% test coverage should give you the list of acceptable values of the flag at the end of the cycle. Reset if you see deviations.

- **monitor the stack pointer**. In the beginning of the main loop, compare the stack pointer with its expected value. Reset on deviation.

Share Improve this answer Follow

answered Apr 25, 2016 at 17:05

Dmitry Grigoryev
3,194 ● 1 ● 29 ● 59

---

28

What could help you is a watchdog. Watchdogs were used extensively in industrial computing in the 1980s. Hardware failures were much more common then - another answer also refers to that period.

A watchdog is a combined hardware/software feature. The hardware is a simple counter that counts down from a number (say 1023) to zero. TTL or other logic could be used.

The software has been designed as such that one routine monitors the correct operation of all essential systems. If this routine completes correctly = finds the computer running fine, it sets the counter back to 1023.

The overall design is so that under normal circumstances, the software prevents that the hardware counter will reach zero. In case the counter reaches zero, the hardware of the counter performs its one-and-only task and resets the entire system. From a counter perspective, zero equals 1024 and the counter continues counting down again.

This watchdog ensures that the attached computer is restarted in a many, many cases of failure. I must admit that I'm not familiar with hardware that is able to perform such a function on today's computers. Interfaces to external hardware are now a lot more complex than they used to be.

An inherent disadvantage of the watchdog is that the system is not available from the time it fails until the watchdog counter reaches zero + reboot time. While that time is generally much shorter than any external or human intervention, the supported equipment will need to be able to proceed without computer control for that timeframe.

Share Improve this answer Follow

edited Apr 28, 2016 at 9:37
Lundin
212k ● 45 ● 270 ● 426

answered Apr 26, 2016 at 22:41
OldFrank
878 ● 6 ● 7

10 Binary counter watchdogs with TTL standard ICs is indeed a 1980s solution. Don't do that. Today, there doesn't exist a single MCU on the market without built-in watchdog circuitry. All you need to check is if the built-in watchdog has an individual clock source (good, most likely the case) or if it inherits its clock from the system clock (bad). – Lundin Apr 27, 2016 at 11:12

1 Or implement the watchdog in an FPGA: [ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20130013486.pdf](ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20130013486.pdf) – nos Apr 27, 2016 at 13:07

2 Still used extensively in embedded processors, incidentally. – Graham Apr 27, 2016 at 16:02

5 @Peter Mortensen Kindly stop your edit spree on every answer to this question. This is not Wikipedia, and those links are not helpful (and I'm sure everyone knows how to find Wikipedia anyhow...). Many of your edits are incorrect because you don't know the topic. I'm doing roll-backs on your incorrect edits as I come across them. You are not turning this thread better, but worse. Stop editing. – Lundin Apr 28, 2016 at 9:44

Jack Ganssle has a good article on watchdogs: [ganssle.com/watchdogs.htm](ganssle.com/watchdogs.htm) – Igor Skochinsky Apr 29, 2016 at 11:11

---

25

*This answer assumes you are concerned with having a system that works correctly, over and above having a system that is minimum cost or fast; most people playing with radioactive things value correctness / safety over speed / cost*

Several people have suggested hardware changes you can make (fine - there's lots of good stuff here in answers already and I don't intend repeating all of it), and others have suggested redundancy (great in principle), but I don't think anyone has suggested how that redundancy might work in practice. How do you fail over? How do you know when something has 'gone wrong'? Many technologies work on the basis everything will work, and failure is thus a tricky thing to deal with. However, some distributed computing technologies designed for scale *expect* failure (after all with enough scale, failure of one node of many is inevitable with any MTBF for a single node); you can harness this for your environment.

