

# Great Watchdogs

Version 1.2, updated January 2004

**Jack G. Ganssle**

jack@ganssle.com

**The Ganssle Group**

PO Box 38346

Baltimore, MD 21231

(410) 504-6660

fax (647) 439-1454

*Jack Ganssle believes that embedded development can be much more efficient than it usually is, and that we can – and must – create more reliable products. He conducts one-day seminars that teach ways to produce better firmware faster. For more information see [www.ganssle.com](http://www.ganssle.com).*

## Building a Great Watchdog

Launched in January of 1994, the Clementine spacecraft spent two very successful months mapping the moon before leaving lunar orbit to head towards near-Earth asteroid Geographos.

A dual-processor Honeywell 1750 system handled telemetry and various spacecraft functions. Though the 1750 could control Clementine's thrusters, it did so only in emergency situations; all routine thruster operations were under ground control.

On May 7, 1994, the 1750 experienced a floating point exception. This wasn't unusual; some 3000 prior exceptions had been detected and handled properly. But immediately after the May 7 event downlinked data started varying wildly and nonsensically. Then the data froze. Controllers spent 20 minutes trying to bring the system back to life by sending software resets to the 1750; all were ignored. A hardware reset command finally brought Clementine back on-line.

Alive, yes, even communicating with the ground, but with virtually no fuel left.

The evidence suggests that the 1750 locked up, probably due to a software crash. While hung the processor turned on one or more thrusters, dumping fuel and setting the spacecraft spinning at 80 RPM. In other words, it appears the code ran wild, firing thrusters it should never have enabled; they kept firing till the tanks ran nearly dry and the hardware reset closed the valves.

The mission to Geographos had to be abandoned.

Designers had worried about this sort of problem and implemented a software thruster timeout. That, of course, failed when the firmware hung.

The 1750's built-in watchdog timer hardware was not used, over the objections of the lead software designer. With no automatic "reset" button, success of the mission rested in the abilities of the controllers on Earth to detect problems quickly and send a hardware reset. For the lack of a few lines of watchdog code the mission was lost.

Though such a fuel dump had never occurred on Clementine before, roughly 16 times before the May 7 event hardware resets from the ground had been required to bring the spacecraft's firmware back to life. One might also wonder why some 3000 previous floating point exceptions were part of the mission's normal firmware profile.

Not surprisingly, the software team wished they had indeed used the watchdog, and had not implemented the thruster timeout in firmware. They also noted, though, that a normal, simple, watchdog may not have been robust enough to catch the failure mode.

Contrast this with Pathfinder, a mission whose software also famously hung, but which was saved by a reliable watchdog. The software team found and fixed the bug, uploading new code to a target system 40 million miles away, enabling an amazing roving scientific mission on Mars.

Watchdog timers (WDTs) are our failsafe, our last line of defense, an option taken only when all else fails - right? These missions (Clementine had been reset *16 times* prior to the failure) and so many others suggest to me that WDTs are not emergency outs, but integral parts of our systems. The WDT is as important as `main()` or the runtime library, it's an asset that is likely to be used, and maybe used a lot.

Outer space is a hostile environment, of course, with high intensity radiation fields, thermal extremes and vibrations we'd never see on Earth. Do we have these worries when designing Earth-bound systems?

Maybe so. Intel revealed that the McKinley processor's ultra fine design rules and huge transistor budget means cosmic rays may flip on-chip bits. The Itanium 2 processor, also sporting an astronomical transistor budget and small geometry, includes an on-board system management unit to handle transient hardware failures. The hardware ain't what it used to be – even if our software were perfect.

But too much (all?) firmware is not perfect. Consider this unfortunately true story from Ed VanderPloeg:

“The world has reached a new embedded software milestone: I had to reboot my hood fan. That's right, the range exhaust fan in the kitchen. It's a simple model from a popular North American company. It has six buttons on the front: 3 for low, medium, and high fan speeds and 3 more for low, medium, and high light levels. Press a button once and the hood fan does what the button says. Press the same button again and the fan or lights turn off. That's it. Nothing fancy. And it needed rebooting via the breaker panel.”

