Key Learnings from Past Safety-Critical System Failures

Ever had a DVD player freeze? A mobile phone crash and reboot? A home router that required a reset? Welcome to the 21st century, where every device has at least one processor. Without question, our daily lives are enhanced by the embedded systems around us. And many of us take for granted that these devices will do their intended jobs tirelessly and correctly, day in and day out, without fail.

But, not all embedded systems are blenders and DVD players. Just today, your life was in the hands of multiple embedded systems. From traffic light controls and automotive controls, to life-saving medical devices, and avionics and energy production, embedded systems are entrusted to perform correctly and keep us safe. Yet, many safety-critical devices do not operate correctly 100% of the time.

Here we examine some of the more notable firmware failures, describing the products, the defects, the root causes and what could have been done better. We'll discuss what we've learned, where we are today, and what the future may hold. We've all heard the quote, "Those who cannot remember the past are condemned to repeat it." So, why is it that when it comes to safety- or mission-critical firmware, we seem to have amnesia?

The remainder of this page is a transcript of a 1-hour webinar. A recording of the webinar is available at https://vimeo.com/105886749.

Slide 1: Safety-Critical Firmware: What can we learn from past failures?

Good afternoon and thank you for attending Barr Group's Webinar on “Safety Critical Firmware: What can we learn from past failures?” My Name is Jennifer and I will be moderating today’s Webinar. Our presenters today will be Michael Barr Chief Technical Officer at Barr Group and Dan Smith Principal Engineer of Bar Group.

Today’s presentation will last for approximately 40 minutes after which there will be a monitored question and answer session. To submit a question during the event, type your question into the rectangular space at the bottom right and click the send button. Questions will be addressed at the conclusion of the presentation.

At the start of the webinar check to be sure that your audio speakers are on and then the volume is turned up. Please make sure to close out all background programs and turn off anything that might affect your audio feed during the webinar. I am pleased to present Michael Barr and Dan Smith’s Webinar. Safety-Critical Firmware “What can we learn from past failures?”

Slide 2: Michael Barr, CTO

Thank you and welcome everyone. I'm Michael Barr the Chief Technical Officer of the Barr Group.

Slide 3: Barr Group - The Embedded Systems Experts

Our company helps other companies make their embedded systems safer and more secure. What I mean by that is that we don't make a products of our own rather we help client companies make medical devices and industrial controls and consumer electronics and other types of products and we do that in various different ways. We do actual product development work were we take on some or all of the mechanical, electrical and software design to assists companies.

And we also provide engineering guidance and consulting to engineering managers and directors and vice-presidents who are interested in leading change in their organizations to improve the process or to re-architect or architect the embedded systems and embedded software always with the focus on making them, ultimately more reliable systems, safer and more secure. And in addition to that we also have a mission of training; in part of that is motivated to our desire to improve as many embedded systems and make as many embedded systems safer more reliable as possible. We deliver that training both in private settings at companies and also publicly such as at our upcoming public boot camps.

Slides 4-5: Upcoming Public Courses

Before we get to today's course, I just want to alert you that we regularly do trainings both at private companies and you can see on our website a list of all the courses that we offer. And we also have public trainings including some upcoming boot camps. These are our weeklong, four and a half day, hands-on, intensive software training courses.

The course titles, dates, and other details for all of our upcoming public courses are available at the Training Calendar area of our website.

Slide 6: Dan Smith, Principal Engineer

The final thing I would like to do before get started is to introduce today's speaker. Mr. Dan Smith is a principal engineer at the Barr Group. He has more than twenty years of experience designing embedded hardware and software for a variety of different industries and using a variety of different tools in terms of real-time operating systems processors and development tools, et cetera. But always with a focused on secure and safe systems and with that I introduce Dan and let him take it from here. Thank you.

Slide 7: Overview of Today's Webinar

Thanks Michael and thank you Jennifer as well. So welcome everybody, thank you for joining us for this webinar today and I'd like to give you a quick overview of what we're going to be talking about. So what we're going to do is take a look at four specific case studies where software had a significant role in the failure of a critical system and we're doing that because we want to examine what the root cause was and understand how that problem could have been prevented.

As you will see, in retrospect when we look at what the root cause at these problems were, it'll seem pretty obvious what could have been done to prevent it but obviously it's not as obvious as we would think because otherwise we wouldn't be continuously repeating these problems over and over again. And ultimately it's going to turn out that the answer is a combination of education and knowledge, training, process, good software process, and remaining vigilant towards these bugs and not getting lazy

or complacent. As far as the content of this webinar it's useful if you have a working knowledge of C or C ++ but that's not absolutely necessary.