Here are some ideas:

- Ensure that your entire hardware is replicated `n` times (where `n` is greater than 2, and preferably odd), and that each hardware element can communicate with each other hardware element. Ethernet is one obvious way to do that, but there are many other far simpler routes that would give better protection (e.g. CAN). Minimise common components (even power supplies). This may mean sampling ADC inputs in multiple places for instance.
- Ensure your application state is in a single place, e.g. in a finite state machine. This can be entirely RAM based, though does not preclude stable storage. It will thus be stored in several place.
- Adopt a quorum protocol for changes of state. See RAFT for example. As you are working in C++, there are well known libraries for this. Changes to the FSM would only get made when a majority of nodes agree. Use a known good library for the protocol stack and the quorum protocol rather than rolling one yourself, or all your good work on redundancy will be wasted when the quorum protocol hangs up.
- Ensure you checksum (e.g. CRC/SHA) your FSM, and store the CRC/SHA in the FSM itself (as well as transmitting in the message, and checksumming the messages themselves). Get the nodes to check their

FSM regularly against these checksum, checksum incoming messages, and check their checksum matches the checksum of the quorum.

- Build as many other internal checks into your system as possible, making nodes that detect their own failure reboot (this is better than carrying on half working provided you have enough nodes). Attempt to let them cleanly remove themselves from the quorum during rebooting in case they don't come up again. On reboot have them checksum the software image (and anything else they load) and do a full RAM test before reintroducing themselves to the quorum.
- Use hardware to support you, but do so carefully. You can get ECC RAM, for instance, and regularly read/write through it to correct ECC errors (and panic if the error is uncorrectable). However (from memory) static RAM is far more tolerant of ionizing radiation than DRAM is in the first place, so it *may* be better to use static DRAM instead. See the first point under 'things I would not do' as well.

Let's say you have an 1% chance of failure of any given node within one day, and let's pretend you can make failures entirely independent. With 5 nodes, you'll need three to fail within one day, which is a .00001% chance. With more, well, you get the idea.

Things I would *not* do:

- *Underestimate the value of not having the problem to start off with.* Unless weight is a concern, a large block of metal around your device is going to be a far cheaper and more reliable solution than a team of programmers can come up with. Ditto optical coupling of inputs of EMI is an issue, etc. Whatever, attempt when sourcing your components to source those rated best against ionizing radiation.
- *Roll your own algorithms*. People have done this stuff before. Use their work. Fault tolerance and distributed algorithms are hard. Use other people's work where possible.
- *Use complicated compiler settings in the naive hope you detect more failures.* If you are lucky, you may detect more failures. More likely, you will use a code-path within the compiler which has been less tested, particularly if you rolled it yourself.
- *Use techniques which are untested in your environment.* Most people writing high availability software have to simulate failure modes to check their HA works correctly, and miss many failure modes as a result. You are in the 'fortunate' position of having frequent failures on demand. So test each technique, and ensure its application actual improves MTBF by an amount that exceeds the complexity to introduce it (with complexity comes bugs). Especially apply this to my advice re quorum algorithms etc.

Share Improve this answer edited Apr 28, 2016 at 17:22

Follow

answered Apr 27, 2016 at 15:41

abligh
**25.1k** ● 4 ● 52 ● 87

2 Ethernet is probably not a great idea to use in mission-critical applications. Neither is I2C, outside the PCB itself. Something rugged like CAN would be far more suitable. – Lundin Apr 28, 2016 at 9:34

1 @Lundin Fair point, though anything optically connected (incl. ethernet) should be OK. – abligh Apr 28, 2016 at 17:21

1 The physical media is not so much the reason why Ethernet is unsuitable, but the lack of deterministic real-time behavior. Though I suppose there is nowadays ways to provide somewhat reliable Ethernet too, I just group it together with commercial/toy electronics out of old habit. – Lundin Apr 29, 2016 at 14:07

1 @Lundin that is a fair point, but as I'm suggesting using it to run RAFT, there will be (theoretically) non-deterministic realtime behaviour in the algorithm anyway (e.g. simultaneous leader elections resulting in a rerun election being similar to CSMA/CD). If strict realtime behaviour is needed, arguably my answer has more problems than ethernet (and note at the head of my reply I said 'correct' was likely to be at the expense of 'fast' often). I incorporated your point re CAN though. – abligh Apr 29, 2016 at 14:25

