25 Essentials to Robust Embedded System Design

In celebrating with Circuit Cellar I thought it fitting to share some of the insights I have gained over these past forty years of designing embedded systems. Circuit Cellar is a fitting place for me to share this since it was in Circuit Cellar that we advertised the first embedded system that our company designed, marketed and sold. The product (now no longer available) was called POC-IT (Power-On Cycler & Intermittent Tester) could be used to verify the robustness of your embedded system over 1000's of power outages. It was designed specifically to help ferret out power-up issues in your designs and was used by scores of companies like Dell and Apple in verifying some of their designs.

The reason why POC-IT is so fitting to start the discussion about designing robust embedded systems is not because it was first advertised in Circuit Cellar, but because it exposes a great weakness we have as designers. You see, the product had not been out for more than a month when we started getting reports from the field that POC-IT had power-up issues. It sometimes failed to start following a power cycle. You see we had failed to test POC-IT on POC-IT. I was stunned that we could be so blind. We had pitched about how important it is to test your designs to verify that it can come up correctly 1000's of times. We knew that this was an area that was seldom tested but that often was a source of both hardware and software problems. He had uncovered power up issues in FPGA's and microprocessors but we failed to turn it on itself.

How can that happen? Hubris is certainly one reason. We know others can miss important power up details but not us. But I think there is a more fundamental problem. I think that the systems we design are so complex that we need systematic methods that we can impose upon our designs the help us to remember all the little details. Little check lists – Did I do that? Did I check this? I wish I could say that I never repeat a failure. But I do. Didn't I learn from it? Yes, I learned not to do it again. But I do.

This past week I spent the better part of a week debugging 5 lines of C code. I knew the problem was in those 5 lines. I had ample evidence of how it failed. But I couldn't – nor could anyone else in the company (and we have some very smart people) see what was wrong with those five lines. Only after I was able to create scenarios where I could eliminate one line at a time until I finally got to that one line of code that was failing was I able to see the problem with that one line of code. Why couldn't I see it earlier? It is easy to beat myself up and say – I've got to get smarter. But I found some comfort in my colleagues who also couldn't see it. It came down to the fact that the code violated one of those simple principles about head and tail pointers that I have learned and I have taught. But I missed it.

The second thing stuck in my mind this week is how a subsystem in one of our new designs is failing and has the potential of sinking a very good start-up who has thousands of these in the field. I remember when this vendor was selected for inclusion in the product. The three key architects all said that they were not comfortable with this vendor. There was just something about his way of doing business they didn't like. But he was the only game in town and we didn't have time to design our own. So we all made the decision to go with him. We all know that old adage: If you don't have time to do it right the first time, where do you find the time to do it right the second? But it is the nature of developing robust embedded systems. There are literally thousands of little decisions that we make even in the simplest of projects. How can we minimize repeating mistakes?

So my goal in this article is two fold: To celebrate with Circuit Cellar in 25 years of great service to us engineers and to hammer home some of those principles that many of us learned but that we so often forget. I will divide the essentials into four categories: general rules; rules that exist because things (us and our designs) fail ; rules about testing; and rules about memory use.

When I first presented the question to my team: "What are the essentials of creating a robust embedded system?" the first response was don't use software or electronics. Most of these folk don't remember the old Friden calculators that did math (including square root) without software or electronic hardware. We actually used these where I first worked to perform inertial guidance calculations for the Lunar Excursion Module backup guidance system. Those systems were complex and far from robust. Or perhaps they don't remember how the old I.S. glass machine worked without electronics or software – systematically taking off fingers and toes. I'll go with the software and hardware any day.

But only if we do them right.

General Essentials

Let's start with five general essentials that may seem obvious but which I forget all the time. Although they seem simple, when I avoid them I get in trouble every time.

#1 & 2 The KISS essential

Keep It Simple Stupid but no Simpler. How often do I need to hear this? I like the saying often attributed to Albert Einstein (but was actually Roger Session's paraphrase) about KISS: "Make things *as simple as possible*, but *no simpler*." I am counting these as our first and second essentials. Keep it simple is number one and no simpler is the second. I find this an immense challenge. When we are faced with schedule deadlines and tight budgets it is costly to make a design simple. Some people have a gift at taking a problem and abstracting an elegant and simple solution. I remember giving a design specification to one of my employees a number of years ago when I worked for an aerospace company. After several days he came back with over 20

pages of algorithms and charts defining how the specification should be met in software. I wasn't smart enough to know why it was too complex, but my gut feel said: This is too complex – make it simpler. After another day I turned it over to another young man who returned with an elegant two page algorithm that seem to (at the time and later proved to be) just right.

