Toyota's killer firmware: Bad design and its consequences

Michael Dunn - October 28, 2013

On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

Embedded software used to be low-level code we'd bang together using C or assembler. These days, even a relatively straightforward, albeit critical, task like throttle control is likely to use a sophisticated RTOS and tens of thousands of lines of code.

With all this sophistication, standards and practices for design, coding, and testing become paramount – especially when the function involved is safety-critical. Failure is not an option. It is something to be contained and benign.

So what happens when an automaker decides to wing it and play by their own rules? To disregard the rigorous standards, best practices, and checks and balances required of such software (and hardware) design? People are killed, reputations ruined, and billions of dollars are paid out. That's what happens. Here's the story of some software that arguably never should have been.

For the bulk of this research, EDN consulted Michael Barr, CTO and co-founder of Barr Group, an embedded systems consulting firm, last week. As a primary expert witness for the plaintiffs, the in-depth analysis conducted by Barr and his colleagues illuminates a shameful example of software design and development, and provides a cautionary tale to all involved in safety-critical development, whether that be for automotive, medical, aerospace, or anywhere else where failure is not tolerable. Barr is an experienced developer, consultant, former professor, editor, blogger, and author.

Barr's ultimate conclusions were that:

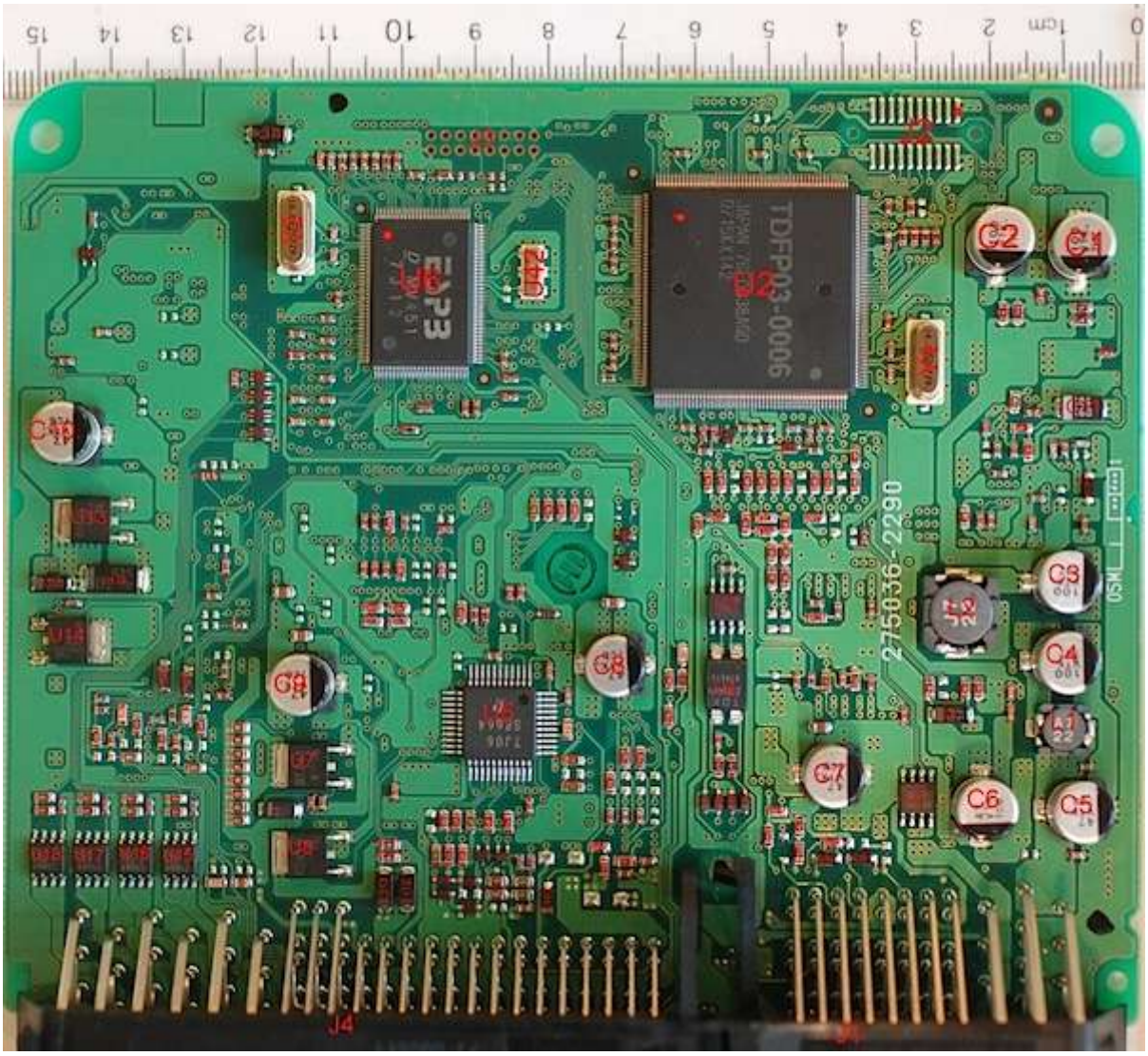
- Toyota’s electronic throttle control system (ETCS) source code is of unreasonable quality.
- Toyota’s source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
- Code-quality metrics predict presence of additional bugs.
- Toyota’s fail safes are defective and inadequate (referring to them as a *“house of cards” safety architecture).*
- Misbehaviors of Toyota’s ETCS are a cause of UA.

