Testing RAM in Embedded Systems

Version 2: April 2007

Version 2.1: April 2007

Jack G. Ganssle
jack@ganssle.com

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 504-6660
fax (647) 439-1454

Developers often adhere to beliefs about the right way to test RAM that are as polarized as disparate feelings about politics and religion. Most are simply wrong. I read a lot of code – a *lot* – and it's rare to find a RAM test that makes much sense.

Obviously, a RAM problem will destroy most embedded systems. Errors reading from the stack will sure crash the code. Problems, especially intermittent ones, in the data areas may manifest bugs in subtle ways. Often you'd rather have a system that just doesn't boot, rather than one that occasionally returns incorrect answers.

Some embedded systems are pretty tolerant of memory problems. We hear of NASA spacecraft from time to time whose core or RAM develops a few bad bits, yet somehow the engineers patch their code to operate around the faulty areas, uploading the corrections over the distances of billions of miles.

Most of us work on systems with far less human intervention. There are no teams of highly trained personal anxiously monitoring the health of each part of our products. It's our responsibility to build a system that works properly when the hardware is functional.

In some applications, though, a certain amount of self-diagnosis either makes sense or is required; critical life support applications should use every diagnostic concept possible to avoid disaster due to a sub-micron RAM imperfection.

So, my first belief about diagnostics in general, and RAM tests in particular, is to clearly define your goals. Why run the test? What will the result be? Who will be the unlucky recipient of the bad news in the event an error is found, and what do you expect that person to do?

Will a RAM problem kill someone? If so, a very comprehensive test, run regularly, is mandatory.

Is such a failure merely a nuisance? For instance, if it keeps a cell phone from booting, if there's nothing the customer can do about the failure anyway, then perhaps there's no reason for doing a test. As a consumer I couldn't care less why the phone stopped working… if it's dead I'll take it in for repair or replacement.

Is production test - or even engineering test - the real motivation for writing diagnostic code? If so, then define exactly what problems you're looking for and write code that will find those sorts of troubles.

Next, inject a dose of reality into your evaluation. Remember that today's hardware is often very highly integrated. In the case of a microcontroller with on-board RAM the chances of a memory failure that doesn't also kill the CPU is small. Again, if the system is a critical life support application it may indeed make sense to run a test as even a minuscule probability of a fault may spell disaster.

Does it make sense to ignore RAM failures? If your CPU has an illegal instruction trap, there’s a pretty good chance that memory problems will cause a code crash you can capture and process. If the chip includes protection mechanisms (like the x86 protected mode), count on bad stack reads immediately causing protection faults your handlers can process. Perhaps RAM tests are simply not required given these extra resources.

Inverting Bits

Too many developers use the simplest of tests - writing alternating 0x55 and 0xAA values to the entire memory array, and then reading the data to ensure it remains accessible. It’s a seductively easy approach that will find an occasional problem (like, someone forgot to load all of the RAM chips), but that detects few real world errors.

Remember that RAM is an array divided into columns and rows. Accesses require proper chip selects and addresses sent to the array -- and not a lot more. The 0x55/0xAA symmetrical pattern repeats massively all over the array; accessing problems (often more common than defective bits in the chips themselves) will create references to incorrect locations, yet almost certainly will return what appears to be correct data.

Consider the physical implementation of memory in your embedded system. The processor drives address and data lines to RAM - in a 16 bit system there will surely be at least 36 of these. Any short or open on this huge bus will create bad RAM accesses. Problems with the PC board are far more common than internal chip defects, yet the 0x55/0xAA test is singularly poor at picking up these, the most likely, failures.

Yet, the simplicity of this test and its very rapid execution has made it an old standby used much too often. Isn’t there an equally simple approach that will pick up more problems?

If your goal is to detect the most common faults (PCB wiring errors and chip failures more substantial than a few bad bits here or there), then indeed there is. Create a short string of almost random bytes that you repeatedly send to the array until all of memory is written. Then, read the array and compare against the original string.

I use the phrase “almost random” facetiously, but in fact it little matters what the string is, as long as it contains a variety of values. It’s best to be include the pathological cases, like 00, 0xaa, ox55, and 0xff. The string is something you pick when writing the code, so it is truly not random, but other than these four specific values you fill the rest of it with nearly any set of values, since we’re just checking basic write/read functions (remember: memory tends to fail in fairly dramatic ways). I like to use very orthogonal values - those with lots of bits changing between successive string members - to create big noise spikes on the data lines.

To make sure this test picks up addressing problems, ensure the string’s length is not a factor of the length of the memory array. In other words, you don’t want the string to be aligned on the same low-order addresses, which might cause an address error to go undetected. Since the string is much shorter than the length of the RAM array, you ensure it repeats at a rate that is not related to the row/column configuration of the chips.