How do we do that? As simple as possible can get us in trouble if we make it too simple. For example, just recently we were designing a multi-drop serial interface to be incorporated into a medical device. A strong case could be made for simplicity of using a single ended interface. But experience tells us that a differential interface will be more robust in the face of de-fibrillaters and all the various noisy electronic instruments it was going to play with. Which meets the KISS principle? The same tough decision comes when you decide whether you go with a 2-wire or a 4-wire interface. 2 wires has less cabling but is more complex in the interface and forces single duplex operation. Again which meets the principle?

Sometimes the trade-off can come down to what you already have in the house. If you have well debugged libraries for handling the 2 wire 485 protocols, the reduced number of wires reduces the overall system complexity even though the software will in fact be more complex.

Sometimes when dealing with algorithm development, the KISS principle can also present ambiguous alternatives. Sometimes, a straightforward linear programming approach can be many times more lines of code and more complex than an elegant algorithm. But the elegant algorithm may be obscure, difficult to change and take too long to come up with. Therein lies the challenge to Keep It Simple Stupid but no simpler.

#3 Define the Problem / Create Clear Spec's

Having a clear set of spec's is essential to every part of a design. We all know this and we always belly ache about how we don't have perfect specifications. But get with it. You won't always have perfect spec's from your customer. But it is essential that you make them as good as possible. And in writing. If your customer is willing, keep pushing back and keep writing it down and refining it as you go.

I have found that essential for every phase of a project – whether it is hardware or software, writing out the spec (on the schematic or in the code) is a wonderful discipline. Write out the spec for the module. Write out the spec for the algorithm. Write out the spec for the circuit. Writing it out forces you to think it through.

End the belly-aching about the lack of good spec's and start creating them.

#4 Don't Skimp on the Tools

Tools are our life blood. If you are a manager and your designers don't have the best tools, you are wasting your money on their salaries. That said, we are not talking about buying tools you don't use or tools that don't pay for themselves or tools that you can rent more cost effectively. Last week we were discussing a problem where one of our cell modem designs exceeded the limit for the 2nd harmonic in spurious emissions. In talking the problem over with the test lab, I discovered that they had a tool that they brought inside the anechoic chamber that was able to tell the cell modem to transmit on such and such a frequency at maximum power. Naively I asked, shouldn't we have such a tool? He responded with a "Yes – but they cost almost a million dollars." Oh. But we found we could rent one for $1000 a day. So I am not talking about being unwise with our money – but being wise with our money.

Many years ago while at the aerospace company I was recommending an HP64000 system that appeared to be a very powerful tool for our software development team. I wrote up the proposal to the VP of Engineering and his question has always haunted me since I formed my own company. "Would you buy it if it way your money?" Well I said then and would continue to say now – Get the best tools that will allow you to do the job as quickly as possible. If a 200 man hour job can be done for 190 hours with a $10,000 instrument, is it worth it. Absolutely.

#5 Read the Documentation

Last year we had a problem that showed up only after we started making the product in 1000 piece runs. The problem was that some builds of the system took a very long time to power up. We had built about 10 prototypes and the design was tested over 1000's of power ups and tested just fine (thanks to POC-IT). Then the 1000 piece run uncovered about a half dozen units that had variable power up times – ranging from a few seconds to more than an hour! Replacing the watchdog chip which controlled the reset line to an ARM9 processor fixed the problem. But why did these half dozen fail. Many hours into the analysis we discovered on the failed units the RESET line out of the chip would pulse but stayed low for these long periods of time. A shot of cold air instantly caused the chip to release the RESET. Was it a faulty chip lot? Nope. Upon a closer read of the documentation it said that you cannot have a pullup resister on the RESET line. For years we had pull ups on RESET lines. We missed this in the documentation.

Like it or not, we have to pour over the documentation of the chips and library calls we use. We have to digest it carefully. We cannot rely on what is intuitive.

Finally, and this is much more necessary than in years past, we have to pour over the errata sheets. And we need to do it before you commit the design. A number of years ago, a customer designed a major new product line around an ATMEL ARM9. This ARM9 had the capability of directly addressing NOR memory up to 128 meg. Except the errata that said that due to a bug it could only address 16 meg. Ouch. Later we had problems with the I^2C bus in the same chip. At times, the bus would lock up and nothing except a power cycle would unlock it. Enter the errata. Under some unmentioned conditions the $I^2$2C state machine can lock up. Ouch. In this case we were able to use a bit bang algorithm rather than the built in I^2C – but obviously at a cost of money, schedule and real time.