1 @Lundin: No system which involves asynchronous aspects can be fully non-deterministic. I think the worst-case behavior of Ethernet can be bounded in the absence of hardware disruptions if software protocols are set up in suitable fashion and devices have unique IDs and there is a known limit to the number of devices (the more devices, the larger the worst-case number of retries). – supercat Dec 15, 2017 at 18:06

23

Since you specifically ask for software solutions, and you are using C++, why not use operator overloading to make your own, safe datatypes? For example:

Instead of using `uint32_t` (and `double`, `int64_t` etc), make your own `SAFE_uint32_t` which contains a multiple (minimum of 3) of uint32_t. Overload all of the operations you want (* + - / << >> = == != etc) to perform, and make the overloaded operations perform independently on each internal value, ie don't do it once and copy the result. Both before and after, check that all of the internal values match. If values don't match, you can update the wrong one to the value with the most common one. If there is no most-common value, you can safely notify that there is an error.

This way it doesn't matter if corruption occurs in the ALU, registers, RAM, or on a bus, you will still have multiple attempts and a very good chance of catching errors. Note however though that this only works for the variables you can replace - your stack pointer for example will still be susceptible.

A side story: I ran into a similar issue, also on an old ARM chip. It turned out to be a toolchain which used an old version of GCC that, together with the specific chip we used, triggered a bug in certain edge cases that would (sometimes) corrupt values being passed into functions. Make sure your device doesn't have any problems before blaming it on radio-activity, and yes, sometimes it is a compiler bug =)

Share Improve this answer Follow

edited May 24, 2016 at 23:25
Mihriban Minaz
**3,039** ● 3 ● 33 ● 52

answered Apr 27, 2016 at 15:32
jkflying
**1,089** ● 7 ● 15

1 A few of these suggestions have something along a similar 'multi-bit sanity check' mindset for detecting corruption, I really like this one with the suggestion of safety-critical custom datatypes the most though – WearyWanderer Apr 29, 2016 at 13:22

3 There are systems in the world where each redundant node was designed and developed by different teams, with an arbiter to make sure they didn't accidentally settle on the same solutions. That way you don't have them all going down for the same bug and similar transients don't manifest similar failure modes. – jwdonahue Mar 2, 2018 at 3:45

This *does not work*. I have actual experience on faults. The problem is critical things such as addresses on the stack can't be replicated without compiler or hardware support. – Joshua Oct 3, 2022 at 2:35

@Joshua: It's possible to design hardware and software to work together in such a fashion that certain critical actions could not be erroneously triggered by any single disruptive event--*even one that could simultaneously load all registers and all bytes of memory with the most vexing values possible*. Doing this is much easier in assembly language than C code, but a program that's mostly in C can do a lot if a compiler behaves like a "high-level assembler" without trying to optimize out "redundant" safety checks. – supercat Aug 10 at 19:11

@supercat: You sure about that? I have encountered this one in the wild: on failure the CPU jumps to a random address within the range permissible for executable code, including to an address proven to be dead by static analysis. – Joshua Aug 10 at 19:16

---

18

Disclaimer: I'm not a radioactivity professional nor worked for this kind of application. But I worked on soft errors and redundancy for long term archival of critical data, which is somewhat linked (same problem, different goals).

The main problem with radioactivity in my opinion is that radioactivity can switch bits, thus **radioactivity can/will tamper any digital memory**. These errors are usually called soft errors, bit rot, etc.

The question is then: **how to compute reliably when your memory is unreliable?**

To significantly reduce the rate of soft errors (at the expense of computational overhead since it will mostly be software-based solutions), you can either:

- rely on the good old **redundancy scheme**, and more specifically the more efficient **error correcting codes** (same purpose, but cleverer algorithms so that you can recover more bits with less redundancy). This is sometimes (wrongly) also called checksumming. With this kind of solution, you will have to store the full state of your program at any moment in a master variable/class (or a struct?), compute an ECC, and check that the ECC is correct before doing anything, and if not, repair the fields. This solution however does not guarantee that your software can work (simply that it will work correctly when it can, or stops working if not, because ECC can tell you if something is wrong, and in this case you can stop your software so that you don't get fake results).