For 64k of RAM, a string 257 bytes long is perfect. 257 is prime, and its square is greater than the size of the RAM array. Each instance of the string will start on a different low order address. 257 has another special magic: you can include every byte value (00 to 0xff) in the string without effort. Instead of manually creating a string in your code, build it in real time by incrementing a counter that overflows at 8 bits.

Critical to this, and every other RAM test algorithm, is that you write the pattern to all of RAM before doing the read test. Some people like to do non-destructive RAM tests by testing one location at a time, then restoring that location's value, before moving on to the next one. Do this and you'll be unable to detect even the most trivial addressing problem.

This algorithm writes and reads every RAM location once, so is quite fast. Improve the speed even more by skipping bytes, perhaps writing and reading every 3rd or 5th entry. The test will be a bit less robust yet will still find most PCB and many RAM failures.

Some folks like to run a test that exercises each and every bit in their RAM array. Though I remain skeptical of the need since most semiconductor RAM problems are rather catastrophic, if you do feel compelled to run such a test, consider adding another iteration of the algorithm just described, with all of the data bits inverted.

Detailed Diagnostics

Sometimes, though, you'll want a more thorough test, something that looks for difficult hardware problems at the expense of speed.

When I speak to groups I'll often ask "what makes you think the hardware *really* works?" The response is usually a shrug of the shoulders, or an off-the-cuff remark about everything seeming to function properly, more or less, most of the time.

These qualitative responses are simply not adequate for today's complex systems. All too often a prototype that seems perfect harbors hidden design faults that may only surface after you've built a thousand production units. Recalling products due to design bugs is unfair to the customer and possibly a disaster to your company.

Assume the design is absolutely ridden with problems. Use reasonable methodologies to find the bugs before building the first prototype, but then use that first unit as a testbed to find the rest of the latent troubles.

Large arrays of RAM memory are a constant source of reliability problems. It's indeed quite difficult to design the perfect RAM system, especially with the minimal margins and high speeds of today's 16 and 32 bit systems. If your system uses more than a couple of RAM parts, count on spending some time qualifying its reliability via the normal hardware diagnostic procedures. Create software RAM tests that hammer the array mercilessly.

Probably one of the most common forms of reliability problems with RAM arrays is pattern sensitivity. Now, this is not the famous pattern problems of yore, where the chips (particularly DRAMs) were sensitive to the groupings of ones and zeroes. Today the chips are just about perfect in this regard. No, today pattern problems come from poor electrical characteristics of the PC board, decoupling problems, electrical noise, and inadequate drive electronics.

PC boards were once nothing more than wiring platforms, slabs of tracks that propagated signals with near perfect fidelity. With very high speed signals, and edge rates (the time it takes a signal to go from a zero to a one or back) under a nanosecond, the PCB itself assumes all of the characteristics of an electronic component - one whose virtues are almost all problematic. It's a big subject (refer to read "High Speed Digital Design -a Handbook of Black Magic" by Howard Johnson and Martin Graham (1993 PTR Prentice Hall, NJ for the canonical words of wisdom on this subject), but suffice to say a poorly designed PCB will create RAM reliability problems.

Equally important are the decoupling capacitors chosen, as well as their placement. Inadequate decoupling will create reliability problems as well.

Modern DRAM arrays are massively capacitive. Each address line might drive dozens of chips, with 5 to 10 pf of loading per chip. At high speeds the drive electronics must somehow drag all of these pseudo-capacitors up and down with little signal degradation. Not an easy job! Again, poorly designed drivers will make your system unreliable.

Electrical noise is another reliability culprit, sometimes in unexpected ways. For instance, CPUs with multiplexed address/data buses use external address latches to demux the bus. A signal, usually named ALE (Address Latch Enable) or AS (Address Strobe) drives the clock to these latches. The tiniest, most miserable amount of noise on ALE/AS will surely, at the time of maximum inconvenience, latch the data part of the cycle instead of the address. Other signals are also vulnerable to small noise spikes.

Many run of the mill RAM tests, run for several hours, as you cycle the product through its design environment (temperature, etc) will show intermittent RAM problems. These are symptoms of the design faults I've described, and always show a need for more work on the product's engineering.

Unhappily, all too often the RAM tests show no problem when hidden demons are indeed lurking. The algorithm I've described, as well as most of the others commonly used, tradeoff speed versus comprehensiveness. They don't pound on the hardware in a way designed to find noise and timing problems.

Digital systems are most susceptible to noise when large numbers of bits change all at once. This fact was exploited for data communications long ago with the invention of the

Gray Code, a variant of binary counting, where no more than one bit changes between codes. Your worst nightmares of RAM reliability occur when all of the address and/or data bits change suddenly from zeroes to ones.

For the sake of engineering testing, write RAM test code that exploits this known vulnerability. Write 0xffff to 0x0000 and then to 0xffff, and do a read-back test. Then write zeroes. Repeat as fast as your loop will let you go.