“Apparently the thing has a micro to control the light levels and fan speeds, and it also has a temperature sensor to automatically switch the fan to high speed if the temperature exceeds some fixed threshold. Well, one day we were cooking dinner as usual, steaming a pot of potatoes, and suddenly the fan kicks into high speed and the lights start flashing. "Hmm, flaky sensor or buggy sensor software", I think to myself.”

“The food happened to be done so I turned off the stove and tried to turn off the fan, but I suppose it wanted things to cool off first. Fine. So after ten minutes or so the fan and lights turned off on their own. I then went to turn on the lights, but instead they flashed continuously, with the flash rate depending on the brightness level I selected.”

“So just for fun I tried turning on the fan, but any of the three fan speed buttons produced only high speed. "What 'smart' feature is this?", I wondered to myself. Maybe it needed to rest a while. So I turned off the fan & lights and went back to finish my dinner. For the rest of the evening the fan & lights would turn on and off at random intervals and random levels, so I gave up on the idea that it would self-correct. So with a heavy heart I went over to the breaker panel, flipped the hood fan breaker to & fro, and the hood fan was once again well-behaved.”

“For the next few days, my wife said that I was moping around as if someone had died. I would tell everyone I met, even complete strangers, about what happened: "Hey, know what? I had to reboot my hood fan the other night!". The responses were varied, ranging from "Freak!" to "Sounds like what happened to my toaster...". Fellow programmers would either chuckle or stare in common disbelief.”

“What's the embedded world coming to? Will programmers and companies everywhere realize the cost of their mistakes and clean up their act? Or will the entire world become accustomed to occasionally rebooting everything they own? Would the expensive embedded devices then come with a "reset" button, advertised as a feature? Or will programmer jokes become as common and ruthless as lawyer jokes? I wish I knew the answer. I can only hope for the best, but I fear the worst.”

One developer admitted to me that his consumer products company could care less about the correctness of firmware. Reboot – who cares? Customers are used to this, trained by decades of desktop computer disappointments. Hit the reset switch, cycle power, remove the batteries for 15 minutes, even pre-teens know the tricks of coping with legions of embedded devices.

Crummy firmware is the norm, but in my opinion is totally unacceptable. Shipping a defective product in any other field is like opening the door to torts. So far the embedded world has been mostly immune from predatory lawyers, but that Brigadoon-like isolation is unlikely to continue. Besides, it's simply unethical to produce junk.

But it's hard, even impossible, to produce perfect firmware. We must strive to make the code correct, but also design our systems to cleanly handle failures. In other words, a healthy dose of paranoia leads to better systems.

A Watchdog Timer (WDT) is an important line of defense in making reliable products.

Well-designed watchdog timers fire off a lot, daily and quietly saving systems and lives without the esteem offered to other, human, heroes. Perhaps the developers producing such reliable WDTs deserve a parade. Poorly-designed WDTs fire off a lot, too, sometimes saving things, sometimes making them worse. A simple-minded watchdog implemented in a non-safety critical system won't threaten health or lives, but can result in systems that hang and do strange things that tick off our customers. No business can

tolerate unhappy customers, so unless your code is perfect (whose is?) it's best in all but the most cost-sensitive apps to build a really great WDT.

An effective WDT is far more than a timer that drives reset. Such simplicity might have saved Clementine, but would it fire when the code tumbles into a really weird mode like that experienced by Ed's hood fan?

## Internal WDTs

Internal watchdogs are those that are built into the processor chip. Virtually all highly integrated embedded processors include a wealth of peripherals, often with some sort of watchdog. Most are brain-dead WDTs suitable for only the lowest-end applications.

Let's look at a few. Toshiba's TMP96141AF is part of their TLCS-900 family of quite nice microprocessors, which offers a wide range of extremely versatile on-board peripherals. All have pretty much the same watchdog circuit. As the data sheet says, "The TMP96141AF is containing watchdog timer of Runaway detecting."

Ahem. And I thought the days of Jinglish were over. Anyway, the part generates a non-maskable interrupt when the watchdog times out, which is either a very, very bad idea or a wonderfully clever one. It's clever only if the system produces an NMI, waits a while, and only then asserts reset, which the Toshiba part unhappily cannot do. Reset and NMI are synchronous.

A nice feature is that it takes two different I/O operations to disable the WDT, so there are slim chances of a runaway program turning off this protective feature.

