Self-testing in embedded systems: Software failure

Colin Walls (/user/Colin Walls)

FEBRUARY 23, 2016

Share 10 G+ Tweet Like 4

(mailto:?subject=Self-testing in embedded systems: Software failure&body=http://www.embedded.com/design/debug-and-optimization/4441376/Self-testing-in-embedded-systems--Software-failure)

All electronic systems carry the possibility of failure. An embedded system has intrinsic intelligence that facilitates the possibility of predicting failure and mitigating its effects. This two-part series reviews the options for self-testing that are open to the embedded software developer, along with testing algorithms for memory and some ideas for self-monitoring software in multi-tasking and multi-CPU systems. In part one (http://www.embedded.com/design/debug-and-optimization/4441375/Self-testing-in-embedded-systems--Hardware-failure), we looked at self-testing approaches to guard against hardware failure. Here in part two, we look at self-testing methods that address software malfunctions.

As was mentioned in the introduction to part one, the acceptance of possible failure is a key requirement for building robust systems. This is extremely relevant when considering the possibility of software failure. Even when great care has been taken with the design, testing and debugging of code, it is almost inevitable that undiscovered bugs lurk in all but the most trivial code. Predicting a failure mode is tough, as this requires knowledge of the nature of the bug that leads to the failure and, if that knowledge were available, the bug would have been expunged during development.

The best approach is to recognize that there are broadly two types of software malfunction: data corruption and code looping. Some defensive code can be implemented to detect these problems before too much damage is done.

Data corruption
Arguably the most powerful feature of the C language is also the most common cause of errors and faults: pointers. Data is most likely to become corrupted if it is written via a pointer. The problem is there is no easy way to detect an invalid pointer. If the pointer is NULL, a dereference results in a trap, so ensuring that a suitable trap handler is installed is a start. A similar trap can handle the situation where an invalid (non-existent) memory address is presented by a pointer. However, if the address is valid, but incorrect, random errors may occur.

A memory management unit (MMU) provides some options to trap erroneous situations, as it gives software control over what memory is considered to be valid at any given time. The classic use of an MMU is with a process model operating system. In this context, the

Download Datasheets

PART NUMBER
e.g. LM317 Search

(https://www.opendatasheets.com/?utm_medium=SearchWidgetLogo&utm_source=embedd

MOST COMMENTED

07.06.2017
Want a robot to iron your shirts? (/electronics-blogs/max-unleashed-and-unfettered/4458601/Want-a-robot-to-iron-your-shirts-)

06.19.2017
Saving power with relays and solenoids (/electronics-blogs/without-a-paddle/4458541/Saving-power-with-relays-and-solenoids)

07.10.2017
A solution looking for a problem (/electronics-blogs/without-a-paddle/4458611/A-solution-looking-for-a-problem)

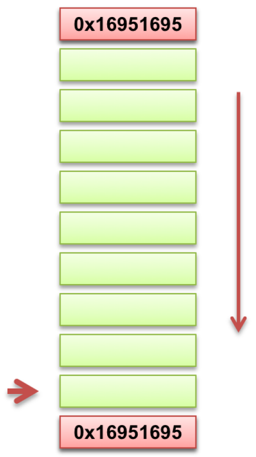
07.14.2017
The Write Stuff: Tutorial template in high-visibility format (/electronics-blogs/the-write-stuff/4458625/The-Write-Stuff--Tutorial-template-in-high-visibility-format)

07.03.2017
Public key infrastructure for the IIoT (/electronics-blogs/say-what-/4458599/Public-key-infrastructure-for-the-IIoT)

code of each task can only access the memory specifically allocated to it. Any attempt to access outside of this area causes an error.

There are two special cases where there is a chance to detect pointer errors: stack overflow/underflow and array bound violation.