Depending on your CPU, the worst locations might be at 0x00ff and 0x0100, especially on 8 bit processors that multiplex just the lower 8 address lines. Hit these combinations, hard, as well.

Other addresses often exhibit similar pathological behavior. Try 0x5555 and 0xaaaa, which also have complementary bit patterns.

The trick is to write these patterns back-to-back. Don't test all of RAM, with the understanding that both 0x0000 and 0xffff will show up in the test. You'll stress the system most effectively by driving the bus massively up and down all at once.

Don't even think about writing this sort of code in C. Any high level language will inject too many instructions between those that move the bits up and down. Even in assembly the processor will have to do fetch cycles from wherever the code happens to be, which will slow down the pounding and make it a bit less effective.

There are some tricks, though. On a CPU with a prefetcher (all x86, 68k, etc.) try to fill the execution pipeline with code, so the processor does back-to-back writes or reads at the addresses you're trying to hit. And, use memory-to-memory transfers when possible. For example:

```
mov    si,0xaaaa
mov    di,0x5555
mov    [si],0xff
mov    [di],[si]
```

This is killer code that will show up all kinds of unexpected problems in a system with marginal electronic design.

On-The-Fly RAM Tests

In 1975 MITS shocked the techie world when they introduced the Altair computer for $400 (in kit form), the same price Intel was charging for the machine's 8080 CPU. According to http://www.bls.gov/cpi/ that's $1500 in today's dollars, which would buy a pretty nifty Dell running at 1000 times the Altair's clock rate, with a dual-core processor, quarter-terabyte of hard disk, a 19" flat panel monitor, and 2 GB of RAM. The Altair included neither keyboard, nor monitor, had only 256 bytes (that's not a typo) of RAM, and no mass storage.

I'd rather have the Dell.

The 256 bytes of RAM were pretty limiting, so the company offered a board with 4k of DRAM for $195 (also in kit form). These boards were simply horrible and offered reliably unreliable performance. Their poor design ensured they'd drop bits randomly and frequently.

Though MITS eventually went out of business the computer revolution they helped midwife grew furiously. As did our demand for more and cheaper memory. Intel produced the 2107 DRAM which stored 16k bits of data. But users reported unreliable operation which by 1978 was traced to the radioactive decay of particles in the chip's packaging material. It turns out that the company built a new fab on the Green River in Colorado, downstream of an abandoned uranium mine. The water used to manufacture the ceramic packages was contaminated with trace amounts of radioactivity. The previous generation of DRAM had a storage charge of about 4 million electrons per bit; the 2107, using smaller geometry, reduced that to one million, about the energy from an alpha particle.

Polonium 210, the same stuff that reputedly killed Russian ex-spy Alexander Litvinenko, occurs naturally in well water and as a decay of product of radon gas. Yet far less than one part per billion, if it were in a DRAM package, would cause several bit flips per minute. This level of radioactivity is virtually undetectable, and led to a crisis at an IBM fab in 1987 when chips were experiencing occasionally random bit flips due to polonium contamination. Many months of work finally determined that one container of nitric acid was slightly "hot." The vendor's bottle cleaning machine had a small leak that occasionally emitted minute bits of Po^{210}. The story of tracing the source of the contamination reads like a detective thriller (IBM Experiments is Soft Fails in Computer Electronics, J. F. Ziegler et al, IBM Journal of Research Development volume 40 number 1, January 1996).

Cosmic rays, too, can flip logic bits, and it's almost impossible to build effective shielding against these high-energy particles. The atmosphere offers the inadequate equivalent of 13 feet of concrete shielding. Experiments in 1984 showed that memory devices had twice as many soft errors in Denver than at sea level.

A 2004 paper (Soft Errors in Electronic Memory by Tezzaron Semiconductor, http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf) shows that because DRAMs have shrunk faster than the charge per cell has gone down, today's parts are less vulnerable to cosmic ray effects than those from years ago. But these energetic particles from deep space can create havoc in new dense SRAMs and FPGAs. The paper claims a system with one GB of SRAM can expect a soft error every two weeks! According to http://www.eetimes.com/story/OEG20010817S0073 a particle with as little as a 10 femtocoulomb charge has enough energy to flip an SRAM bit; a decade ago the larger cells needed five times more energy.

Old-timers remember that most PCs had 9 bit memory configurations, with the ninth reserved for parity to capture DRAM errors. That's ancient history on the desktop, though servers often use parity or error-correcting logic to provide the high degree of reliability users expect. Desktop PC users expect, well, frequent crashes. I wonder how many of those are from cosmic rays, and not a result of Windows problems?

Some systems must run continuous RAM tests as the application executes, sometimes to find hard errors – actual hardware failures – that occur over time, and sometimes to identify soft errors. Are these tests valuable? One wonders. Any sort of error in the stack space will immediately cause the system to crash unpredictably. A hardware flaw – say an address or data line failure – takes out big chunks of memory all at once. Recovery might be impossible, though I've heard claims that memory failures in the deep-space Pioneer probe were "fixed" by modifying the code to avoid those bad locations.