Slide 8: Critical Systems

Considering the title of this webinar it's seems only appropriate that we should define what we mean by a critical system. A pretty standard definition for a critical system is a system whose malfunction can cause injury or death. We can all relate to that, things like avionics, medical devices things like that. We also want to acknowledge the fact that there are a lot of industries that might not be able considered critical per say, things like high security systems, high availability systems, even mission critical systems that are unmanned that might not cause injury or death for example; unmanned underwater exploration or unmanned space exploration but still the cost of failure in a mission like this is very high. So we want to include those on our definition as well.

Also I want to mention that there are all sorts of failure modes but generally failures of critical systems, there's one of two things that happens: either the product takes action when it shouldn't or it does the wrong thing or it fails to do the appropriate action at the right time. So for example; you might have a medical device that delivers too much radiation or you might have a collision avoidance system that instead of steering away from a collision steers directly into it a collision. That would be something that the product actively did.

You can also have failures that are the opposite of that where the product fails to perform its functionality. Things like an airbag failing to deploy when it should or lifesaving medical device failing to deliver the correct therapy at the right time or perhaps the gas detector that doesn't warn when dangerous level of the gas have built up.

Slide 9: Looking Through a Keyhole

Before we proceed I just want to acknowledge up front that the engineering of critical systems is an extremely rigorous process, it's a multi-disciplinary process and we're only going to covering a very small slice of the entire development process; specifically the firmware engineering part. Even then we're going to be talking about the implementation phase. So not firmware architecture, not the design process, not all the testing techniques.

Slide 10: Role of Firmware in Critical Systems

One of the reasons we're focused entirely on firmware in this webinar is the fact that firmware is playing an increasingly important role in critical systems today. Everything from transportation whether it's automobiles, locomotives, avionics things like that, mobile electronics, security systems, medical devices which are in charge of sustaining health or even saving lives et cetera., we could go on and on. I'm sure we have a very large range of industries represented in our webinar audience today.

The reality is that more and more functionality is being pushed in the firmware and as you guys know the more functionality you put in the complexity increases, as the complexity increases so does the potential for bugs. Just think about it many of you probably drove to work today. If you thought about how many times your life was in the hands of a critical system, you'd probably never get out of bed in the morning. Just in your vehicle, just think of how many computers, how many processors there are

plus you have traffic lights; you have other vehicles on the road. There are all sorts of systems that are interacting.

Slide 11: Ripped from the Headlines

So today we're going to be looking at four case studies from the past. But before we jump in to those I very briefly want to mention something that just recently happened. This was late August, 2014. Some of you might have heard about this, I've included a screen shot from the website along with the URL so you can look at it from a Russian news agency. Many of you might be familiar with Galileo; this is Europe's global positioning system.

It's a $13 Billion dollar thirty satellite network and it's being built right now. There are four satellites in orbit. Just recently satellites 5 and 6 were supposed to be put up into their orbits. This was late August as I said and unfortunately due to a problem the satellites where put in a wrong orbit and now most likely there stranded in an unusable orbit and Scientist might be able to do something to "Rescue these." It won't be the first time that a rabbit has been pulled out of a hat so to speak when something like this is happened but it's also very possible that the satellites will have to be destroyed.

What actually happened was, the satellites failed to reach their intended orbit and it's believed or at least it's been alleged that this was cause by software errors and the Frigate M - T rockets upper stage. So according from the new story, the non-standard operation was likely caused by an error in the embedded software operating in full accordance with the embedded software, the rocket has delivered the units to the wrong destination. So that's a quote from the Russian new agency.

Slide 12: Hindsight

Finally before we get in to the four cases I do want to say one other thing. As we're looking at these case studies many of you will probably say to yourselves "How did they let this happened, how could this possibly have escaped in to the field" of course in hindsight the problem is obvious and it seems like it should have been caught but the reality is that the same mistakes happen over and over again by smart people. So clearly it's not as obvious as it might seem.

This is the same thing with security. I spent a lot of time doing security in Barr Group. We do a lot of stuff with security and many of you who work with security know it's a small sub set of problems that cause the majority of security breaches, things like buffer overflows. So clearly this is something that continues to plague us.