Stack space allocation is something of a black art. Although there are static analysis tools around that can help, careful testing during development is wise. This may involve filing the stack with a “fingerprint” value, and then looking at utilization after some period of code execution, or write access breakpoints may be employed. Runtime checks for stack usage are often sensible. This simply requires the addition of “guard words” at either end of the allocated stack space. These are pre-loaded with a unique value, which can be recognized as being untouched. It is logical to use an odd number (as addresses are normally even) and avoid common values like 0, 1 and 0xffffffff. There is then a 4 billion to 1 chance of a false alarm. Like memory tests, the guard words can be checked from a background task or whenever the CPU has nothing better to do. Another possible way to monitor the guard words would be with an MMU that has a fine grain resolution, but such functionality is not common.



In some languages, access to arrays is carefully controlled so that accesses can only occur validly within their allocated memory space. One reason why there is normally no checking in C is that this would introduce a runtime overhead every time an array element if accessed, which is likely to be unacceptable. This could be implemented in C++ by overloading the **[ ]** (array index) operator. However, it would still be possible to make an erroneous access because pointers can be used instead of the array index operator. Normal array element access like this:

```
arr[3] = 99;
```

Can also be written thus:

```
*(arr+3) = 99;
```

However, the most common array access problem is accidentally iterating off of the end, thus:

```
int arr[4];

for (i=0; i<=4; i++)
    arr[i] = 0;
```

To detect this kind of error, guard words, like those used with a stack, may be used

Next page >> (http://www.embedded.com/design/debug-and-optimization/4441376/2/Self-testing-in-embedded-systems--Software-failure)

Want more Embedded? Check out these recent articles:

- The scheduler - options and context save (http://embedded.com/design/operating-systems/4458662/The-scheduler---options-and-context-save?_mc=EMB_FT_DEV_01)
- Is the surveillance industry ripe for a storage change? (http://embedded.com/electronics-blogs/say-what-/4458652/Is-the-surveillance-industry-ripe-for-a-storage-change-?_mc=EMB_FT_DEV_01)
- New version of MIPI CSI-2 looks (literally) beyond mobile phones (http://embedded.com/electronics-blogs/say-what-/4458649/New-version-of-MIPI-CSI-2-looks--literally--beyond-mobile-phones?_mc=EMB_FT_DEV_01)
- The second wave of smart speakers is coming (http://embedded.com/electronics-blogs/say-what-/4458643/The-second-wave-of-smart-speakers-is-coming?_mc=EMB_FT_DEV_01)
- Bluetooth mesh solution cuts time-to-market (http://embedded.com/electronics-blogs/max-unleashed-and-unfettered/4458637/Bluetooth-mesh-solution-cuts-

RELATED CONTENT

02.23.2016 | DESIGN
Self-testing in embedded systems: Software failure (/design/debug-and-optimization/4441376/Self-testing-in-embedded-systems--Software-failure)

02.09.2016 | DESIGN
Self-testing in embedded systems: Hardware failure (/design/debug-and-optimization/4441375/Self-testing-in-embedded-systems--Hardware-failure)

04.16.2015 | DESIGN
Testing code that uses an RTOS API (/design/prototyping-and-development/4439214/Testing-code-that-uses-an-RTOS-API)

11.25.2013 | DESIGN
Vintage multicore (/design/mcus-processors-and-socs/4424833/Vintage-multicore)

03.15.2014 | DESIGN
Multicore basics: AMP and SMP (/design/mcus-processors-and-socs/4429496/Multicore-basics)

time-to-market?_mc=EMB_FT_DEV_01)

Join over 2,000 technical professionals and embedded systems hardware, software, and firmware developers at ESC Minneapolis (http://escminn.com/?_mc=WE_EMB_FT) November 8-9 2017 and learn about the latest techniques and tips for reducing time, cost, and complexity in the development process.

Make sure to follow updates about ESC's talks, programs, and announcements via the Destination ESC blog (http://www.embedded.com/electronics-blogs/4219317/Destination-ESC) on Embedded.com and social media accounts Twitter (https://twitter.com/ESC_Conf), Facebook (https://www.facebook.com/EELiveShow), LinkedIn (https://www.linkedin.com/groups?gid=1403157), and Google+ (https://plus.google.com/110966938586722247542/posts).