- or you can use **resilient algorithmic data structures**, which guarantee, up to a some bound, that your program will still give correct results even in the presence of soft errors. These algorithms can be seen as a mix of common algorithmic structures with ECC schemes natively mixed in, but this is much more resilient than that, because the resiliency scheme is tightly bounded to the structure, so that you don't need to encode additional procedures to check the ECC, and usually they are a lot faster. These structures provide a way to ensure that your program will work under any condition, up to the theoretical bound of soft errors. You can also mix these resilient structures with the redundancy/ECC scheme for additional security (or encode your most important data structures as resilient, and the rest, the expendable data that you can recompute from the main data structures, as normal data structures with a bit of ECC or a parity check which is very fast to compute).

If you are interested in resilient data structures (which is a recent, but exciting, new field in algorithmics and redundancy engineering), I advise you to read the following documents:

- Resilient algorithms data structures intro by Giuseppe F.Italiano, Universita di Roma "Tor Vergata"
- Christiano, P., Demaine, E. D., & Kishore, S. (2011). Lossless fault-tolerant data structures with additive overhead. In Algorithms and Data Structures (pp. 243-254). Springer Berlin Heidelberg.
- Ferraro-Petrillo, U., Grandoni, F., & Italiano, G. F. (2013). Data structures resilient to memory faults: an experimental study of dictionaries. Journal of Experimental Algorithmics (JEA), 18, 1-6.
- Italiano, G. F. (2010). Resilient algorithms and data structures. In Algorithms and Complexity (pp. 13-24). Springer Berlin Heidelberg.

If you are interested in knowing more about the field of resilient data structures, you can checkout the works of Giuseppe F. Italiano (and work your way through the refs) and the **Faulty-RAM model** (introduced in Finocchi et al. 2005; Finocchi and Italiano 2008).

/EDIT: I illustrated the prevention/recovery from soft-errors mainly for RAM memory and data storage, but I didn't talk about **computation (CPU) errors**. Other answers already pointed at using atomic transactions like in databases, so I will propose another, simpler scheme: **redundancy and majority vote**.

The idea is that you simply **do x times the same computation** for each computation you need to do, and store the result in x different variables (with x >= 3). You can then **compare your x variables**:

- if they all agree, then there's no computation error at all.
- if they disagree, then you can use a majority vote to get the correct value, and since this means the computation was partially corrupted, you can also trigger a system/program state scan to check that the rest is ok.
- if the majority vote cannot determine a winner (all x values are different), then it's a perfect signal for you to trigger the failsafe procedure (reboot, raise an alert to user, etc.).

This redundancy scheme is **very fast** compared to ECC (practically O(1)) and it provides you with a **clear signal** when you need to **failsafe**. The majority vote is also (almost) **guaranteed to never produce corrupted output** and also to **recover from minor computation errors**, because the probability that x computations give the same output is infinitesimal (because there is a huge amount of possible outputs, it's almost impossible to randomly get 3 times the same, even less chances if x > 3).

So with majority vote you are safe from corrupted output, and with redundancy x == 3, you can recover 1 error (with x == 4 it will be 2 errors recoverable, etc. -- the exact equation is `nb_error_recoverable == (x-2)` where x is the number of calculation repetitions because you need at least 2 agreeing calculations to recover using the majority vote).

The drawback is that you need to compute x times instead of once, so you have an additional computation cost, but's linear complexity so asymptotically you don't lose much for the benefits you gain. A fast way to do a majority vote is to compute the mode on an array, but you can also use a median filter.

Also, if you want to make extra sure the calculations are conducted correctly, if you can make your own hardware you can construct your device with x CPUs, and wire the system so that calculations are automatically duplicated across the x CPUs with a majority vote done mechanically at the end (using AND/OR gates for example). This is often implemented in airplanes and mission-critical devices (see triple modular redundancy). This way, you would not have any computational overhead (since the additional calculations will be done in parallel), and you have another layer of protection from soft errors (since the calculation duplication and majority vote will be managed directly by the hardware and not by software -- which can more easily get corrupted since a program is simply bits stored in memory...).

edited Mar 18, 2017 at 21:10

answered May 1, 2016 at 18:56

gaborous
16.5k ● 11 ● 89 ● 104

9

One point no-one seems to have mentioned. You say you're developing in GCC and cross-compiling onto ARM. How do you know that you don't have code which makes assumptions about free RAM, integer size, pointer size, how long it takes to do a certain operation, how long the system will run for continuously, or various stuff like that? This is a very common problem.