Motorola's widely-used 68332 variant of their CPU32 family (like most of these 68k embedded parts) also includes a watchdog. It's a simple-minded thing meant for low-reliability applications only. Unlike a lot of WDTs, user code must write two different values (0x55 and 0xaa) to the WDT control register to insure the device does not time out. This is a very good thing – it limits the chances of rogue software accidentally issuing the command needed to appease the watchdog. I'm not thrilled with the fact that any amount of time may elapse between the two writes (up to the timeout period). Two back-to-back writes would further reduce the chances of random watchdog tickles, though once would have to insure no interrupt could preempt the paired writes. And the 0x55/0xaa twosome is often used in RAM tests; since the 68k I/O registers are memory mapped, a runaway RAM test could keep the device from resetting.

The 68332's WDT drives reset, not some exception handling interrupt or NMI. This makes a lot of sense, since any software failure that causes the stack pointer to go odd will crash the code, and a further exception-handling interrupt of any sort would drive the part into a "double bus fault". The hardware is such that it takes a reset to exit this condition.

Motorola's popular Coldfire parts are similar. The MCF5204, for instance, will let the code write to the WDT control registers only once. Cool! Crashing code, which might do all sorts of silly things, cannot reprogram the protective mechanism. However, it's possible to change the reset interrupt vector at any time, pretty much invalidating the clever write-once design.

Like the CPU32 parts a 0x55/0xaa sequence keeps the WDT from timing out, and back-to-back writes aren't required. The Coldfire datasheet touts this as an advantage since it can handle interrupts between the two tickle instructions, but I'd prefer less of a window. The Coldfire has a fault-on-fault condition much like the CPU32's double bus fault, so reset is also the only option when WDT fires – which is a good thing.

There's no external indication that the WDT timed out, perhaps to save pins. That means your hardware/software must be designed so at a warm boot the code can issue a from-the-ground-up reset to every peripheral to clear weird modes that may accompany a WDT timeout.

Philip's XA processors require two sequential writes of 0xa5 and 0x5a to the WDT. But like the Coldfire there's no external indication of a timeout, and it appears the watchdog reset isn't even a complete CPU restart – the docs suggest it's just a reload of the program counter. Yikes – what if the processor's internal states were in disarray from code running amok or a hardware glitch?

Dallas Semiconductor's DS80C320, an 8051 variant, has a very powerful WDT circuit that generates a special watchdog interrupt 128 cycles before automatically – and irrevocably – performing a hardware reset. This gives your code a chance to save the system, and leave debugging breadcrumbs behind before a complete system restart begins. Pretty cool.

**Summary: What's Wrong With Many Internal WDTs:**

- A watchdog timeout must assert a hardware reset to guarantee the processor comes back to life. Reloading the program counter may not properly reinitialize the CPU's internals.
- WDTs that issue NMI without a reset may not properly reset a crashed system.
- A WDT that takes a simple toggle of an I/O line isn't very safe
- When a pair of tickles uses common values like 0x55 and 0xaa, other routines – like a RAM test – may accidentally service the WDT.
- Watch out for WDTs whose control registers can be reprogrammed as the system runs; crashed code could disable the watchdog.
- If a WDT timeout does not assert a pin on the processor, you'll have to add hardware to reset every peripheral after a timeout. Otherwise, though the CPU is back to normal, a confused I/O device may keep the system from running properly.