The point of this webinar is not the point the finger and criticizes or taunt or say, "ha-ha." We all are capable of making these kinds of mistakes. The point of this webinar is just to educate ourselves and hopefully develop a greater awareness so that we have a less of a likelihood of repeating the same problems in the future.

Slide 13: Therac-25

Okay so our first case study we're going to talk about the Therac-25, which was a radiation therapy machine obviously for cancer treatment. Its predecessors date to the 1970's. The time frame we're talking about now is early to mid-1980's. Without getting into a lot of details about the architecture it was essentially an embedded deck-PDP 11 mini computer. This device – this machine generated

radiation that was sent to the patient in order to destroy cancerous tumors. It had two modes of operation; the first was direct electron beam therapy which uses a low power electron bean whose energy was spread using scanning magnets. The second mode of operation which was a newer mode operation was called megavolt x-ray mode and this delivered much higher energy x-rays to the patient.

Slide 14: Therac: What Happened?

So what happen with the Therac-25? Well to give you a complete discussion on this would take an hour, so I'm going to try to cut to the chase and get right to the core defect here; first is there's a race condition that allowed the high powered x-ray beam to be activated in on without this beams spreader or diffuser plate in place to distribute the energy. So in other words you had this high-powered x-ray beam going right into the patient and not being diffuse at all to due to a race condition.

And there's a second software problem that was responsible for another failure mode which resulted massive radiation overdosing. The software used in eight bit unsigned counter are as parameters were entered into the machine and this was continuously being checked against the prescription. Once the parameter is matched up the counter was reset to zero and the treatment could be started but here's the tragic flaw. Obviously the maximum value of this counter is 255 but what would happen if the software incremented that counter deliberately of course when it holds the value of 255?

Well of course it's going to wrap around to zero that's well defined and just as we discussed when this magic value of zero is in the counter this indicates that we're ready to begin the treatment. Albeit in this case with the wrong parameters. So if the operator started treatment when the counter was accidentally overflowed and incremented to a value of zero you can imagine the tragic results that would occur.

Slide 15: Therac: Findings

Well after all this unfolded in addition to finding the root cause of a couple of these bugs that we just discussed. A bunch of other troubling other problems, defects, breakdowns were discovered. First it was deemed that the software development process was woefully inadequate and immature essentially resulting an un-testable software. They also determined that the reliability modelling was very incomplete and the failure mode analysis was also very inadequate.

There was no independent review of the critical software. In fact it's not clear just how detailed the review was even within the organization. There's improper reuse of software from older models when it wasn't a direct carry over. Lastly one of the issues that resulted in the raise condition was even though a multi-tasking operating system was used there was improper inter task synchronization thus leading to one of the problems.

A couple other points of note here; first is the system was implemented entirely in assembly language and the second point dimension is that the system used its own in-house operating system. Now granted both of these were more common 30 years ago. It was not unheard of either of these techniques but I think today we can all agree either of these practices would be frowned upon.

Slide 16: Ariane 5 / Flight 501

Our second case study is going to focus on Ariane 5 rocket. And specifically were going to talk about its maiden flight which is in June 1996, also known as Flight 501.

A little background on Ariane 5, it was the successor to the smaller Ariane 4 rocket. Ariane 5 was designed to carry much bigger, heavier payloads and in fact today it's really the standard launch vehicle for the European Space Agency. The payload for this maiden flight was named Cluster and it consisted of four fairly heavy each about 1,200 kilogram spacecraft and the mission for these spacecraft was to study the earth's magnetosphere.

Unfortunately as we're about to see this payload never made it into space and the mission was scrapped.

Slide 17: Flight 501 Failure

So what actually happened? How did Fight 501 fail? We'll talk about the root cause on the next slide but first I want to describe what actually happened.

About 37 seconds into the launch both inertial navigation systems, it had two systems on there, two computers running the same software. They crashed, they were on the same software so the same defect that resulted one crashing resulted the other one crashing, no surprise in hindsight. This caused the thrusters to steer or swivel into extreme positions which were absolutely not correct for the flight path that was indented.

So the vehicle parted from its intended flight path and especially at the speeds the rocket was travelling at this put, extreme stresses on the rocket and mechanically speaking the vehicle actually began to break up and this triggered the onboard flight termination system. So it deliberately self-destructed, thank goodness, but then result was that the mission was at failure and the cause including the destroyed payloads was approximately $370 Million dollars.