Embedded.com, EE Times, EBN, EDN, and Planet Analog are owned by AspenCore. The Embedded Systems Conference is owned by UBM.

Share 10 G+ Tweet Like 4

(mailto:?subject=Self-testing in embedded systems: Software failure&body=http://www.embedded.com/design/debug-and-optimization/4441376/Self-testing-in-embedded-systems--Software-failure)

< Previous Page 1 of 2 Next > (/design/debug-and-optimization/4441376/2/Self-testing-in-embedded-systems--Software-failure)

5 COMMENTS

WRITE A COMMENT

Cdhmanning (/User/Cdhmanning) **POSTED: FEB 27, 2016 1:09 AM EST**

IMHO far too many people use the watchdog as a get out of jail card and use it to "fix" bad code/system design.

If I was designing a pacemaker, the first thing I would do is split it over two CPUs. One just does the heart rate control and the other does all the fancy stuff. That way if the fancy CPU goes out to lunch the core mission is still maintained. Micros cost less than 20c each. No excuse.

Resets often don't fix problems and can make them worse. I've seen code get into a reset loop due to a damaged sensor giving a bad reading and tripping guard code which ended up causing a reset. The system just ended up doing about 50 resets a second and nothing useful. A far better design would have been to structure the system to handle broken sensors and still keep going.

One area of robotic research that can help for this is behaviour based programming to use behaviours to keep delivering values even when sensors fail.

REPLY (/?SCREENTOVIEW=LOGIN&REDIRECTTO=HTTP://WWW.EMBEDDED.COM/DESIGN/DEBUG-AND-OPTIMIZATION/4441376/SELF-TESTING-IN-EMBEDDED-SYSTEMS--SOFTWARE-FAILURE)

Colin Walls (/User/Colin%20Walls) **POSTED: FEB 27, 2016 3:16 AM EST**

All good points @Cdhmanning
Maybe I did not make it clear enough that the use of a watchdog should be a last resort, not an excuse for sloppy code or bad system design.

REPLY (/?SCREENTOVIEW=LOGIN&REDIRECTTO=HTTP://WWW.EMBEDDED.COM/DESIGN/DEBUG-AND-OPTIMIZATION/4441376/SELF-TESTING-IN-EMBEDDED-SYSTEMS--SOFTWARE-FAILURE)

Cdhmanning (/User/Cdhmanning) **POSTED: FEB 28, 2016 4:25 PM EST**

Certainly...

A watchdog should only every be triggered by an "act of god": micro failure etc.

If code ever causes a watchdog to go off, you have problems.

Unfortunately far too many people use watchdog as a primary robustness mechanism.

REPLY (/?SCREENTOVIEW=LOGIN&REDIRECTTO=HTTP://WWW.EMBEDDED.COM/DESIGN/DEBUG-AND-OPTIMIZATION/4441376/SELF-TESTING-IN-EMBEDDED-SYSTEMS--SOFTWARE-FAILURE)

SteveMerrick (/User/SteveMerrick) **POSTED: MAR 24, 2016 1:28 PM EDT**

I think Colin made it clear that, sometimes, it is difficult to detect an error condition in any other way. In this case, the use of a watchdog is the difference between continuing wrong behaviour and a pause followed by the restoration of correct operation. In the case of a hardware (e.g. sensor) fault, it's difficult to decide how to deal with it. It can't be cured (by the firmware), so you can either reboot until it's fixed, or cease operation altogether, yes?

REPLY (/?SCREENTOVIEW=LOGIN&REDIRECTTO=HTTP://WWW.EMBEDDED.COM/DESIGN/DEBUG-AND-OPTIMIZATION/4441376/SELF-TESTING-IN-EMBEDDED-SYSTEMS--SOFTWARE-FAILURE)

Colin Walls (/User/Colin%20Walls) **POSTED: MAR 24, 2016 1:40 PM EDT**

Exactly.