But for better or worse, if your requirements demand on-the-fly RAM tests, how will you conduct these without altering critical variables and code?

Hard Errors

Since hard and soft errors manifest themselves in completely different ways the tests for these problems are very different. Let's look at hard errors first.

Before designing a test it's wise to consider exactly what kinds of failures may occur. On-chip RAM won't suffer from mechanical failures (such problems would probably take out the entire device), but off-chip memory uses a sometimes very complex net of wiring and circuitry. Wires break; socketed chips rattle loose or corrosion creates open circuits; and chips fail. Rarely – oh, so rarely – a single or small group of bits fail inside a particular memory device, but today such problems are hardly ever seen. It probably, except in systems that have the most stringent reliability requirements, makes sense to not look for single bit failures, as such tests are very slow. Most likely the system will crash long before any test elicits the problem.

Pattern sensitivity is another failure mode that used to be common, but which has all-but-disappeared. The "walking ones" test was devised to find such problems, but is computationally expensive and destroys the contents of large swaths of memory. There's little reason to run such a test now that the problem has essentially disappeared.

So the tests should look for dead chips and bad connections. If there's external memory circuitry, say bus drivers, decoders, and the like, any problem those parts experience will appear as complete chip failures or bad connections.

DRAMs have a special case when external circuitry generates refresh cycles (note that a lot of embedded processors take care of this internally). Unfortunately it's all but impossible to construct a test to find a refresh problem, other than at the initial power on self test when no other code is running. The usual approach fills memory with a pattern, and then stops all execution for a second or two before checking that the correct pattern still exists. Few users want to see their system periodically lock up for the sake of the test. On second thought, that explains the behavior of my PC…

I want to draw a distinction between actually *testing* RAM to ensure that it's functional, and that of insuring the contents of memory are *consistent*, or have reasonable values. The latter we'll look at when considering soft errors later in this paper.

Traditional memory tests break down when running in parallel with an executing application. We can't blithely fill RAM with a pattern that overwrites variables, stacks and maybe even the application code. On-the-fly RAM tests must cautiously poke at just a few locations at a time, and then restore the contents of these values. Unless there's some a priori information that the locations aren't in use, we'll have to stop the application code for a moment while conducting each step. Thus, the test runs sporadically, stealing occasional cycles from the CPU to check just a few locations at a time.

In simpler systems that constantly cycle through a main loop, it's probably best to stick a bit of code in the loop that checks a few locations and then continues on with other, more important activities. A static variable holds the last address tested so the code snippet knows where to pick up when it runs again. Alternatively run a periodic interrupt whose ISR checks a few locations and then returns. In a multitasking system a low-priority task can do the same sort of thing.

If any sort of preemption is happening, turn interrupts off so the test itself can't be interrupted with RAM in a perhaps unstable state. Pause DMA controllers as well and shared memory accesses.

But what does the test look like?

The usual approach is to stuff 0x5555 in a location, verify it, and then repeat using 0xAAAA. That checks exactly nothing. Snip an address line with wire cutters: the test will pass. Nothing in the test proves that the byte was written to the *correct* address.

Instead, let's craft an algorithm that checks address and data lines. For instance:

```
bool test_ram(){
unsigned int save_lo, save_hi;
bool error = FALSE;
static unsigned int test_data=0;
static unsigned long *address = START_ADDRESS;
static unsigned int offset;

  push_intr_state();
  disable_interrupts();
9   save_lo                 = *address;

10   for(offset=1; offset<=0x8000; offset=offset<<1){
11    save_hi               = *(address+offset);
12    *address              =   test_data;
13    *(address+offset)     =  ~test_data;
14    if(*address           !=  test_data)error=TRUE;
15    if(*(address+offset) != ~test_data)error=TRUE;
16    *(address+offset)     =  save_hi;
17    test_data+=1;}
18  *address                =  save_lo;

19  pop_intr_state();
20  return error;}
```

`START_ADDRESS` is the first location of RAM. In lines 9 and 11, and 16 and 18, we save and restore the RAM locations so that this function returns with memory unaltered. But the range from line 9 to 18 is a “critical section” – an interrupt that swaps system context while we’re executing in this range may invoke another function that tries to access these same addresses. To prevent this line 8 disables interrupts (and be sure to shut down DMA, too, if that’s running). Line 7 preserves the state of the interrupts; if `test_ram()` were invoked with them off we sure don’t want to turn them enabled! Line 19 restores the interrupts to their pre-disabled state. If you can guarantee that `test_ram()` will be called with interrupts enabled, simplify by removing line 7 and changing 19 to a starkly minimal interrupt enable.