The answer is usually automated unit testing. Write test harnesses which exercise the code on the development system, then run the same test harnesses on the target system. Look for differences!

Also check for errata on your embedded device. You may find there's something about "don't do this because it'll crash, so enable that compiler option and the compiler will work around it".

In short, your most likely source of crashes is bugs in your code. Until you've made pretty damn sure this isn't the case, don't worry (yet) about more esoteric failure modes.

answered Apr 27, 2016 at 16:09

Follow

Graham
1,705 ● 10 ● 20

9

You want 3+ slave machines with a master outside the radiation environment. All I/O passes through the master which contains a vote and/or retry mechanism. The slaves must have a hardware watchdog each and the call to bump them should be surrounded by CRCs or the like to reduce the probability of involuntary bumping. Bumping should be controlled by the master, so lost connection with master equals reboot within a few seconds.

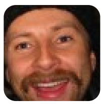
One advantage of this solution is that you can use the same API to the master as to the slaves, so redundancy becomes a transparent feature.

**Edit:** From the comments I feel the need to clarify the "CRC idea." The possibilty of the slave bumping it's own watchdog is close to zero if you surround the bump with CRC or digest checks on random data from the master. That random data is only sent from master when the slave under scrutiny is aligned with the others. The random data and CRC/digest are immediately cleared after each bump. The master-slave bump frequency should be more than double the watchdog timeout. The data sent from the master is uniquely generated every time.

Share Improve this answer Follow

edited May 2, 2016 at 13:35

answered Apr 27, 2016 at 14:16



Jonas Byström
**26.1k** ● 23 ● 104 ● 152

---

8 I'm trying to fathom a scenario where you can have a master outside the radiation environment, able to communicate reliably with slaves inside the radiation environment, where you couldn't just put the slaves outside of the radiation environment. – fostandy Apr 29, 2016 at 4:16

1 @fostandy: The slaves are either measuring or controlling using equipment that needs a controller. Say a geiger counter. The master does not need reliable communication due to slave redundancy. – Jonas Byström Apr 29, 2016 at 7:18

5 Introducing a master will not automatically mean increased security. If slave x has gone crazy due to memory corruption, so that it is repeatedly telling itself "master is here, master is happy", then no amount of CRCs or barked orders by the master will save it. You would have to give the master the possibility to cut the power of that slave. And if you have a common-cause error, adding more slaves will not increase safety. Also keep in mind that the amount of software bugs and the amount of things that can break increase with complexity. – Lundin Apr 29, 2016 at 13:57

5 That being said, it would of course be nice to "outsource" as much of the program to somewhere less exposed, while keeping the electronics inside the radioactive environment as simple as possible, if you have that option. – Lundin Apr 29, 2016 at 13:58

---

Use a cyclic scheduler. This gives you the ability to add regular maintenance times to check the correctness of

8

critical data. The problem most often encountered is corruption of the stack. If your software is cyclical, you can reinitialize the stack between cycles. Do not reuse the stacks for interrupt calls; set up a separate stack of each important interrupt call.

Similar to the Watchdog concept is deadline timers. Start a hardware timer before calling a function. If the function does not return before the deadline timer interrupts then reload the stack and try again. If it still fails after 3/5 tries you need reload from ROM.

Split your software into parts and isolate these parts to use separate memory areas and execution times (especially in a control environment). Example: signal acquisition, prepossessing data, main algorithm and result implementation/transmission. This means a failure in one part will not cause failures through the rest of the program. So while we are repairing the signal acquisition, the rest of tasks continues on stale data.