A damning summary to say the least. Let's look at what lead him to these conclusions:

Hardware

Although the investigation focused almost entirely on software, there is at least one HW factor: Toyota claimed the 2005 Camry's main CPU had error detecting and correcting (EDAC) RAM. It didn't. EDAC, or at least parity RAM, is relatively easy and low-cost insurance for safety-critical systems.

Other cases of throttle malfunction have been linked to tin whiskers in the accelerator pedal sensor. This does not seem to have been the case here.



The Camry ECM board. U2 is a NEC (now Renesas) V850 microcontroller.

Software

The ECM software formed the core of the technical investigation. What follows is a list of the key findings.

Mirroring (where key data is written to redundant variables) was not always done. This gains extra significance in light of …

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's

investigation showed that 94% was closer to the truth. On top of that, stack-killing, MISRA-C rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

Two key items were not mirrored: The RTOS' critical internal data structures; and—the most important bytes of all, the final result of all this firmware—the TargetThrottleAngle global variable.

Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Toyota's ETCS used a version of OSEK, which is an automotive standard RTOS API. For some reason, though, the CPU vendor-supplied version was not certified compliant.

Unintentional RTOS task shutdown was heavily investigated as a potential source of the UA. As single bits in memory control each task, corruption due to HW or SW faults will suspend needed tasks or start unwanted ones. Vehicle tests confirmed that one particular dead task would result in loss of throttle control, and that the driver might have to *fully remove their foot from the brake during an unintended acceleration event* before being able to end the unwanted acceleration.

A litany of other faults were found in the code, including buffer overflow, unsafe casting, and race conditions between tasks.

Thousands and thousands

Thousands and thousands

The Camry ETCS code was found to have 11,000 global variables. Barr described the code as “spaghetti.” Using the Cyclomatic Complexity metric, 67 functions were rated untestable (meaning they scored more than 50). The throttle angle function scored more than 100 (unmaintainable).

Toyota loosely followed the widely adopted MISRA-C coding rules but Barr’s group found 80,000 rule violations. Toyota's own internal standards make use of only 11 MISRA-C rules, and five of those were violated in the actual code. MISRA-C:1998, in effect when the code was originally written, has 93 required and 34 advisory rules. Toyota nailed six of them.

Barr also discovered inadequate and untracked peer code reviews and the absence of any bug-tracking system at Toyota.

NASA, which was involved in an earlier investigation, discussed in its report the five fail-safe modes implemented in the ETCS. They comprise three limp-home modes, RPM limiting, and finally, engine shutdown. All fail-safes are handled by the same task. What if that task dies or malfunctions?