The test itself is simplicity itself. It stuffs a value into the first location in RAM, and then, by shifting a bit and adding that to the base address, to other locations separated by an address line. This code is for a 64k space, and in 16 iterations it ensures that the address, data, and chip select wiring is completely functional, as is the bulk functionality of the memory devices.

To cut down interrupt latency, you can remove the loop and test one pair of locations per call.

The code does not check for the uncommon problem of a few locations going bad inside a chip. If that’s a concern construct another test that replaces lines 10 to 18 with:

```
*address               =   0x5555;
if(*address            !=  0x5555)error=TRUE;
*address               =   0xAAAA;
if(*address            !=  0xAAAA)error=TRUE;
*address               =  save_lo;
address+=1;
```

… which cycles every bit at every location, testing one address each time the routine is called. Despite my warning above, the 0x5555/0xAAAA pair works because the former test checked the system's wiring.

There are a lot of caveats – don't program these in C, for instance, unless you can ensure the tests won't touch the stack. And the value of these tests is limited since address and data bus errors most likely will crash the system long before the test runs. But in some applications using banks of memory a wiring fault might affect just a small subsection of RAM. In other systems the on-going test is important, even if meaningless, to meet the promises some naïve fool in marketing printed on the brochure.

While many systems sport power-on self-tests, in some cases, for good reasons or bad, we're required to check RAM continuously as the system runs. Since the RAM test necessarily runs very slowly, so to not turn the CPU into a tortoise, I think that in most cases these tests are ineffective. Most likely the damage from bad RAM will be done long before the test detects a problem. But sometimes the tests are required for regulatory or marketing reasons. It makes sense to try to craft the best solution possible in the hope that the system will pick up at least some errors.

It'll ease your conscience as well. No one wants to write crummy code.

Soft Errors

Soft errors are transient. Conventional RAM tests, as described earlier, won't generally detect them. Instead of looking for errors per se, a soft RAM test checks for memory *consistency*. Does each location contain what we expect?

It's a simple matter to checksum memory and compare that to a previous sum or a known-good value. But variables change constantly in a running program. That's why they're called "variables." It's generally futile to try and track the microsecond to microsecond contents of hundreds or thousands of these, unless you have some knowledge that an important subset stays constant… ah, "constants" for instance.

Sometimes it takes several machine cycles to set a variable. A 16 bit processor updating a 32 bit long will execute several machine instructions to save the data. An interrupt between those instructions leaves the variable half-changed. If your RAM test runs as a task or is invoked by a timer ISR, it may examine a variable in an indeterminate state. Be sure to preserve the reentrancy of every value that's part of the RAM test.

I once saw an interesting solution to the problem of finding soft errors in a program's data space. Every access to every variable was encapsulated by a driver, which computed a new checksum for the data block after each write operation. That's simpler than re-summing memory; instead figure how the checksum must change based on the previous and new values of the item being changed. Since two values – the variable and checksum – were updated the code used semaphores to be reentrant-safe. Though slow and cumbersome, it did work.

Many systems boot from relatively-slow flash, copy the code into faster RAM, and run the code from RAM. Others boot from a network into RAM. Ulysses had himself tied to a mast to resist the temptation to write self-modifying code (the Odyssey was all Greek to me), a wise move that, if emulated, makes it trivial to look for soft errors in the code. On boot checksum the executable and have the RAM test repeat that checksum, comparing the result to the initial value.

Since there's never a write to firmware, don't worry about reentrancy when checksumming the program.

It's reasonable to run such tests on RAM-based program code since the firmware doesn't change during execution. And the odds are good that if an error occurs, the application won't crash before the check gets around to testing the bad location, since the program likely spends much of its time stuck in loops or off handling other activities. In today's feature-rich products the failure could occur in a specific feature that's not often used.

Is a CRC better than a simple checksum? After all, if a couple of bits in different locations fail it's entirely possible the checksum won't signal a problem. CRCs dominate communications technology since noisy channels can corrupt many bytes. But they're computationally much more expensive than a checksum, and one goal for RAM testing is to run through all locations relatively quickly while leaving as many CPU cycles as possible for the system's application code. And soft RAM errors are not like noisy communications; most likely a soft error will result in merely one word being wrong. I prefer checksums' simplicity and speed.

Caveats

While marketers might like the sound of "always self-tests itself!" on a datasheet, the reality is rather grim. It's tough to do any sort of consistency check on quickly-changing locations, such as the stack. The best one can hope for is to check only sections of the memory array.

Unfortunately on-the-fly RAM tests pose a Hobson's Choice with respect to performance. We want to run the tests often and fast, picking up errors before they wreak havoc. But that's computationally expensive, especially when reentrancy considerations mandate taking locks or disabling interrupts. If these tests are truly important one must push the application's code to a low priority and lavish the bulk of CPU cycles on the tests. Few of us are willing to do that.

Processors with caches pose particularly-thorny problems. Any cached value will not get tested. One can glibly suggest a flush before each test, but, since the check runs often and fast, such action will more or less make the cache worthless.

