The Ganssle Group

Perfecting the Art of Building Embedded Systems

The Ganssle Group
PO Box 38346, Baltimore, MD 21231
Email info@ganssle.com
© 2000,2001, 2002 The Ganssle Group

Tips for finding bugs before they occur.

Published in ESP, January 2001

Proactive Debugging Ideas

Academics who study software engineering have accumulated an impressive number of statistics about where bugs come from and how many a typical program will have once coding stops but before debugging starts. Somehow I can't find any data that indicates how much time typical organizations spend actually removing these defects from their applications.

After visiting hundreds of companies and chatting with thousands of developers I'm left with a qualitative sense that code debug burns about half the total engineering time for most products. That suggests minimizing debugging is the first place to focus our schedule optimization efforts.

Defect reduction is not rocket science. Start with a well-defined spec/requirements document, use a formal process of inspections on all work products (from the spec to the code itself), and give the developers powerful and reliable tools. Skip any of these and bugs will consume too much time and sap our spirits.

But no process, no matter how well defined, will eliminate all defects. We make mistakes!

Typically 5-10% of the source code will be wrong. Inspections catch 70% to 80% of those errors. A little 10,000 line program will still, after employing the very best software engineering practices, contain hundreds of bugs we've got to chase down before shipping. Use poor practices and the number (and development time) skyrockets.

Debugging is hard, slow, and frustrating. Since we know we'll have bugs, and we know some will be godawful hard to find, let's look for things we can do to our code to catch problems automatically or more easily. I call this *proactive debugging*, which is the art of anticipating problems and instrumenting the code accordingly.

Stacks and Heaps

Do you use the standard, well known method for computing stack size? After all, undersize the stack and your program will crash in a terribly-hard-to-find manner. Allocate too much and you're throwing money away. High speed RAM is expensive.

The standard stack sizing methodology is to take a wild guess and hope. There is no scientific approach, nor even a rule of thumb. This isn't too awful in a single-task environment, since we can just stick the stack at the end of RAM and let it grow downwards, all the time hoping, of course, that it doesn't bang into memory already used. Toss in an RTOS, though, and such casual and sanguine allocation fails, since every task needs its own stack, each of which we'll allocate using the "take a guess and hope" methodology.

Take a wild guess and hope. Clearly this means we'll be wrong from time to time; perhaps even usually wrong. *If we're doing something that will likely be wrong, proactively take some action to catch the likely bug.*

Some RTOSes, for instance, include monitors that take an exception when any individual stack grows dangerously small. Though there's a performance penalty, consider turning these on for initial debug.

In a single task system, when debugging with an emulator or other tool with lots of hardware breakpoints, configure the tool to automatically (every time you fire it up) set a memory-write breakpoint near the end of the stack.

As soon as I allocate a stack I habitually fill each one with a pattern, like 0x55aa, even when using more sophisticated stack monitoring tools. After running the system for a minute or a week – whatever's representative for that application – I'll stop execution with the debugger and examine the stack(s). Thousands of words loaded with 0x55aa means I'm wasting RAM… and few or none of these words means the end is near, the stack's too small, and a crash is immanent.

Heaps are even more problematic. `Malloc()` is a nightmare for embedded systems. As with stacks, figuring the heap size is tough at best, a problem massively exacerbated by multitasking. `Malloc()` leads to heap fragmentation – though it may contain vast amounts of free memory, the heap may be so broken into small, unusable chunks that `malloc()` fails.

In simpler systems it's probably wise to avoid `malloc()` altogether. When there's enough RAM allocating all variables and structures statically yields the fastest and most deterministic behavior, though at the cost of using more memory.

When dynamic allocation is unavoidable, by all means remember that `malloc()` has a return value! I look at a tremendous amount of firmware yet rarely see this function tested. It must be a guy thing. Testosterone. We're gonna `malloc` that puppy, by gawd, and that's that! Fact is, it may fail, which will cause our program to crash horribly. If we're smart enough – proactive enough – to test every `malloc()` then an allocation error will still cause the program to crash horribly, but at least we can set a debug trap, greatly simplifying the task of finding the problem.

An interesting alternative to `malloc()` is to use multiple heaps. Perhaps a heap for 100 byte allocations, one for 1000 bytes and another for 5000. Code a replacement `malloc()` that takes the heap identifier as its argument. Need 643 bytes? Allocate a 1000 byte block from the 1000 byte heap. Memory fragmentation becomes extinct, your code runs faster, though some RAM will be wasted for the duration of the allocation. A few commercial RTOSes do provide this sort of replacement `malloc().`

Finally, if you do decide to use the conventional `malloc(),` at least for debugging purposes link in code to check the success of each allocation. www.snippest.org/MEM.TXT (the link is case-sensitive) and companion files is Walter Bright's memory allocation test code, put into the public domain many years ago. MEM is a few hundred lines of C that replaces the library's standard memory functions with versions that diagnose common problems.

MEM looks for out-of-memory conditions, so if you've inherited a lot of poorly written code that doesn't properly check `malloc()`'s return value, use MEM to pick up errors. It verifies that frees match allocations. Before returning a block it sets the memory to a non-zero state to increase the likelihood that code expecting an initialized data set fails.

An interesting feature is that it detects pointer over- and under-runs. By allocating a bit more memory than you ask for, and writing a signature pattern into your pre- and post-buffer memory, when the buffer is freed MEM can check to see if a pointer wandered beyond the buffer's limits.