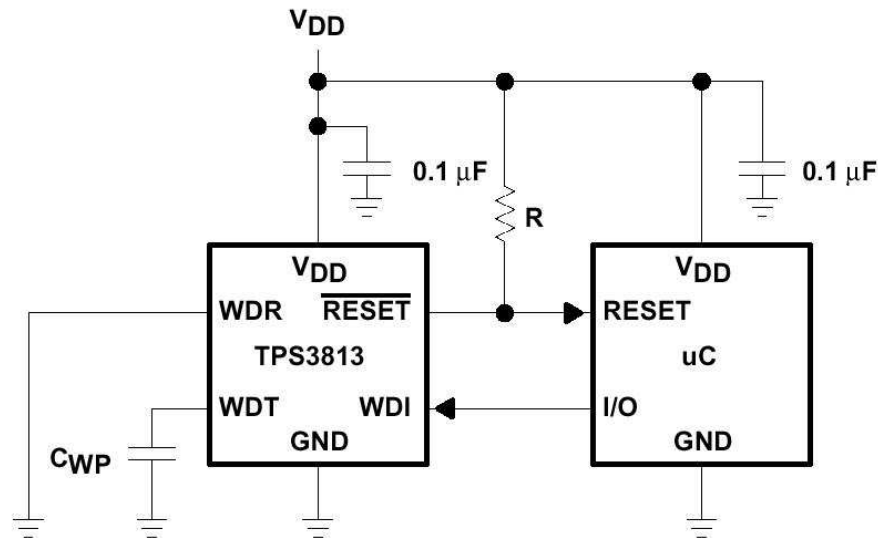
## External WDTs

Many of the supervisory chips we buy to manage a processor's reset line include built-in WDTs.

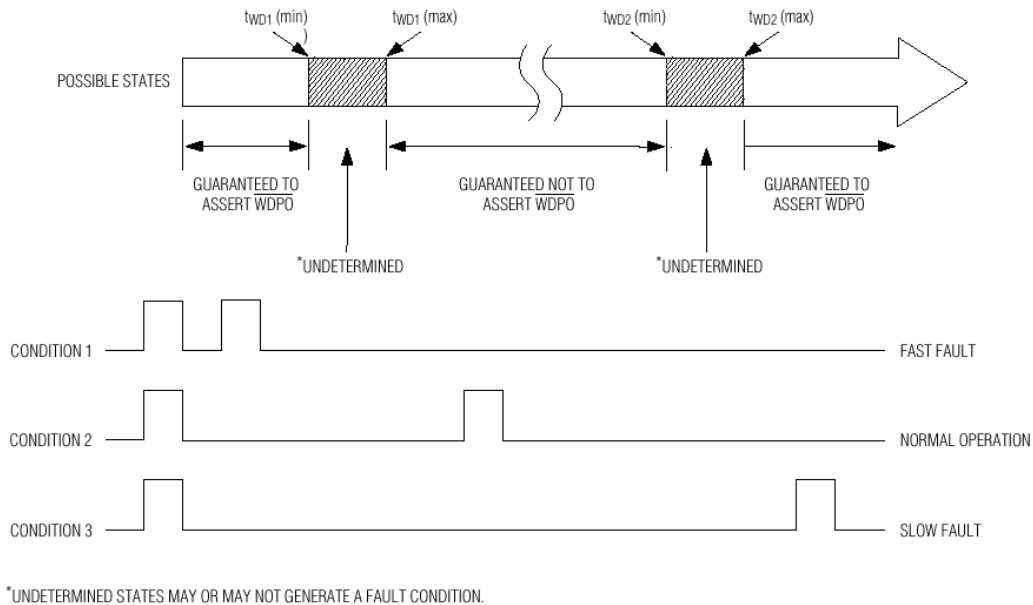
TI's UCC3946 is one of many nice power supervisor parts that does an excellent job of driving reset only when  $V_{CC}$  is legal. In a nice small 8 pin SMT package it eats practically no PCB real estate. It's not connected to the CPU's clock, so the WDT will output a reset to the hardware safe-ing mechanisms even if there's a crystal failure. But it's too darn simple: to avoid a timeout just wiggle the input bit once in a while. Crashed code could do this in any of a million ways.

TI isn't the only purveyor of simplistic WDTs. Maxim's MAX823 and many other versions are similar. The catalogs of a dozen other vendors list equally dull and ineffective watchdogs.

But both TI and Maxim do offer more sophisticated devices. Consider TI's TPS3813 and Maxim's MAX6323. Both are "Window Watchdogs". Unlike the internal versions described above that avoid timeouts using two different data writes (like a 0x55 and then 0xaa), these require tickling within certain time bands. Toggle the WDT input too slowly, too fast, or not at all, and a timeout will occur. That greatly reduces the chances that a program run amok will create the precise timing needed to satisfy the watchdog. Since a crashed program will likely speed up or bog down if it does anything at all, errant strobing of the tickle bit will almost certainly be outside the time band required.



