

Guidelines for Creating Robust Embedded Systems

Part 1 - Introduction

By Bob Japenga

Scope

This white paper is intended to provide a brief overview of lessons learned in creating robust systems over the past 35 years of embedded systems development (starting with the Intel 4040). It is divided into a number of parts – each addressing a different issue. This part introduces the topic. Each subsequent part will address each one in depth.

Other white papers concentrate on different issues like coding standards¹ or general software architectural issues² in order to create robust embedded systems. These are of great value and an important part of creating robust systems. This series of papers will concentrate on specific lessons we have learned the hard way by creating non-robust embedded systems when a bullet proof system was required. And we write them down because “wisdom leaks.” We need to review these periodically because these pearls of wisdom often get forgotten in the heat of the development cycle.

Summary

We all know how to create non-robust systems (embedded or otherwise). Not paying attention to details, working with fuzzy specifications, the unreliability of the hardware (isn't it always their problem?!), making changes at the last minute, and when the complexity exceeds the experience base of the team are just a few of the ways we can create these systems.

But over the years there have been a few things that we have learned that have helped us create systems that run 24/7 fifty-two weeks a year. Here is the list (in no particular order) that we will attempt to flush out in this paper. Each part of this white paper will address one of these guidelines:

1. Design for and test that there are no memory leaks.
2. Design systems where out of bounds memory references are caught and allow the system to recover.
3. Limit the availability of I/O to be accessed
4. Gracefully handle of out of memory conditions
5. Gracefully handle of out of disk conditions
6. Providing a watchdog on all of the critical tasks to verify that they are all working properly and taking appropriate action. Watchdogs serviced by a simple interrupt handler are not of much value because a critical task or thread could have crashed and the watchdog will continue to be serviced.
7. Design for and verify data integrity – Don't assume that the data written is always going to be the data read
8. Design in data redundancy in critical areas

9. Provide liberal error logging that includes pruning of logged data
10. Design systems that work even after failures have occurred. For example, if an error forces tripping the watchdog, that same error should not trip the watchdog 50 times in 50 minutes.
11. Design systems that know when to throw in the towel and not go into endless loops (“You got to know when to hold em, know when to fold em”)
12. Don’t assume that your system will power up every time – test it through 1000’s of power cycles
13. Design your system so that it can be powered down at every instruction and not become non-functional (i.e. it may lose some data but not become non-recoverable)
14. Avoid pre-emptive scheduling if at all possible
15. Avoid sharing variables across threads or tasks. If you must do this, make sure that the variables are read to or written in a cycle assembly language instruction.
16. Test your system at beyond the boundaries of normal operation (Stress testing).
17. Design your system with built in spare time, spare memory and spare disk space. Measure these “spares” under carefully planned stress testing.
18. Design your system so that typically unattainable boundaries can still be tested.
19. Be fully aware of your stack utilization requirements and measure stack utilization during carefully planned stress testing.
20. When writing driver code, pay careful attention to volatile hardware registers. With memory mapped I/O and programming in C, use the “volatile” prefix for all hardware registers that can be read.
21. Design systems for which you can get all of the source code and the means to build them
22. Provide automatically generated version generation to allow for field verification of the version
23. Create a design log of all areas of the design that will be difficult if not impossible to test and create a special test plan for each areas (analysis, simulation, module testing, etc)
24. Chose some Software Engineering Methodology that includes:
 - a. Design Reviews
 - b. Code Reviews
 - c. Configuration control
 - d. Configurable Tools
 - e. Bug tracking

In [Part 2](#) – we will look at Memory Leaks

¹ An excellent example (albeit a bit over-the-top) is available as *MISRA-C2004 Guidelines for the use of the C language in critical systems*. See <http://www.misra.org.uk/>

² An example of a host of these kind of white papers is *Development Guidelines for Dependable Real-Time Embedded Systems* available from the IEEE
<http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/4488216/4493499/04493674.pdf?temp=x>