Watchdog

Many embedded systems use watchdog timers to rein in errant processors; in safety-critical systems, they're mandatory. But as systems increase in complexity, the watchdog subsystem must mirror that complexity.

Ideally in a multitasking system, every active task should be required to "check in" to the watchdog. In the Toyota ETCS, the watchdog was satisfied by nothing more than a timer-tick interrupt service routine (ISR). A slow tick. If the ISR failed to reset the watchdog, the ETCS could continue to malfunction due to CPU overload for up to 1.5s before being reset. But keep in mind that for the great majority of task failure scenarios, the ISR would continue happily running along without resetting the controller.

It was also found that most RTOS error codes indicating problems with tasks were simply ignored – a definite MISRA-C violation.

Who watches the watcher?

Toyota's ETCS board has a second processor to monitor the first. The monitor CPU is a 3rd-party part, running firmware unknown to Toyota, and presumably developed without any detailed knowledge of the main CPU's code.

This is potentially a good thing, as it would be a truly independent overseer. This chip communicates with the main CPU over a serial link, and also contains the ADC that digitizes the accelerator pedal position.

Anyone working with safe systems knows that single points of failure are to be avoided at almost any cost, yet here is one – the single ADC that feeds both CPUs their vehicle state information.

Also, the failsafe code in this monitor CPU relies on the proper functioning of a main CPU task Barr identified to the jury only as "Task X" (due to secrecy rules surrounding the source code itself), an arguably outsize task handling everything from cruise-control to diagnostics to failsafes to the core function of converting pedal position to throttle angle. Task X could be viewed as another single point of failure.

Resolutions

What can be learned from this story of software gone wrong? Here are some thoughts, inspired by Toyota's experience:

- It all starts with the engineering culture. If you have to fight to implement quality, or conversely, if others let you get away with shoddy work, quality cannot flourish. The culture must support proper peer review, documented rule enforcement, use of code-quality tools and metrics, etc.
- In complex systems, it's impossible to test all potential hardware- and software-induced failure scenarios. We must strive to implement all possible best practices, and use all the tools at our disposal, to create code that is failure-resistant *by design*.
- Use model-based design where suitable.
- Use tools with the proper credentials, not an uncertified RTOS as was done here.
- The system must undergo thorough testing by a separate engineering team. Never make the mistake of testing your own design. (To be true, Toyota's overall test strategy was not specifically described.)
- The underlying hardware must work with the firmware to support reliability goals:
  - Single points of failure, in HW and SW, must be avoided.
  - Architectural techniques that contribute to reliability, such as lockstep CPUs, EDAC memory, properly implemented watchdogs, MMU to implement full task isolation and protection, must be

implemented.
    - A thorough FMEA to characterize failure modes and guide design improvements should be employed.

Are you involved with safety-critical devices? If so, are you satisfied with the quality processes and culture at your company? What are your thoughts on Toyota's design work and the investigation's findings?

Below is an interview with Michael Barr after his EE Live! keynote "Killer Apps: Embedded Software's Greatest Hit Jobs".

Also see:

- The mythical software engineer
- Toyota Underestimated 'Deadly' Risks, EE Live! keynoter says
- Toyota Case: Single Bit Flip That Killed
- Why Toyota's Oklahoma Case Is Different
- Acceleration Case: Jury Finds Toyota Liable
- Toyota, drive by wire, and our failure to learn from experience
- Toyota learns the tyranny of software complexity
- Toyota fined for accelerator pedal sticking, April 5, 2010
- Toyota accelerations revisited—hanging by a (tin) whisker
- Unintended acceleration and other embedded software bugs
- Firmware forensics: best practices in embedded software source-code discovery
- Dead code, the law, and unintended consequences
- Unintended acceleration

Resources:

- Tin whisker analysis of an automotive engine control unit (published study)
- NASA Tin Whisker page
- Electrical Failure of an Accelerator Pedal Position Sensor Caused by a Tin Whisker and Discussion of Investigative Techniques Used for Whisker Detection
- How "brake override" stops runaway cars (Consumer Reports video)

Follow Michael Dunn at