*TI's TPS3813 is easy to use and offers a nice windowing WDT feature.*



*Window timing of Maxim's equally cool MAX6323.*

## Characteristics of Great WDTs

What's the rationale behind an awesome watchdog timer? The perfect WDT should detect all erratic and insane software modes. It must not make any assumptions about the condition of the software *or* the hardware; in the real world anything that can go wrong will. It must bring the system back to normal operation no matter what went wrong, whether from a software defect, RAM glitch, or bit flip from cosmic rays.

It's impossible to recover from a hardware failure that keeps the computer from running properly, but at the least the WDT must put the system into a safe state. Finally, it should leave breadcrumbs behind, generating debug information for the developers. After all, a watchdog timeout is the yin and yang of an embedded system. It saves the system, keeping customers happy, yet demonstrates an inherent design flaw that should be addressed. Without debug info, troubleshooting these infrequent and erratic events is close to impossible.

What does this mean in practice?

An effective watchdog is independent from the main system. Though all WDTs are a blend of interacting hardware and software, something external to the processor must always be poised, like the sword of Damocles, ready to intervene as soon as a crash occurs. Pure software implementations are simply not reliable.

There's only one kind of intervention that's effective: an immediate reset to the processor and all connected peripherals. Many embedded systems have a watchdog that initiates a non-maskable interrupt. Designers figure that firing off NMI rather than reset preserves

some of the system's context. It's easy to seed debugging assets in the NMI handler (like a stack capture) to aid in resolving the crash's root cause. That's a great idea, except that it does not work.

All we really know when the WDT fires is that something truly awful happened. Software bug? Perhaps. Hardware glitch? Also possible. Can you insure that the error wasn't something that totally scrambled the processor's internal logic states? I worked with one system where a motor in another room induced so much EMF that our instrument sometimes went bonkers. We tracked this down to a sub-nanosecond glitch on one CPU input, a glitch so short that the processor went into an undocumented weird mode. Only a reset brought it back to life.

Some CPUs, notably the 68k and ColdFire, will throw an exception if a software crash causes the stack pointer to go odd. That's not bad... except that any watchdog circuit that then drives the CPU's non-maskable interrupt will unavoidably invoke code that pushes the system's context, creating a second stack fault. The CPU halts, staying halted till a reset, and only a reset, comes along.

Drive reset; it's the only reliable way to bring a confused microprocessor back to lucidity. Some clever designers, though, build circuits that drive NMI first, and then after a short delay pound on reset. If the NMI works then its exception handler can log debug information and then halt. It may also signal other connected devices that this unit is going offline for a while. The pending reset guarantees an utterly clean restart of the code. Don't be tempted to use the NMI handler to safe dangerous hardware; that task always, in every system, belongs to a circuit external to the possibly confused CPU.

Don't forget to reset the whole computer system; a simple CPU restart may not be enough. Are the peripherals absolutely, positively, in a sane mode? Maybe not. Runaway code may have issued all sorts of I/O instructions that placed complex devices in insane modes. Give every peripheral a *hardware* reset; software resets may get lost in all of the I/O chatter.

Consider what the system must do to be totally safe after a failure. Maybe a pacemaker needs to reboot in a heartbeat (so to speak)... or maybe backup hardware should issue a few ticks if reboots are slow.

One thickness gauge that beams high energy gamma rays through 4 inches of hot steel failed in a spectacular way. Defective hardware crashed the code. The WDT properly closed the protective lead shutter, blocking off the 5 curie cesium source. I was present, and watched incredulously as the engineering VP put his head in path of the beam; the crashed code, still executing something, tricked the watchdog into opening the shutter, beaming high intensity radiation through the veep's forehead. I wonder to this day what eventually became of the man.

A really effective watchdog cannot use the CPU's clock, which may fail. A bad solder joint on the crystal, poor design that doesn't work well over temperature extremes, or

numerous other problems can shut down the oscillator. This suggests that no WDT internal to the CPU is really safe. Most share the processor's clock.

Under no circumstances should the software be able to reprogram the WDT or any of its necessary components (like reset vectors, I/O pins used by the watchdog, etc). Assume runaway code runs under the guidance of a malevolent deity.