Everything needs CRCs. If you execute out of RAM, even your *.text* needs a CRC. Check the CRCs regularly if you are using a cyclical scheduler. Some compilers (not GCC) can generate CRCs for each section and some processors have dedicated hardware to do CRC calculations, but I guess that would fall out side of the scope of your question. Checking CRCs also prompts the ECC controller on the memory to repair single bit errors before it becomes a problem.

Use watchdogs for bootup, not just once operational. You need hardware help if your bootup runs into trouble.

Share Improve this answer Follow

edited May 10 at 15:15
Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Sep 23, 2016 at 7:07
Gerhard
7,031 ● 8 ● 55 ● 83

7

How about running many instances of your application. If crashes are due to random memory bit changes, chances are some of your app instances will make it through and produce accurate results. It's probably quite easy (for someone with statistical background) to calculate how many instances do you need given bit flop probability to achieve as tiny overall error as you wish.

Share Improve this answer Follow

answered Apr 25, 2016 at 16:40
ren
3,983 ● 9 ● 52 ● 100

2 Surely an embedded system would much prefer safety critical catches in one instance of a robust application than just firing off several instances, upping the hardware requirements and to some extent hoping on blind luck that at least one instance makes it through okay? I get the idea and it's valid, but lean more towards the suggestions that don't rely on brute force – WearyWanderer Apr 29, 2016 at 13:26

7

What you ask is quite complex topic - not easily answerable. Other answers are ok, but they covered just a small part of all the things you need to do.

As seen in comments, it is not possible to fix hardware problems 100%, however it is possible with high probabily to reduce or catch them using various techniques.

If I was you, I would create the software of the highest Safety integrity level level (SIL-4). Get the IEC 61513 document (for the nuclear industry) and follow it.

Share Improve this answer Follow

edited May 23, 2017 at 12:10
Community Bot
1 ● 1

answered Apr 26, 2016 at 12:03
BJовић
64.1k ● 45 ● 178 ● 279

11 Or rather, read through the technical requirements and implement those that make sense. A large part of the SIL standards is nonsense, if you follow them dogmatically you will end up with unsafe and dangerous products. SIL certification today is mainly about producing a ton of documentation and then bribe a test house. The SIL level says nothing about the actual safety of the system. Instead, you'll want to focus on the actual technical safety measures. There are some very good ones in the SIL documents, and there are some complete nonsense ones. – Lundin Apr 27, 2016 at 11:07

7

Someone mentioned using slower chips to prevent ions from flipping bits as easily. In a similar fashion perhaps use a specialized cpu/ram that actually uses multiple bits to store a single bit. Thus providing a hardware fault tolerance because it would be very unlikely that all of the bits would get flipped. So 1 = 1111 but would need to get hit 4 times to actually flipped. (4 might be a bad number since if 2 bits get flipped its already ambiguous). So if you go with 8, you get 8 times less ram and some fraction slower access time but a much more reliable data representation. You could probably do this both on the software level with a specialized compiler(allocate x amount more space for everything) or language implementation (write wrappers for data structures that allocate things this way). Or specialized hardware that has the same logical structure but does this in the firmware.

Share Improve this answer Follow

answered Apr 28, 2016 at 3:34

Alex C
513 ● 1 ● 4 ● 10

---

7

Perhaps it would help to know does it mean for the hardware to be "designed for this environment". How does it correct and/or indicates the presence of SEU errors ?

At one space exploration related project, we had a custom MCU, which would raise an exception/interrupt on SEU errors, but with some delay, i.e. some cycles may pass/instructions be executed after the one insn which caused the SEU exception.

Particularly vulnerable was the data cache, so a handler would invalidate the offending cache line and restart the program. Only that, due to the imprecise nature of the exception, the sequence of insns headed by the exception raising insn may not be restartable.

We identified the hazardous (not restartable) sequences (like `lw $3, 0x0($2)`, followed by an insn, which modifies `$2` and is not data-dependent on `$3`), and I made modifications to GCC, so such sequences do not occur (e.g. as a last resort, separating the two insns by a `nop`).

Just something to consider ...