#6 Remember that "Problems Happen"

We live in a broken world. Things like "Murphy's Law" and entropy tell us that in spite of the wonder of the creation (like how the eye works) there are forces at work that cause bad things to happen. Things don't always work correctly. It is amazing to me how many things actually work. On our web site, we list some of our guiding principles and the way we phrase it is "Imperfection Reigns." When a friend saw it one day he said – "Won't that turn off potential customers? Shouldn't you be telling him that you do things right? Imperfection reigns? How about 'There might be a few problems.'" I told him that this principle guides a lot of how we design our systems.

As I was writing this section, I just finished a 4 hour project of staining my deck with a stain that said: "Don't apply if there is any chance of rain within 24 hours." We haven't gotten much rain this year but I was looking for a day with 0% chance of rain all day. Today was it. When I finished staining and sat down to write, I rechecked the weather. Fist pump. Yes! Still 0% chance of rain. One hour later I heard the pounding of rain on my driveway. No! Problems happen.

If we could engrain that truth into every design and every interface our designs would be much more robust. Here are 12 essentials in embedded design that exist because "Problems happen."

#7 Logging

Many of our designs are "headless" and have no user interface. Many of our designs are buried in some product or located in remote areas. No one was there when the device had a hiccough. Well defined logging can help make your system more robust because there are going to be problems and you might as well find out as much as you can about the problems when they happen. Here are some general guidelines that we apply here:

1. Use an existing logging facility if you can. It should have all of the following features.
2. Unless you truly have unlimited memory, provide a self pruning cap on all logs. Linux syslog feature has the built in but if you don't have any limiting, put it in.
3. Attempt to provide the most amount of information in the least amount of space. One way we do this is by limiting the number of times the same error can be logged. I cannot tell you how many times I look at a log file and find the same error logged over and over again. Knowing your memory limitation and the frequency of the error, after a set number of identical logs, start logging only every 100 of them or only allow some many per hour but include the count. I haven't found a good and flexible set of algorithms for this and it seems like we have to invent it every time (please email me if you know such a library) but

it essential. Some failures are best kept in error counters. For example communications errors in a noisy environment should be periodically logged with a counter – you don't usually need to know every occurrence.
4. Create multiple logs concerning multiple areas. For example, network errors and communications errors are better kept in their own log apart from processing errors. This will help a single error from flooding all logs with its own problem and thus locking out all others.
5. Timestamp all logs – ideally with date and time – but I understand that all of our systems don't have date and time. As a minimum it could be in milliseconds since power up.
6. Establish levels of logging. Some logging is only applicable during debugging. Build that into your logging.
7. Avoid side effects – I hate it when the designer tells me that if he turns logging on the system will come to its knees. That's just a bad logging design.
8. Make the logs easy to automatically parse.

#8 Use Watchdog Timers

No longer used in just the realm of fault tolerant software, independent watchdog timers are put on systems because we know something can go wrong and prevent it from being fully functional. Sometimes the dogs reset the processor and sometimes they just safe the device and notify a user. However they work they are an essential part of any system design. Here are the major guidelines we use:

1. Independent of the processor – The last thing you want is for the processor core to lock up and the watchdog to also lock up.
2. Do not tickle the watchdog during an interrupt (the interrupt can keep going while a critical thread is locked up)
3. Do not tickle the watchdog until you are sure that all of your threads are running and stable
4. Provide a way for your debugger to have break points without tripping the watchdog
5. If the watchdog can be disabled, provide a mechanism for this to be detected

I provide many more guidelines for watchdog design in a white paper on our web site: http://microtoolsinc.com/RobustGuidelines7.pdf

#9 Allow for Recovery from Errors

We are currently working on a system that the customer has specified that upon the first serious fault, the unit shut down and have to be powered down and then back up. This is in a medical product that will cause great discomfort in the patient when it stops working. Hopefully we can get this changed before it goes to production. Many systems can recover from serious faults and thus be made more robust. I remember my first computer controlled car (1981 Dodge Aries). I was driving back from a vacation when the Engine Control Computer failed. We were on the highway in heavy traffic. The car sputtered for a few seconds and backfired (not a good thing but

remember this was among the early designs) and then resumed normal operation. I pumped my fist. Way to go Chrysler. Over the next 180,000 miles, this never happened again.