Build a watchdog that monitors the *entire system's* operation. Don't assume that things are fine just because some loop or ISR runs often enough to tickle the WDT. A software-only watchdog should look at a variety of parameters to insure the product is healthy, kicking the dog only if everything is OK. What is a software crash, after all? Occasionally the system executes a HALT and stops, but more often the code vectors off to a random location, continuing to run instructions. Maybe only one task crashed. Perhaps only one is still alive – no doubt that which kicks the dog.

Think about what can go wrong in your system. Take corrective action when that's possible, but initiate a reset when it's not. For instance, can your system recover from exceptions like floating point overflows or divides by zero? If not, these conditions may well signal the early stages of a crash. Either handle these competently or initiate a WDT timeout. For the cost of a handful of lines of code you may keep a 60 Minutes camera crew from appearing at your door.

It's a good idea to flash an LED or otherwise indicate that the WDT kicked. A lot of devices automatically recover from timeouts; they quickly come back to life with the customer totally unaware a crash occurred. Unless you have a debug LED how do you know if your precious creation is working properly, or occasionally invisibly resetting? One outfit complained that over time, and with several thousand units in the field, their product's response time to user inputs degraded noticeably. A bit of research showed that their system's watchdog properly drove the CPU's reset signal, and the code then recognized a warm boot, going directly to the application with no indication to the users that the time-out had occurred. We tracked the problem down to a floating input on the CPU, that caused the software to crash - up to several thousand times per second. The processor was spending most of its time resetting, leading to apparently slow user response. An LED would have shown the problem during debug, long before customers started yelling.

Everyone knows we should include a jumper to disable the WDT during debugging. But few folks think this through. The jumper should be inserted to enable debugging, and removed for normal operation. Otherwise if manufacturing forgets to install the jumper, or if it falls out during shipment, the WDT won't function. And there's no production test to check the watchdog's operation.

Design the logic so the jumper disconnects the WDT from the reset line (possibly though an inverter so an inserted jumper sets debug mode). Then the watchdog continues to function even while debugging the system. It won't reset the processor but will flash the

LED. The light will blink a lot when breakpointing and singlestepping, but should never come on during full-speed testing.

#### **Characteristics of Great WDTs:**

- Make no assumptions about the state of the system after a WDT reset; hardware and software may be confused.
- Have *hardware* put the system into a safe state.
- Issue a hardware reset on timeout.
- Reset the peripherals as well.
- Insure a rogue program cannot reprogram WDT control registers.
- Leave debugging breadcrumbs behind.
- *Insert* a jumper to disable the WDT for debugging; *remove* it for production units.

## **Using an Internal WDT**

Most embedded processors that include high integration peripherals have some sort of built-in WDT. Avoid these except in the most cost-sensitive or benign systems. Internal units offer minimal protection from rogue code. Runaway software may reprogram the WDT controller, many internal watchdogs will not generate a proper reset, and any failure of the processor will make it impossible to put the hardware into a safe state. A great WDT must be independent of the CPU it's trying to protect.

However, in systems that really must use the internal versions, there's plenty we can do to make them more reliable. The conventional model of kicking a simple timer at erratic intervals is too easily spoofed by runaway code.

A pair of design rules leads to decent WDTs: kick the dog only after your code has done *several unrelated* good things, and make sure that erratic execution streams that wander into your watchdog routine won't issue incorrect tickles.

This is a great place to use a simple state machine. Suppose we define a global variable named "state". At the beginning of the main loop set `state` to 0x5555. Call watchdog routine A, which adds an offset – say 0x1111 – to `state` and then insures the variable is now 0x6666. Return if the compare matches; otherwise halt or take other action that will cause the WDT to fire.

Later, maybe at the end of the main loop, add another offset to `state`, say 0x2222. Call watchdog routine B, which makes sure `state` is now 0x8888. Set `state` to zero. Kick the dog if the compare worked. Return. Halt otherwise.

Suppose the code crashes and for inscrutable reasons probably having to do with Murphy's Law and the perversity of nature vectors into `wdt_b()` just before the `kick_dog` command. The protection mechanism of the state machine won't help.

Perhaps it's safe to assume that the code will again crash when `wdt_b()` returns, so the system will miss the next watchdog tickle. But... perhaps not – who knows what evil lurks in the mind of runaway software?