Slide 18: Flight 501 - Cause

Now let's talk about the cause of the failure. The Inertial navigation system on Ariane 5 was reused from Ariane 4 but unfortunately the assumptions about Ariane 4 didn't necessarily apply to Ariane 5. Flight 501 had a much greater horizontal velocity and this value was tracked in a 64-bit floating point value. In the software this needed to be converted and stored into a 16-bit integer but because of the greater velocity there was an overflow when this conversion took place.

And another tragic thing is that the overflow checks that could have caught this and tried to react or recover from this were omitted for efficiency. Something I want to point out here is that the implementation language here was not C even though this could have happen in the C programming language but actually the programming language here was Ada and this just goes to show you even though the language like Ada is renowned for being a more safe language a lot of avionics and things like that are implemented in Ada. It just goes to show you certain kinds of defects – the programming language can't keep you from making.

Another sad one about this whole thing is that the software that had this error which caused the thrusters to swivel under the wrong direction and put the rocket off track – this was not even needed after launch. So this software such not have even been running along the rocket while have been launched.

Slide 19: MISRA C: 2012

So I mention on the previous slide that some of this runtime checking that could have caused the overflow. This checking was admitted for reasons of efficiency. I want to point out is that using a coding standard such as Misra C: 2012 I suspect that everybody listening today is familiar with Misra C – the Misra C coding guidelines but if you look at the coding guidelines I'm talking about the 2012 version. Directive 4.1 For example says “Run-time failures shall be minimized.”

You could probably rephrase that or get a little bit more verbose and say, “Code should be written to anticipate and defend against run-time errors by adding run-time checks throughout the code.” We all know that C's run-time environment is very light-weight, that's what makes the language so efficient and so compact. But you can have things like unchecked array accesses, divided by zero errors, issues with dynamic memory allocation et cetera. Presumably most of you working critical systems don't have dynamic allocation in the first place. But what I'm saying is that with C's very light-weight run-time environment, the burden is actually on you, the programmer to do this kind of error checking. The run-time is not going to trap these kinds of issues and then put you into safe-mode. So, effectively the tactic you really need to adopt is to implement extensive run-time checking in your code.

Slide 20: Assertions

So, how do we perform this dynamic checking at run-time? Well, most programming languages including the C programming language actually have a built-in mechanism for this. So, with C and C++ there's a macro called assert() and you get it by including the correct pattern file and by using this you can confirm that your assumptions at run-time are actually being adhered to. The expression pass to assert() is expected to always evaluate to true. Let's look at the example here we have one in code; we have a function called 'isInRange'.

You pass three parameters a lower bound and upper bound, and a value you want to check against. Now, presumably when you call 'isInRange', lower_bound should be less than or equal to upper_bound it doesn't make any sense for the lower _bound to be greater than the upper_bound. So, the first thing that this routine does is checked just to make sure that the lower_bound is less than or equal to upper_bound. If that's not the case, there's no meaning for comparison you can check to see if something is in range.

Slide 21: Removing Assertions

So, I just been advocating using assertions to do – to perform this dynamic run-time checking, but I need to aknowledge that there is a cost to using them. There's both a run-time cost in other word is extra CPU processing that has to go on and this is going to increase your code size because these assertion checks do result and greater code.

My personal belief is that you should never disable these assertions even in your production code. This does mean that you need to budget enough resources (CPU and memory) to run all assertions enabled it

at all times. Presumably if you're using assertions, you're going to use them during testing and during debugging, but what a lot of organizations do is by the time they go to production, they disable assertions and that's very easily done usually through just a single command line compiler switch.

Presumably the thinking is that by the time you're going to production, none of the assertions are triggering, you feel pretty good about it so you can go ahead and remove them. In my opinion – and other people that I respect a great deal – this only makes sense (disabling assertions for production code) if you believe that testing catches all problems and I think we can all agree that testing does not catch all problems. So, the fact that your assertions are no longer triggering or firing in the code in your lab does not mean that on the field but especially with a lot of deployed devices and environments you might not have anticipated. It does not mean that those assertions won't trigger.

We're get tricky as if you're very resource constraint which does happen in the embedded world and you're running into a barrier either your CPU limited or memory limited and obvious candidate is just go ahead and disable assertions. I still say that I think that's just almost that never the right thing to do. Now we feel strongly enough about the importance of keeping assertions enable them in your code, if you have that possibility that if you're still not convinced, we're going to give you a couple of resources towards the end of this webinar from people whose names you might know and hopefully this will convince you of the importance of keeping these assertions enabled. Because it's really one of the most important things you can do to keep your software from going off the rails.