Geodesic Systems (www.geodesic.com) builds a commercial and much more sophisticated memory allocation monitor targeted at desktop systems. They claim that 99% of all PC programs suffer from memory leaks (mostly due to memory that is allocated but never freed). I have no idea how true this statement really is, but the performance of my PC sure seems to support their proposition. On a PC a memory leak isn't a huge problem, since the programs are either closed regularly or crash sufficiently often to let the OS reclaim that leaked resource.

Firmware, though, must run for weeks, months, even years without crashing. If 99% if PC apps suffer from leaks, I'd imagine a large number of embedded projects share similar problems. One of MEM's critical features is that it finds these leaks, generally before the system crashes and burns.

MEM is a freebie and requires only a small amount of extra code space, yet will find many classes of very common problems. The wise developer will link it, or other similar tools, into every project proactively, before the problems surface.

For those who want more powerful tools, Applied Microsystems (www.amc.com) provides CodeTEST, a hardware tool targeted at embedded projects that tracks memory and other problems.

Seeding Memory

Bugs lead to program crashes. A "crash", though, can be awfully hard to find. First we notice a symptom – the system stops responding. If the debugger is connected, stopping execution shows that the program has run amok. But why? Hundreds of millions of instructions might elapse between ours seeing the problem and starting troubleshooting. No trace buffer is that large.

So we're forced to recreate the problem – if we can – and use various strategies to capture the instant when things fall apart. Yet "crash" often means that the code branches to an area of memory where it simply should not be. Sometimes this is within the body of code itself; often it's in an address range where there's neither code nor data.

Why do we continue to leave our unused ROM space initialized to some default value that's a function of the ROM technology and not what makes sense for us? Why don't we make a practice of setting all unused memory, both ROM and RAM, to a software interrupt instruction that immediately vectors execution to an exception handler?

Most CPUs have single byte or single word opcodes for a software interrupt. The Z80's RST7 was one of the most convenient, as it's 0xff which is the defaults state of unprogrammed EPROM. x86 processors all support the single byte INT3 software interrupt. Motorola's 68k family, and other processors, have an illegal instruction word.

Set all unused memory to the appropriate instruction, and write a handler that captures program flow if the software interrupt occurs. The stack often contains a wealth of clues about where things were and what was going on when the system crashed, so copy it to a debug area. In a multitasking application the OS's task control block and other data structures will have valuable hints. Preserve this critical tidbits of information.

Make sure the exception handler stops program flow; lock up in an infinite loop or something similar, insure all interrupts and DMA are off, so to stop the program from wandering away.

There's no guarantee that seeding memory will capture all crashes, but if it helps in even a third of the cases you've got a valuable bit of additional information to help diagnose problems.

But there's more to initializing memory than just seeding software interrupts. Other kinds of crashes require different proactive debug strategies. For example, a modern microprocessor might support literally hundreds of interrupt sources, with a vector table that dispatches ISRs for each. Yet the average embedded system might use a few, or perhaps a dozen, interrupts. What do we do with the unused vectors in the table?

Fill them, of course, with a vector aimed at an error handler! It's ridiculous to leave the unused vectors aimed at random memory locations. Sometime, for sure, you'll get a spurious interrupt, something awfully hard to track down. These come from a variety of sources, such as glitchy hardware (you're probably working on a barely functional hardware prototype, after all).

More likely is a mistake made in programming the vectors into the interrupting hardware. Peripherals have gotten so flexible that they're often impossible to manage. I've used parts with hundreds of internal registers, each of which has to be set just right to make the device function properly. Motorola's TPU, which is just a lousy timer, has a 142 page databook that documents some 36 internal registers. For a timer. I'm not smart enough to set them correctly first try, every time. Misprogramming any of these complex peripherals can easily lead to spurious interrupts.

The error handler can be nothing more than an infinite loop. Be sure to set up your debug tool so that every time you load the debugger it automatically sets a breakpoint on the handler. Again, this is nothing more than anticipating a tough problem, writing a tiny bit of code to capture the bug, and then configuring the tools to stop when and if it occurs.

In March 2000 (*More on Wandering Code*) I wrote about another memory problem that surfaces too often: sometimes code accesses memory in silly ways. Hook a logic analyzer to an embedded product and you might find that it reads or writes to memory that is not used. Or writes to ROM. Clearly these are harbingers of problems, even if an observable symptom hasn't manifested itself. As I described in that article, a very little bit of simple logic, or the use of extra chip select pins, creates an instant "stupid memory access" monitor that can find bugs you may never know existed… till a customer complains about odd behaviors.

Testing

My new Toyota's Check Engine light came on a week after picking up the car. As an old embedded hand I knew exactly what to do: pull over, turn the engine off, and restart. The light went out and hasn't come back on.

Even my kids know how to deal with quirky toys and other electronic goodies: cycle power. As embedded products become ever more common we customers see all sorts of interesting and frustrating failures.

Perhaps one of the most proactive things we can do to isolate bugs is to build a comprehensive test plan, rather than the ad hoc qualification that seems de rigor.

I was talking to a group of developers from an appliance company, one that builds blenders. Being a wise guy I joked that they probably get to test these products with Margaritas. "Well, yes," they replied, "but due to the political climate we test with virgin Margaritas."

But that changed. After shipping the first of a new series of blenders all failed. Turns out, the alcohol depresses the freezing point of the mix, loading the motor more, burning out the drive transistors. Now they test with real Margaritas.

These are the happiest engineers I've ever met…