Is this fear paranoid? You bet. But the WDT might be the last line of defense between deflecting the Earth-bound asteroid and utter disaster, or at least in rebooting the pacemaker before grandpa collapses. Assuming that crashed code will operate in any benign mode is naïve.

So `wdt_b()` double-checks variable “state” to insure the system halts (so the watchdog can issue a reset) even if rogue code wandered into `wdt_b()` just before issuing the `kick_dog`.

This is a trivial bit of code, but now runaway code that stumbles into any of the tickling routines cannot errantly kick the dog. Further, no tickles will occur unless the entire main loop executes in the proper sequence. If the code just calls routine B repeatedly, no tickles will occur because it sets `state` to zero before exiting.

Add additional intermediate states as your fear of litigation dictates.

Normally I detest global variables, but this is a perfect application. Cruddy code that mucks with the variable, errant tasks doing strange things, or any error that steps on the global will make the WDT timeout.

Do put these actions in the program's main loop, not inside an ISR. It's fun to watch a multitasking product crash – the entire system might be hung, but one task still responds to interrupts. If your watchdog tickler stays alive as the world collapses around the rest of the code, then the watchdog serves no useful purpose.

If the WDT doesn't generate an external reset pulse (some processors handle the restart internally) make sure the code issues a hardware reset to all peripherals immediately after start-up. That may mean working with the EEs so an output bit resets every resettable peripheral.

If you must take action to safe dangerous hardware, well, since there's no way to guarantee the code will come back to life, stay away from internal watchdogs. Broken hardware will obviously cause this... but so can lousy code. A digital camera was recalled recently when users found that turning the device off when in a certain mode meant it could never be turned on again. The code wrote faulty info to flash memory that created a permanent crash.

```
main(){
    state=0x5555;
    wdt_a();
    .
    .
    .
    state+=0x2222;
    wdt_b();
}
```

```
wdt_a(){
    if (state!= 0x5555) halt;
    state+=0x1111;
}
```

```
wdt_b(){
    if (state!= 0x8888) halt;
    kick dog;
    if (state!= 0x8888) halt;
    state=0;
}
```

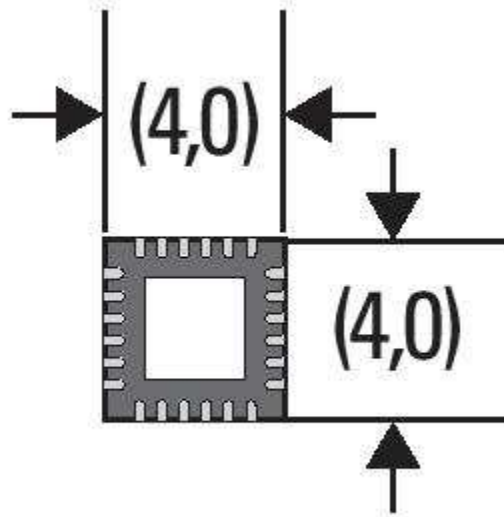
*Pseudocode for handling an internal WDT*

## **An External WDT**

The best watchdog is one that doesn't rely on the processor or its software. It's external to the CPU, shares no resources, and is utterly simple, thus devoid of latent defects.

Use a PIC, a Z8, or other similar dirt-cheap processor as a system health monitor. These parts have an independent clock, on-board memory, and the built-in timers we need to build a truly great WDT. Being external, you can connect an output to hardware interlocks that put dangerous machinery into safe states.

But when selecting a watchdog CPU check the part's specs carefully. Tying the tickle to the watchdog CPU's interrupt input, for instance, may not work reliably. A slow part – like most PICs – may not respond to a tickle of short duration. Consider TI's MSP430 family or processors. They're a very inexpensive (half a buck or so) series of 16 bit processors that use virtually no power and no PCB real estate.