Slide 22: Flight 501 - Lessons

So, what are some of the key can take a ways or lessons learned from Flight 501? Well, one of the most important is that re-use of software can be hazardous when you're deciding to re-use software from our previous project, the burden is on you, the developer, to ensure that it will function properly in the new environment. That's not always easy as easy as it sounds, we have two examples of that the Therac-25 and Flight 501. Another issue that we actually covered in our previous webinar on coding standards is that mixing data types and expressions here in this case, it's mixing float and integer. It can be very hazardous and it can be a source of very subtle and difficult to define problems.

Another perhaps obvious lesson is there's no reason to be executing unnecessary software. As I mentioned the software, the cause that self-destruction of the Ariane 5, didn't even need to be running in the first place. I personally believe that one of the most important lessons, if not the most important lesson from Flight 501, is that you disabled assertions in your production code at your own peril. In other words, if you have the options, if you have the CPU and code space to keep those assertions enabled, I cannot recommend strongly enough that you keep them in your code. And obviously, it goes outside of the discipline of firmware engineering, it's extremely important to consider the failure modes of the product that you working on and how those can interact.

Slide 23: What About Testing?

So, just a brief word here about testing before we move on to our next case study. Obviously, all of these systems that we're looking at were tested. They underwent significant testing and obviously testing is extremely important but it's not sufficient for proving the correctness of your product.

Testing can never prove the absence of bugs. We've seen that over and over again, bugs always escape to the field and these are often the ones are that are very difficult to reproduce. And another thing to

keep in mind is that tests typically are software as well and there can be bugs in your test code as well. So, the most important thing to remember is simply that tests are just one part of an overall strategy.

Slide 24: Patriot Missile System

For our third case study, we're going to talk about the Patriot Missile System. The Patriot Missile System is a highly mobile surface-to-air missile system originally designed to shoot down enemy aircraft.

It became a household term in 1991 at the onset of the Persian Gulf War where the system had been adapted, modified to shoot down Scud missiles. And apparently there was at least some kind of qualified success. Unfortunately, the Patriot successes, whatever they were, are overshadowed by one particularly deadly failure.

Slide 25: Patriot Missile Failure

In February of 1991, an enemy missile struck American troop barracks in Saudi Arabia when a battery of Patriot missiles failed to intercept the target the incoming Scud missile. In fact, the Patriot missile never launched. The result was 28 dead soldiers and over 100 other casualties. The root cause was determined to be a software error in the system's clock. Specifically, an accumulated clock drift that worsened the longer the system had been an operation.

The problem had been identified approximately two weeks before the incident when the Israeli army reported that they've noticed that the longer the system was online the less accurate it became, but no patch was available at the time of the incident. The recommended work around was to reboot the system. At the time of the incident at Dhahran, the patriot missile system has been operational for approximately 100 hours. When you work through the math of the software error, this ends up resulting in the clock drift of approximately 1/3 of the second and this translates into a tracking error of approximately 600 meters. One interesting side note: during the Gulf war the patriot missile systems had six firmware updates and each of these updates that was supplied require the missile system to be offline for 1 to 2 hours.

Slide 26: The Patriot Software Bug

So let's talk a little bit about the details of the software defect that resulted in this incident. The Patriot missile system software had two versions of system time. So, the clock tick on the system was 1/10 of the second, a hundred million seconds. So there was one clock time which was an integer number of ticks, each tick represented 1/10 of a second. There's also a decimal representation of the system time used elsewhere. Problem is and some of you know there's no way to represent a value of 0.1 in binary.

So for example of value of 0.5 or 0.25 so negative exponents a power of 2 we're talking binary here can be represented exactly. Something like 0.1 cannot be represented exactly, it's what’s called a nonterminating sequence. The problem there is the conversion from integer ticks to these decimal values results in rounding error, due the imprecision and after approximately 100 hours, this results in an about of 1/3 of a second drift. Radar works by accurately measuring the timing between pulses and the reflections. For this to work well, everybody has to be working off the same time base.

The patriot missile system software had been modified to calculate floating point time more accurately than having previously when it was updated to track these very fast Scud missiles. But, not all parts of

the software were updated so the result was the timing measurements were made with their very accurate clock, those timing measurements were being compared against timing measurements made with the less accurate clock which was susceptible to this rounding error. So, when these two compared against each other, there would be a discrepancy.

