

Guidelines for Creating Robust Embedded Systems

Part 4 – Referencing I/O

September 17, 2009

By Bob Japenga

Scope

This is Part 4 of a white paper 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). This part will address guidelines for referencing I/O for reliable and robust operation. These guidelines are most applicable to systems where an errant output would create irreversible or catastrophic results (shut down a power plant, cause someone's heart to stop, etc). These systems usually require redundant I/O in hardware and careful systems and software design to prevent single point failures.

Limit the availability of I/O to be accessed from all places

The goal with these guidelines is to prevent run away software from accidentally triggering I/O. Many embedded systems could care less about this. Errant I/O will do no harm in most systems. But sometimes the outputs are critical and can cause problems or in other cases the software can actually harm the hardware with improper ordering of I/O instructions.

I have to admit that although this is a guideline that we generally follow, when we did not follow it, it has never (to our knowledge) gotten us in trouble.

Most modern OS's give us this protection. But many of us are still writing robust embedded systems for DOS and other microprocessors with not operating systems so this is still an issue. Even with systems without OS's, there are still tricks that can be done to prevent accidentally outputting the wrong data or at the wrong time. Even when using OS's with address and I/O protected spaces, kernel space software usually has no protection and all I/O can be referenced at all times.

None of these techniques are fool proof. But they can, in the right combination, reduce the probability of harmful occurrences to almost zero.

If the output of a particular signal is very serious, we need help from the hardware folks to create some sort of hardware interlock. Basically, without the hardware interlock set, the software cannot output the critical output. In areas of ultra critical outputs, the hardware interlock should be provided in hardware. With slightly less critical outputs, the interlock can be provided with a separate software controlled signal. In developing software controlled critical outputs it should always take two instructions to create the output (or two functions). If possible (because you thought about this at systems design), the first one should "arm" the output with a hardware interlock. If possible, the

output instructions should be as dissimilar as possible (either in addressing or operation). We have done this when we put software in the loop for nuclear safety, for armament systems, for fly-by-wire guidance systems or even for managing data that absolutely cannot be lost.

Other I/O is less critical but perhaps can create minor problems or nuisances. Basically we can use the same technique but use a software interlock. You start by limiting the number of instructions that actually write to the I/O. Then provide two calls to actuate the output. It doesn't provide a lot of protection to follow these guidelines and then write the code:

```
armTheOutput(23);  
triggerTheOutput(23);
```

Errant or run away software can just as easy jump to this code. Adding a parameter will provide some protection. But it is better if the application can arm and trigger in as separate a way as possible.

And then, when all is done, don't do as we did and not provide some sort of check that the arming mechanism (either software or hardware) is always armed. One time we designed such a mechanism and found that the hardware "arming" mechanism was always set. Paraphrasing Bob Vila¹ "Verify twice and cut down on bugs." In other words, verify by analysis that you did it right and then actually test it.

Finally, as is imposed by a number of operating systems, have all I/O go through kernel level drivers. If allowed by the operating system and hardware, limit the I/O space of each kernel level driver.

In [Part 5](#) we will discuss handling out of memory conditions.

¹ "Measure twice and cut once"