Share Improve this answer Follow

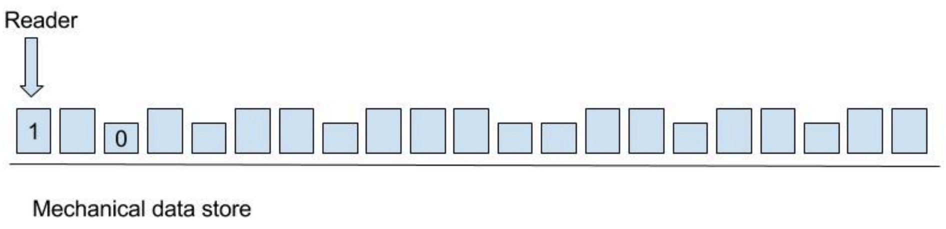
answered Apr 28, 2016 at 7:42

chill
16.8k ● 2 ● 41 ● 48

---

7

If your hardware fails then you can use mechanical storage to recover it. If your code base is small and have some physical space then you can use a mechanical data store.



There will be a surface of material which will not be affected by radiation. Multiple gears will be there. A mechanical reader will run on all the gears and will be flexible to move up and down. Down means it is 0 and up means it is 1. From 0 and 1 you can generate your code base.

Share Improve this answer Follow

edited May 4, 2016 at 17:30
Peter Mortensen
**31.6k** ● 22 ● 109 ● 133

answered Apr 27, 2016 at 14:24
Hitul
**357** ● 1 ● 5 ● 11

---

2 Perhaps an optical medium such as a CD-ROM would meet this definition. It would have the added bonus of a large capacity. – user5069935 Apr 28, 2016 at 12:45

2 Yes it will be similar but cd rom will use lesser but this will be fully mechanical system. – Hitul Apr 29, 2016 at 6:19

6

Firstly, **design your application around failure**. Ensure that as part of normal flow operation, it expects to reset (depending on your application and the type of failure either soft or hard). This is hard to get perfect: critical operations that require some degree of transactionality may need to be checked and tweaked at an assembly level so that an interruption at a key point cannot result in inconsistent external commands. **Fail fast** as soon as any *unrecoverable* memory corruption or control flow deviation is detected. Log failures if possible.

Secondly, where possible, **correct corruption and continue**. This means checksumming and fixing constant tables (and program code if you can) often; perhaps before each major operation or on a timed interrupt, and storing variables in structures that autocorrect (again before each major op or on a timed interrupt take a majority vote from 3 and correct if is a single deviation). Log corrections if possible.

Thirdly, **test failure**. Set up a *repeatable* test environment that flips bits in memory psuedo-randomly. This will allow you to replicate corruption situations and help design your application around them.

Share Improve this answer Follow

answered May 2, 2016 at 10:47

MrBigglesworth
385 ● 1 ● 9

3

Given supercat's comments, the tendencies of modern compilers, and other things, I'd be tempted to go back to the ancient days and write the whole code in assembly and static memory allocations everywhere. For this kind of utter reliability I think assembly no longer incurs a large percentage difference of the cost.

Share Improve this answer Follow

answered Apr 27, 2016 at 18:40
Joshua
43.1k ● 9 ● 78 ● 148

I'm a big fan of assembly language (as you can see from my answers to other questions), but I don't think this is a good answer. It's fairly possible to know what to expect from the compiler for most C code (in terms of values living in registers vs. memory), and you can always check to see that it's what you expected. Hand-writing a *large* project in asm is just a ton of extra work, even if you have developers that are very comfortable writing ARM asm. Maybe if you want to do stuff like compute the same result 3 times, writing some functions in asm makes sense. (compilers will CSE it away)
– Peter Cordes Apr 30, 2016 at 2:12

The higher risk otherwise that has to be balanced against it is upgrading the compiler can leave you with unexpected changes. – Joshua Apr 30, 2016 at 2:38

Here are huge amount of replies, but I'll try to sum up my ideas about this.

2

Something crashes or does not work correctly could be result of your own mistakes - then it should be easily to fix when you locate the problem. But there is also possibility of hardware failures - and that's difficult if not impossible to fix in overall.