Slide 27: Perils of Floating Point, 1

So, we've got a couple slides that illustrate some of the hazards that you can run into when you're using the floating point. So, I'm sure most of you can understand this program here, we're going to call test one and test two. We assign a value 0.1 to its standard 32 bit single precision floating point value and then we print it. And sure enough when we print it, we see value of .1 and .2 up out to six decimal places.

All looks as expected right, so far so good. Now let's say we just change the print out to print with higher precision, so show us more than what the standard print out shows and look what happens. It turns out that when you assign 0.1 into the variable, it's actually stored as a bunch of zeroes and then there's that little one at the end. And let's say you now add another 0.1 to it, look how the error accumulates. Now the 0.2, it turns out it's really 0.2 a bunch of zeroes and then a three at the end. Well imagine if this is happening every a tenth of a second for 100 hours, 360,000 times how that error accumulates.

Slide 28: Perils of Floating Point, 2

I'd like to illustrate one other problem by using floating point, again it comes down to rounding but I'm going to show you in a different way. So test3() here, we have two different variables f1 and f2, both single precision floating point values. If you look at this slide while I'm talking, you'll see that essentially what we're doing is we're calculating the sum of 0.1, a 0.3, a 0.7 and we do it two different ways in f1 and f2.

With f1 we initialize it to 0.1 then we add 0.3 and 0.7, we expect the value to be exactly 1.1. With f2 we initialize it to 0.3 but then to get up to 1.1 we actually have a loop and we add 0.1 to it eight times. And then what we do is we print the two values at the end. And you'd expect them to be the same but actually when we run the program and we print the two values – look at this. By now it's probably not surprising that f1 doesn't print to exactly 1.1 in fact you can see the error here. It's something to be aware but you were aware of that last time.

But look at this: f2, even though we essentially expect it to get to the same value, it has a different error. And that's because each time we add a value that cannot be expressed exactly in binary, there's a rounding error. So the more operations you perform to get to the same point, you're going to have a greater error. So the point of these last two slides is simply to make you aware of the fact that using floating point math in computers can be very tricky and you need to be very aware of the potential problems that you can run into.

Slide 29: Accumulated Error

So getting back to the rounding error due to floating point and the time difference between the clocks. The end result of this discrepancy was that a scud missile, whose launch was detected by early warning satellites, was not tracked correctly by the Patriot's ground radar. And the system determined that there

was no missile threat. Unfortunately the radar was just looking in the wrong part of the sky due to the clock drift problem.

So how did this bug which seems so obvious make it into the field? Didn't anybody do any kind of testing? Well as Michael mentioned in his keynote in April at EE Live, you can almost imagine the kind of testing and the test procedures that were performed and I would imagine, as Michael mentioned, that each test started by having a newly powered on or rebooted system for a clean slate for a fresh set initial conditions.

So this whole concept of running a system for a hundred hours and then seeing how accurate it is probably was in no kind of test plan. Although I certainly hope that such a procedure would be in place today. Things like this, gradual resource leaks, long-term timing drifts, et cetera, may only be found by testing systems in exactly the manner and environment that they'll be used. So that's something very important to keep in mind, especially for critical systems.

Slide 30: Patriot Missile Failure: Lessons

So what are some of the lessons learned from the Patriot Missile failure? Well, obviously mixing data types such as floating point, fixed point, but even mixing signed and unsigned integers as we saw in the previous webinar on our coding standard, this can be very tricky. In fact hopefully you just saw that even using floating point alone by itself could be tricky due to rounding errors and things like that. When you're tracking any kind of quantity, it's very important that you're consistent in your units and your data types and that you understand issues of precision, rounding, data type conversion, et cetera.

And again at the risk of beating a dead horse, please keep in mind that testing will not catch all problems. Either because the tests are inadequate or there are some defects, some kinds of bugs that are so rare and so unusual that they will slip through testing. And a side note to that, as I mentioned before: the test environment it's very important that you replicate the intended environment as closely as possible when you're doing your testing.

Slide 31: Mars Climate Orbiter

Our final case study today is the Mars Climate Orbiter, one thing I want to point out from the outset is that the problem that I am going to describe was not a bug or defect in the embedded computer on the orbiter. I want to mention that right out front. That said, it does illustrate a software defect problem that I want to cover it because it's very important and the consequences were very sad. So, the Mars Climate Orbiter was a small robotic space probe, about 750 lbs so probably about 300 Kg, designed to study the climate and atmosphere of Mars.