We need to design our systems so that when things fail it can keep going – if even with minimal operation. In many cases, minimal operation is enough. There is much to be said about this and perhaps can be an up coming article.

#10 Allow for a Field or Remote Software Update if at all possible

Of course, your software will never need to be updated after it passes your verification testing and is released to production. But just in case, your system will be more robust if this is a possibility. So many of the devices we design now days are connected to the internet. It irks me when I detect a software glitch in my car or phone and am told that the only way to fix it is to replace the module. Ouch. And don't forget your FPGA's and PLD's. Many of these can be reprogrammed in the field.

#11 Never trust anyone over 30 mm from your software

In other words, don't trust anything that comes to you externally. Expect the unexpected. Not just users who have an uncanny way of breaking our software. But for any external device that has software in it. Yes you have an API with the device, but it can have bugs. If the function can make or break you operationally, you should verify that all operations were successful. I detest protocols that provide no feedback as to whether or not the data was accepted. One of our designs interfaces with a solar panel inverter over RS-485. I can read a power or energy reading and have no confidence that the data actually represents what was sent. Our design must assume that the data cannot be trusted.

#12 Check every possible error

Many years ago, we sold a disk deftragmenter. The first products from Microsoft and PC Tools took several hours to defragment a 20 megabyte hard drive. We came up with an algorithm to reduc that to minutes and yet was perfectly safe across power downs. A company was interested in incorporating our code into their product which was a disk compression driver. After we licensed the source code, the came back to us and told us that we were not checking every function call for errors. So we went through all of code and add code to always check the return code on every function for errors. Their product went on to be a great success (it was bought by Microsoft) but we learned our lesson. Always check for return codes and check for errors in all of your communications with other devices.

#13 Many people make software work

A firm principle at our company is to involve as many as feasible in the review of designs, architectures, code and test plans and when trouble shooting problems. Realistically as many as feasible is only one or two, but we believe strongly in it. I never cease to be amazed at my own and other's blind spots when it comes to designing complex embedded systems. Even with all that we do to put together many heads, I am shocked at how often we submit a design to our non-technical

customers and they come back with howlers. Why didn't we think of that? Because we are all limited.

Here are a couple of things we do.

1. Put together a tiger team when attacking a problem or reviewing a design. Try to match the skill and gift to the review process. I find that some people are good at finding/solving some kind of problems and whereas other very good designers are almost worthless at this.
2. Provide some kind of incentive for a successful solution. Break out a bottle of sparkling. Celebrate the team effort – even if one person solved the problem.

There is much more I could say but that will cover it for now.

#14 Allow for spare I/O, spare memory and spare real-time

Early on in my career with an aerospace company, a regular part of our spec's from our customers included the requirement for 50% spare memory (RAM, EEPROM, FLASH drive) and 50% spare real time. I had never thought of that before but that principle has helped me create robust systems for many years. Spare I/O has also covered my behind when those unexpected problems come up. Even with all that good experience just last week our customer added enough I/O to force us to another PIC chip that costs $.30 cents more than the other chip. Board area and system cost were not issues. Why didn't we do that the first time. Because we think we know better.

#15 Incorporate Redundancy

A very important principle that we find essential to robust embedded design in the proper placement of redundancy wherever a failure would cause more than just loss of functionality. Safety, loss of valuable data, and reliability can all be drive the need for redundancy. For example, in a postage scale that we designed, all of the critical postage data was kept redundantly in two different technologies of memory. In case a medical product, we designed hardware redundancy into the safety loop so as to take software out of the safety verification loop. But where cost allows and reliability is key, we can add significant reliability by identifying the least reliable parts and double up parts. An example of a low reliability part is a serial EEPROM. Even with 1,000,000 erase/write cycles, logging and other demands can quickly chew those up. Rather than reduce the lifetime of the product, just add a redundant chip.

#16 Testing, Testing and More Testing

We have a motto at our company: "If it's not tested, it doesn't work." A key to creating robust embedded system designs requires all kinds of testing. We need to be creative at how test our systems. Here are some testing principles that help us create robust systems.

#17 Test to Break / Not just to Verify Functionality

Very often we create a test plan that covers all of the operational requirements from the Software Requirements Specification. If that is all we do, we often find that there are many problems that

never get uncovered. In 1995 we hired a brilliant high school student to do some testing for us. He loved to break things. He also loved to explore and find soft underbellies of our designs. Very often with a sly smile and a twinkle in his eye he would ask: “Did you really intend the product to light on fire when you used this option?” He taught us that we needed to train all of our testers to be somewhat devious and think outside the box and think about how to break the design.