I would recommend first to try to catch the problematic situation by logging (stack, registers, function calls) - either by logging them somewhere into file, or transmitting them somehow directly ("oh no - I'm crashing").

Recovery from such error situation is either reboot (if software is still alive and kicking) or hardware reset (e.g. hw watchdogs). Easier to start from first one.

If problem is hardware related - then logging should help you to identify in which function call problem occurs and that can give you inside knowledge of what is not working and where.

Also if code is relatively complex - it makes sense to "divide and conquer" it - meaning you remove / disable some function calls where you suspect problem is - typically disabling half of code and enabling another half - you can get "does work" / "does not work" kind of decision after which you can focus into another half of code. (Where problem is)

If problem occurs after some time - then stack overflow can be suspected - then it's better to monitor stack point registers - if they constantly grows.

And if you manage to fully minimize your code until "hello world" kind of application - and it's still failing randomly - then hardware problems are expected - and there needs to be "hardware upgrade" - meaning invent such cpu / ram / ... -hardware combination which would tolerate radiation better.

Most important thing is probably how you get your logs back if machine fully stopped / resetted / does not work - probably first thing bootstap should do - is a head back home if problematic situation is entcovered.

If it's possible in your environment also to transmit a signal and receive response - you could try out to construct some sort of online remote debugging environment, but then you must have at least of communication media working and some processor/ some ram in working state. And by remote debugging I mean either GDB / gdb stub kind of approach or your own implementation of what you need to get back from your application (e.g. download log files, download call stack, download ram, restart)

Share Improve this answer Follow

answered Apr 28, 2016 at 7:06

TarmoPikaro
5,193 ● 2 ● 56 ● 67

1 Sorry, but the question is about radioactive environment where hardware failures will occour. Your answer is about general software optimization and how to find bugs. But in this situation, the failures aren't produced by bugs – jeb May 25, 2016 at 8:38

Yes, you can blame also earth gravity, compiler optimizations, 3rd party library, radioactive environment and so on. But are you sure it's not your own bugs ? :-) Unless proven - I don't believe so. I have ran once upon a time some firmware update and testing poweroff situation - my software survived all poweroff situations only after I've fixed all my own bugs. (Over 4000 poweroffs during night time) But it's difficult to believe that there was bug in some cases. Especially when we are talking about memory corruption. – TarmoPikaro May 25, 2016 at 8:56

1

I've really read a lot of great answers!

Here are my two cents: build a statistical model of the memory/register abnormality, by writing a software to check the memory or to perform frequent register comparisons. Further, create an emulator, in the style of a virtual machine where you can experiment with the issue. I guess if you vary junction size, clock frequency, vendor, casing, etc. would observe a different behavior.

Even our desktop PC memory has a certain rate of failure, which however doesn't impair the day-to-day work.

Share Improve this answer edited May 10 at 15:09

Follow

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Mar 7, 2018 at 18:41
user9016329

0

Interestingly nobody mentioned that compilers usually have some flags to make the final binaries more secure. Usually, those actions apply to cyber security threats, but if we use our imagination, a 'random bit flip' might be a cyber attack (null terminated strings aren't null terminated anymore).

It was tagged with `gcc`, so here is a relevant answer how to make a binary more secure.
Most relevant here:

> - `-mmitigate-rop` Attempt to compile code without unintended return addresses, making ROP just a little harder.
> - `-fstack-protector-all` You need -fstack-protector-all to guarantee guards are applied to all functions, although this will likely incur a performance penalty. Consider -fstack-protector-strong as a middle ground. The -Wstack-protector flag here gives warnings for any functions that aren't going to get protected.

> - `-D_FORTIFY_SOURCE=2` Buffer overflow checks. See also difference between =2 and =1.
> - `-Wl,-z,relro,-z,now` RELRO (read-only relocation).
> - `-Wl,-z,noexecstack` Non-executable stack.

These options either give compiler warning (that you can make error), or a runtime error, preventing the damaged program to cause havoc.