The probe was launched in December 1998 and began its journey to the red planet. Less than a year later, it had essentially reached Mars. Unfortunately, on September 23, 1999, the orbiter disintegrated as it passed too close to Mars' upper atmosphere on the wrong trajectory because incorrect information from ground-based computers had been passed to it. The root cause was that the thrust impulse command for the thrusters was produced in imperial units, specifically pounds-seconds, instead of metric units, newton-seconds as specified. It would be similar to ordering one liter of water and receiving 1 gallon of water. The number is correct but the units are wrong and that makes all the difference in the world.

Slide 32: Units are Important

So at the risk of stating the obvious, computers are used to calculate things but most of these calculations are performed on quantities that have units. So for example if you're measuring pressure, whatever units you're using maybe it's kilopascals or whatever, if you're measuring velocity perhaps it's in meters per second or miles per hour. If you're measuring flow it might be liters per minute or milliliters per second or whatever, these all have units. These all have dimensions.

And there are essentially two kinds of mistakes that are commonly made. The first kind of mistake is that the same fundamental dimension is used but a different system, imperial or metric, is used and they're mixed together. So here we have an example in API that you're supposed to call it with the value in meters per second but the bug is that someone's passing it a value of velocity in miles per hour. So they're both velocities but they're the wrong measurement systems. The other kind of problem, which unfortunately is more common, is a disagreement in the fundamental dimensions.

So look at this here we have set acceleration and most of you probably know acceleration is the first derivative of velocity with respect to time. But here what we're passing it is essentially a velocity. We calculate the difference between two positions and divide it by the amount of time it took to move from position one to position two. So what we're really passing to it is a velocity not an acceleration. The units don't even match up. So just think about the products that you work on, your own products. If you're working on medical devices, maybe you're measuring flow or pressure or a voltage change over time. If you're working in transportation, velocity, acceleration, temperature, et cetera. So this is everywhere no matter what product you're working on.

Slide 33: Dimensional Analysis

So now I'm going to introduce a term to you, some of you might have heard of this before, some of you this might be a new term, it's called dimensional analysis. Now in C and many other programming language, the standard types don't have any concept of units. So here is an example, int speed = 1234, well what does that 1234 mean? It means whatever the programmer intends it to mean but it's not obvious or intuitive just from reading the code what the units are if there's no implicit decimal point in there et cetera.

So what we want to do is see if we can find a way to use the language's type system to help us prevent making these kinds of mistake. And it turns out that we can. I'm going to show you an example in C and an example in C++ and I'm also going to show you how you can use static analysis tools to catch this kind of problem. Because these are the subtle kinds of problems that are very difficult to debug.

Slide 34: Using Flexlint 9 to Expose Dimension / Unit Problems

So here's a very simple example of using meaningful typesets and a static analysis tool's ability to warn about incorrect mathematical operations. For this demonstration I'm using a tool that we use quite a bit called Flexelint, it's Flexelent 9 the newest version. The way this works is that we define unique types for distance in meters in meters, distance in time, sec and velocity. The real magic happens on line five, which is a C comment but its special format indicates special options to the static analysis tool in this case Flexelint. Line five tells the tool to perform strong type checking for the types shown in parenthesis.

It also indicates the relationship between the types, here showing the relationship between velocity, distance and time. This goes a long way towards addressing C's very weak type checking. Remember that even though we have a typedef for velocity, as far as the C programming language and the compiler are concerned; they're all just the same types double. And it will let us assign any type of double to a velocity even if that double reflects stock price or the constant pi or any other value. Because a double is just a double to the compiler. Now notice that this is very simple example. You can define as many types as you need and define every appropriate relationship between the types and the static analysis tool will catch that for you.

So if you look at lines 10 through 13 in the code and you look at the expressions on the right hand side of the assignment, you'll notice that only line 11 doesn't evaluate to a distance over time when you cancel out all the units. And you notice that in the output from the static analysis tool it warns us that on line 11 we're making an invalid assignment. And that's all because of the type checking that was enabled on line 5 in the comment.

Slide 35: C – Don't Use Naked Numbers

Now in addition to using a static analysis tool, which I strongly recommend, there are things you could do with even the C programming language to try to at least protect yourself a little bit. And that is to use a somewhat object oriented approach by defining different types or classes with different units.

So even if you're using something like an integer for speed, whether you're using centimeters per second or miles per hour, by wrapping that in a structure or a class, the types are not compatible. You can't do direct assignment and if you try to pass a pointer to one type through a function which is expecting a pointer to another type you'll get a warning, certainly if you're using a static analysis tool, it'll tell you that. So what happens is you wrap these values in classes that you create for each different units that you want to use and then you just pass around the pointers or handles to these different types.