#18 Stress Testing

Another principle we have learned in testing to create robust systems was that you needed to stress the unit. Our years in the aerospace industry taught us about “shake and bake” (heating and vibrating our designs) as one means of stress testing. But how about stressing the software? This requires creativity. Let me list two things just to get you thinking about stress testing. On a network device, we often pound the unit with “denial of service” kinds of input and see how the unit responds. With serial interfaces we like to place it in an extremely noisy or simulate an extremely noisy environment. The bottom line: Be creative at putting on your designs all the stress you can think of.

#19 Off design testing

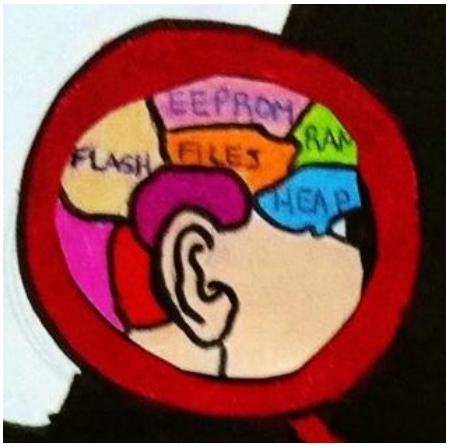
This is another trick that we have learned over time. If you want to flush out problems in your design push it to off design limits. Does the system normally connect to host every 30 seconds? Try connecting every 5 seconds and observe what happens. Does the cell modem handle text messages every 5 minutes? Send them every 5 seconds. Some things you find may be obvious. But you may find some holes that you never expected.

#20 Power Cycling

Oh – did we talk earlier about cycling power 1000’s of times. This is a must. I am embarrassed to say how many problems we have uncovered when we cycle power this many times. Set it up so that you have a way to detect that the system is successfully up. And then let it go.

Memory Management

We have already talked about having adequate spare memory but there are two other things that are essential to robust designs: Leak detection and Out of Memory planning



#21 Memory Leak Detection

The first time I created an object oriented user interface I quickly discovered that I stranded a lot of dynamic memory allocations. Given enough time, my software would have eaten all of the memory available. That’s when I learned about memory leak detection software. It starts with design. You have to hammer into your designers that even a small memory leak can kill an embedded system. You have be choose libraries with care. We are currently using an XML parser that makes it very difficult not to create memory leaks. And create them we do. Every system you design that uses dynamic memory of any kind needs to be tested over the full operational range to verify that no memory leaks are present.

#22 Out of Memory / Disk space Planning

Two of the most neglected segments of designing robust embedded systems is planning for how you deal with out of memory and out of disk space. From my experience in using a variety of embedded devices it rare to find a system that handles these conditions gracefully. And that is why it needs to be handled in the beginning of the design. Plan for it. Your system will be more robust for it.

User Interface

I have an MP3 player from Sansa that has the most maddening user interface. You never know where you are in the menus and nothing is intuitive. A key to a robust embedded design is an intuitive user interface. Now days even headless devices have browser interfaces so it is rare for an embedded device that we design not to have a user interface. There are two key principles that if followed will give your design a reputation for being a robust design: Assume that no one reads the user manual; Give your users clear instructions as to where they are at all times and give them a mechanism to get back to the main home screen.

#23 If you can't explain it to Mom, it ain't clear

That's another way to say: "Assume no one reads the user manual." Last night I was reading about the Boston electronic parking meters (http://usabilitylessons.wordpress.com/category/general/). Truly one wonders who reviewed that user interface. If you want to make robust embedded systems with a user interface, they need to have intuitive interfaces or you may be surprised at what the user comes up with. This takes time and effort but is well worth it. Try it out on the uninitiated. Engineers are the worst kind of people to test user interfaces. Try your kids or grand kids. Our first bug on our first product was found by the one year old son of my business partner. There are many good resources concerning how to create intuitive user interfaces but I leave that for you to research.

#24 Provide a way for your users to know where they are in the user interface

Final Shots

#25 Remember, remember, remember

Last but not least in our list of 25 essential principles of robust embedded design is the discipline of remembering these (and other good design principles) every design. This should be easy but for me it's not. Perhaps as I get older and gain all of this wisdom I tend to forget one of more of these as a new design progresses. However you do it – be it a check list or some kind of a review process, asking yourself the kind of questions presented in this article will help you create more robust designs.

You can find more excellent white papers that we have written at www.microtoolsinc.com.