Dual-ported RAM or memory shared between processors must be locked during the tests. It's possible to use a separate CPU just to run the test (I've seen it done), but bus bandwidth will cripple the application processor's performance.

Finally, what do you do if the test detects and error? Log the problem, leaving some debugging breadcrumbs behind. But don't continue to run the application code. Recovery, unless there's some a priori knowledge about the problem, is usually impossible. Halt and let the watchdog issue a reset or go into a safe mode.

Hi Rel Apps

Soft RAM errors are a reality that some systems cannot tolerate. You'd hate to lose a billion-dollar space mission from a single microsecond-long glitch. Banks might not be too keen on a system in which a cosmic ray changes variable `check_amount` from $1.00 to $1000001.00. Worse would be holding a press conference to explain the loss of 100 passengers because the code, though perfect, read one wrong location and "crashed."

In these situations we need to *mitigate*, rather than test for, errors. When failure is not an option it's time to rely on additional hardware. The standard solution is to widen each RAM location to add an error correcting code (ECC). There will be a substantial amount of hardware needed to encode and decode ECCs on every memory access as well.

A word of 2^n bits needs n+1 additional check bits to correct for any single-bit error. That is, widen RAM by 6 bits to fix any one bit error in a 32 bit word. Use n+2 extra bits to correct any single-bit error, but to detect (and flag) two-bit errors.

Note that ECC will protect any RAM error, even one that's in the stack.

A few wrinkles remain. Poor physical organization of the memory array can defeat any ECC scheme. In the olden days DRAM was available only in one bit wide chips. A 16KB array used sixteen 16kb x 1 devices. Today vendors sell all sorts of wide (x 4, x 8, etc) configurations. If the ECC code is stored in the same chip as the data, a multiple-bit soft error may prevent the error correction… even detection. Proper design means separating data and codes into different parts. The good news is that cosmic rays usually only toggle a single bit.

Another problem: soft errors can accumulate in memory. A proton zaps location 0x1000. The program reads that location and sends the corrected value to the CPU. But 0x1000 still has an incorrect value stored. With enough bad luck another cosmic ray may strike again; now the location has two errors, exceeding the hardware correction capability. The system crashes and Sixty Minutes is knocking at your door.

Complement ECC hardware with "scrubbing" code that occasionally reads and rewrites every location in RAM. ECC will clean up the data as it goes to the processor; the rewrite fixes the bad location in memory. A minimally-intrusive background task can suck a few cycles now and then to perform the scrub. Reentrancy is an issue.

Some systems invoke a DMA controller every few hours to scrub RAM. Though DMA sounds like the perfect solution it usually runs independent of the program's operation and may corrupt any non-reentrant activity going on in parallel. Unless you have an exquisite sense of how the code updates every variable, be wary of DMA in this application.

ECC is expensive. Are there any software-only solutions?

Sometimes developers maintain multiple copies of critical variables, encapsulating access to them through a driver that checks to ensure all are identical. That leaves the stack unprotected. The heap might be vulnerable as well, since a corrupt entry in the allocation table (invisible to the C programmer) could take out every `malloc()` and `free()`. One could preallocate every block on boot, though then the heap offers little of value over fixed arrays.

Such copies may use as much extra memory as does ECC.

There may be some value in protecting critical, slowly-changing, data, but my observation is that developers typically value ease of implementation over doing a real failure analysis.

Conclusion

Soft RAM errors will become more of a problem as memory sizes grow and device geometry shrinks. Yet they're nearly intractable in terms of software solutions, and hardware approaches are costly and eat plenty of PCB real estate. If your CPU has an MMU, at the very least turn that on and build tasks in their own memory spaces. A stack error, for instance, will likely crash just one task. The MMU exception handler can take some sort of action to restore the code or put the system into a safe state.

Better Firmware... *Faster!*

A One Day Seminar

April 23, 2008 - Chicago
Hilton, Oak Lawn
9333 Cicero Ave.
Oak Lawn, IL

April 25, 2008 - Denver
Embassy Suites Hotel
7525 East Hampden Ave.
Denver Tech Center
Denver, CO

May 19, 2008
London, UK
Jurys Inn Heathrow
Eastern Perimeter Rd.
Hatton Cross, Hounslow

Presented by Jack Ganssle, technical editor of *Embedded Systems Programming Magazine*, **author** of *6 books and over 600 articles*

Registration form on last page of this brochure

Limited seating; sign up now and guarantee a spot.

The Ganssle Group
PO Box 38346
Baltimore, MD 21231
(410) 504-6660
fax: (647) 439-1454

register@ganssle.com
www.ganssle.com

For Engineers and Programmers

This seminar will teach you new ways to build higher quality products in half the time.