So here you see two different classes defined for speed in miles per hour and speed in centimeters per second. And then we have functions, the first two being constructive functions, one to construct speed in centimeters per second object from its natural type and another one to actually convert a speed object that's in miles per hour into a speed object that's in centimeters per second. Presumably we would have all sorts of balance checking and things like that inside these constructive functions.

And then the last thing is we have this API adjust speed centimeters per second where if we want to increment or adjust the speed positive or negative, we pass it to speed object that we want to adjust and then the actual adjustment. And notice that the adjustment which we don't want to change is a pointer to a constant because that shouldn't be changed. The first parameter current is the object that's going to be changed. And one other thing I'd like to point out is that these encapsulated types won't take up any more storage than the native types.

Slide 36: Even Better – Use C++

The last point I want to make regarding dimensional analysis, if you are using C++ you have all the tools you need to do this in an ideal way where you can catch problems at compile time. C++ with its stronger type system, you can exploit templates and native programming to enforce dimensional correctness in

your calculations at compile time. Unfortunately, I could spend an hour talking about this, people much smarter than me have given long talks on this, it's a very important topic.

If you want more information on this, if you are using C++ I would point you to a couple of resources. The first – I'm giving you the links at the bottom of this slide. The first is a paper from Scott Meyers called Dimensional Analysis in C++ which goes through it in very good detail. Also there's a boost library called Boost Units, which is used for exactly this purpose. So if you're using C++ please check out these resources.

Slide 37: Filtering Out the Defects

So now that we've looked at some of these case studies and we've seen the defects and perhaps how they would've been caught, let's turn this into a picture. So the picture we want you to have in mind is this picture here with a series of cascaded screens or filters or sieves. And each one of these represents a step in the process. A tactic that you can use to reduce the defect. So each step in this process extracts defects, removes them so by the time you get to the end, only a very few defects have escaped.

Yes it's true that we're showing a few defects escaping at the end because honestly that's probably the most realistic picture. The point is that there's no one single thing you could do to make your system, your critical system safe and secure. Each of these sieves is going to catch things that the others won't but you're never going to get to perfection. And that's why I emphasize so strongly the importance of keeping in these dynamic run time checks in your code. Certainly there's overlap in these different phases, in other words there are things that you might catch in static analysis that you can also catch in code inspection but the point is each of these has strengths that will catch things that the other phases won't.

Slide 38: Key Takeaways

So what are the key takeaways here, obviously there is no such thing as bug free software. There is very, very high quality software but there's no such thing really as bug free software. And one of the implications therefore is that testing is not sufficient to catch all bugs. What you really want to do is employ a strategy of defense and depth. So you want to employ everything at your disposal to get your software as robust as it possibly can be. For example, employ a coding standard whether it's the MISRA coding guidelines or the Barr Group Coding standard or a combination that defines a safe subset of the C language to keep you from getting into those dusty corners of the language that you don't really understand. Also having a very robust process including static analysis, code inspections, et cetera., is very, very important.

And obviously it's always better to prevent something in the first place than to have it escape and to fix it. And knowledge, education is one of the most important ways you could do that. So if that's something of interest to you, for example taking a deeper dive into some of these topics we've covered here as well as covering some new topics we haven't had time for, consider taking our one day class on developing Safety Critical Firmware on September 23 in Detroit.

Slide 39: Further Reading

And then lastly in closing, I want to leave you with a few references for further reading. The first item, it's an editorial from 2010 in the L. A. Times and it talks about the fleeting nature of difficult to

reproduce software problems. And in particular it talks about the importance of using assertions or sanity checks and how one day a team at JPL in Pasadena, California saw an assertion catch something that should have been impossible. Ultimately the triggered assertion led to uncovering a defect that would've been almost impossible to reproduce otherwise.

The second resource is both an article and a video by Gerard Holzmann. He's a senior research scientist at NASA's Jet Propulsion Laboratory. And the article and the video describes how the software for the Curiosity Rover was created.

And lastly, the third resource is Better Embedded System Software. It's a blog by Phil Koopman, talks in depth about all sorts of aspects of developing firmware for critical systems. Very good reading, I highly recommend it.

So that's it, thank you very much for joining us today. We look forward to seeing you next time or maybe even two weeks from now at our one-day course in Detroit. I'm now going to turn the presentation back over to Jennifer.