80% of all embedded systems are delivered late…

Sure, you can put in more hours. Be a hero. But *working harder is not a sustainable way to meet schedules.* We'll show you how to plug productivity leaks. How to manage creeping featurism. And ways to balance the conflicting forces of schedules, quality and functionality.

… yet it's not hard to double development productivity

Firmware is the most expensive thing in the universe, yet we do little to control its costs. Most teams deliver late, take the heat for missing the deadline, and start the next project having learned nothing from the last. Strangely, *experience* is not correlated with *fast*. But *knowledge* is, and we'll give you the information you need to build code more efficiently, gleaned from hundreds of embedded projects around the world.

Bugs are the #1 cause of late projects…

New code generally has *50 to 100 bugs* per thousand lines. Traditional debugging is the *slowest* way to find bugs. We'll teach you better techniques proven to be up to 20 times more efficient. And show simple tools that find the nightmarish real-time problems unique to embedded systems.

… followed by poor scheduling

Though capricious schedules assigned without regard for the workload are common, even developers who make an honest effort usually fail. We'll show you how to decompose a product into schedulable units, and how to use killer techniques like Wideband Delphi to create more accurate estimates.

Learn from the Industry's Guru

Spend a day with Jack Ganssle, well-known author of the most popular books on embedded systems, technical editor and columnist for Embedded Systems Programming, and designer of over 100 embedded products. You'll learn new ways to produce projects *fast* without sacrificing quality. This seminar is the only non-vendor training event that shows you *practical* solutions that you can implement *immediately*. We'll cover technical issues – like how to write embedded drivers and isolate performance problems – as well as practical process ideas, including how to manage your people and projects. *After taking this class you'll receive a certificate awarding you 0.7 Continuing Education Units.*

Seminar Leader

Jack Ganssle has written over 600 articles in Embedded Systems Programming, EDN, and other magazines. His five books, **The Art of Programming Embedded Systems**, **The Art of Developing Embedded Systems**, The **Embedded Systems Dictionary**, **The Firmware Handbook**, and **Embedded Systems, World Class Designs** are the industry's standard reference works

Jack lectures internationally at conferences and to businesses, and has been the keynote speaker at the Embedded Systems Conferences in both Boston and San Francisco. He founded three companies, including one of the largest embedded tool providers. His extensive product development experience forged his unique approach to building better firmware faster.

Jack has helped over 600 companies and thousands of developers improve their firmware and consistently deliver better products on-time and on-budget.

Course Outline

Languages

- C, C++ or Java?
- Code reuse—a myth? How can you benefit?
- Controlling stacks and heaps.

Structuring Embedded Systems

- *Manage* features… or miss the schedule!
- Using multiple CPUs.
- Five design schemes for faster development.

Overcoming Deadline Madness

- Negotiate realistic deadlines… or deliver late.
- Scheduling - the science versus the art.
- Overcoming the biggest productivity busters.

Stamp Out Bugs!

- Unhappy truths of ICEs, BDMs, and debuggers.
- *Managing* bugs to get good code fast.
- *Quick* code inspections that keep the schedule on-track.
- Cool ways to find hardware/software glitches.

Managing Real-Time Code

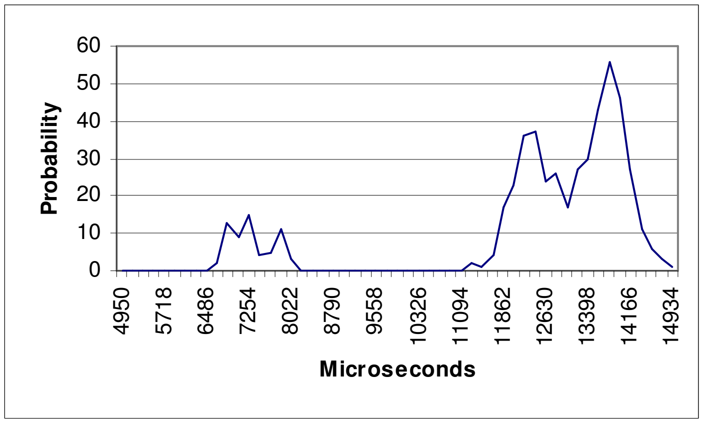
- Design *predictable* real-time code.
- Managing reentrancy
- Troubleshooting and eliminating *erratic crashes*.
- Build better interrupt handlers.

Interfacing to Hardware

- Understanding high-speed signal problems.
- Building peripheral drivers faster.
- Inexpensive performance analyzers

How to Learn from Failures… and Successes

- Embedded disasters, and *what we must learn*.
- Using postmortems to accelerate the product delivery.
- Seven step plan to firmware success.



Do your routines execute in a usec or a week? This function is all over the map, from 6 to 15 msec. You'll learn to write real-time code proactively, finding timing issues early.

Why Take This Course?

Frustrated with schedule slippages? Bugs driving you batty? Product quality sub-par? **Can you afford *not* to take this class?**

We'll teach you how to get your products to market faster with fewer defects. Our recommendations are *practical*, *useful today*, and *tightly focused* on embedded system development. Don't expect to hear another clever but ultimately discarded software methodology. You'll also take home a 150-page handbook with algorithms, ideas and solutions to common embedded problems.

Registration Form on Last Page

Here is what some of our attendees have said:

Thanks for a great seminar. We really enjoyed it! We're already putting the ideas you gave us to use.
J. Sargent, CSC

I like your practical, no nonsense advice backed up with numbers, your dynamic presentation style, and the nice handout that you gave us. I will definitely recommend your seminar to other programmers.
Ed Chehovin, US Navy

I just wanted to say thanks for a great seminar last week. Already the information you gave has proven useful – I used that ISR trick and we finally found an error we've been chasing for months.
Sandeep Miran

Thank you so much for a great class! Now my co-workers think I'm the guru!
Dana Woodring, Northrup Grumman

Did you know that…

… doubling the size of the code results in much more than twice the work? In this seminar you'll learn ways unique to embedded systems to partition your firmware to keep schedules from skyrocketing out of control.

… you can reduce bugs by an order of magnitude before starting debugging? Most firmware starts off with a 5-10% error rate – 500 or more bugs in a little 10k LOC program. Imagine the impact finding all those has on the schedule! Learn simple solutions that don't require revolutionizing the engineering department.

… you can create a predictable real-time design? This class will show you how to measure the system's performance, manage reentrancy, and implement ISRs with the least amount of pain. You'll even study real timing data for common C constructs on various CPUs.

… a 20% reduction in processor loading slashes development time? Learn to keep loading low while simplifying overall system design.

… few watchdog timers are properly implemented? Most are partial solutions to a complex problem. We'll show you how to build an awesome WDT.

… most interrupt-driven timers are improperly coded? Subtle asynchronous issues always lead to erratic timer reads and crashes. The solutions are not obvious, but easy to implement.

… reuse is usually a waste of time? Most companies fail miserably at it. Though promoted as the solution to the software crisis, it's much tougher than advertised. You'll learn the ingredients of successful reuse.

Busy Schedule? . . .

If you can't take the time to travel, we can present this seminar at your facility.

We will train all of your developers and focus on the challenges unique to your products and team.

Thanks for the terrific seminar here at ALSTROM yesterday! It got rave reviews from a pretty tough crowd.

Cheryl Saks, ALSTROM

Thanks for a valuable, pragmatic, and informative lesson in embedded systems design. All the attendees thought it was well worth their time.

Craig DeFilippo, Pitney Bowes

I just wanted to thank you again for the great class last week. With no exceptions, all of the feedback from the participants was extremely positive. We look forward to incorporating many of the suggestions and observations into making our work here more efficient and higher quality.

Carol Batman, INDesign LLC

Contact us for info on how we can bring this seminar to your company

E-mail: info@ganssle.com
or call us at 410-504-6660

What are you doing to upgrade your skills? What are you doing to help your engineers succeed?

Do you consistently produce quality firmware on schedule? *If not . . . what are you doing about it?*

Better Firmware... *Faster!*

A one-day class
April 23, 2008 - Chicago
April 25, 2008 - Denver
May 19, 2008 - London

Spend a day with Jack Ganssle, Embedded System Programming's Technical Editor and columnist, and learn new ways to get your products to market faster, and improve your resume with the 0.7 Continuing Education Unite you'll be awarded.

Registration Information

All of this, plus 150 pages of handouts, for just $695 per person. Plus you will receive a personalized certificate of completion at the end of the course.

Groups of 3 or more registering together pay only $595 each.

Register early and save. Sign up a month in advance, and receive a $50.00 discount.

Fax this form to 647-439-1454. Or, register by phone at 410-504-6660 or via email to register@ganssle.com.

Cancellations made more than 14 days prior to the class are refundable less a $50 fee. Cancellations made within 14 days are non-refundable, but are 100% transferable to all courses we offer.

April 23, 2008
Chicago, IL
Hilton Oak Lawn
9333 Cicero Ave.
Oak Lawn, Chicago

April 25, 2008
Denver, CO
Embassy Suites Hotel
7525 East Hampden Ave.
Denver, CO

May 19, 2008
London, UK
Jurys Inn Heathrow
Eastern Perimeter Rd
Hatton Cross, Hounslow

Registration Form

Today's Date: ____________________

Name: ____

Company: ____

Mailing address: ____

City, State, Zip: ____

Phone: ____________________ Extension:_____ ___

Fax: ____________________

Email: ____________________

Location: Chicago _______Denver ______London_______

Number of attendees: ________

Purchase Order Attached. P.O. Number: _____________

Charge to: Visa MasterCard American Express

Card Number: _________________________ Expires: _______

Name on Card: ___________________________

Signature: ___________________________

Fax this to 647-439-1454. Or, call us at 410-504-6660.