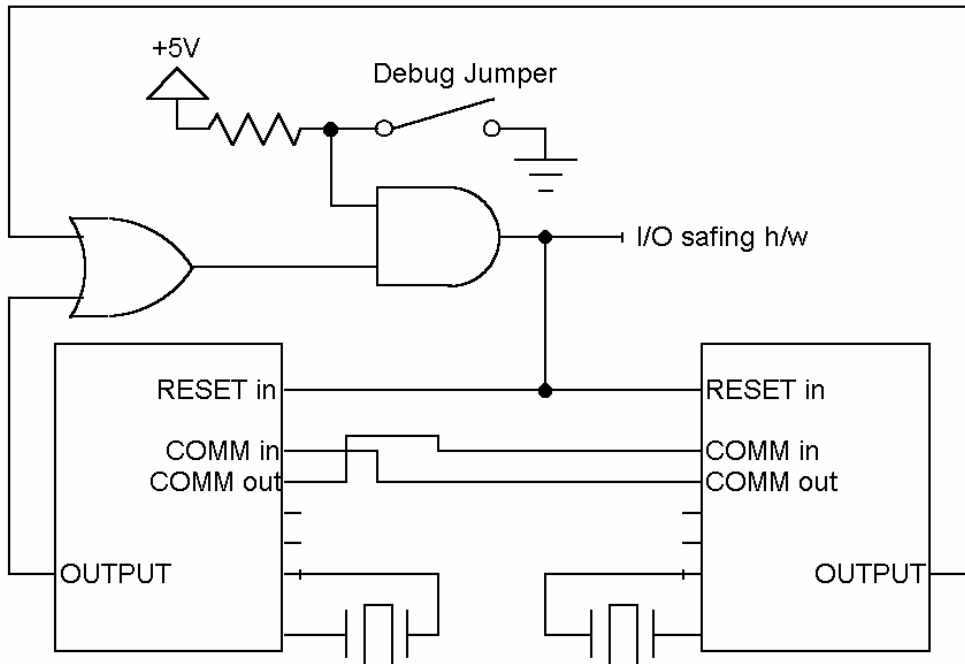


*The MSP430, a 16 bit CPU- that uses no PCB real estate. For metrically-challenged readers, this is about 5/32" x 5/32".*

Tickle it using the same sort of state-machine described above. Like the windowed watchdogs (TI's TPS3813 and Maxim's MAX6323), define min *and* max tickle intervals, to further limit the chances that a runaway program deludes the WDT into avoiding a reset.

Perhaps it seems extreme to add an entire computer just for the sake of a decent watchdog. We'd be fools to add extra hardware to a highly cost-constrained product. Most of us, though, build lower volume higher margin systems. A fifty cent part that prevents the loss of an expensive mission, or that even saves the cost of one customer support call, might make a lot of sense.

In a multiprocessor system it's easy to turn all of the processors into watchdogs. Have them exchange "I'm OK" messages periodically. The receiver resets the transmitter if it stops speaking. This approach checks a lot of hardware and software, and requires little circuitry.



*Watchdog for a dual-processor system – each CPU watches the other.*

## WDTs for Multitasking

Tasking turns a linear bit of software into a multidimensional mix of tasks competing for processor time. Each runs more or less independently of the others... which means each can crash on its own, without bringing the entire system to its knees.

You can learn a lot about a system's design just by observing its operation. Consider a simple instrument with a display and various buttons. Press a button and hold it down; if the display continues to update, odds are the system multitasks.

Yet in the same system a software crash might go undetected by conventional watchdog strategies. If the display or keyboard tasks die, the main line code or a WDT task may continue to run.

Any system that uses an ISR or a special task to tickle the watchdog but that does not examine the health of all other tasks is not robust. Success lies in weaving the watchdog into the fabric of all of the system's tasks - which is happily much easier than it sounds.

First, build a watchdog task. It's the only part of the software allowed to tickle the WDT. If your system has an MMU mask off all I/O accesses to the WDT except those from this task, so rogue code traps on an errant attempt to output to the watchdog.

Next, create a data structure that has one entry per task, with each entry being just an integer.

When a task starts it increments its entry in the structure. Tasks that only start once and stay active forever can increment the appropriate value each time through their main loops.

Increment the data *atomically* – in a way that cannot be interrupted with the data half-changed. ++TASK<sub>i</sub> (if TASK is an integer array) on an 8 bit CPU might not be atomic, though it's almost certainly OK on a 16 or 32 biter. The safest way to both encapsulate and insure atomic access to the data structure is to hide it behind another task. Use a semaphore to eliminate concurrent shared accesses. Send increment messages to the task, using the RTOS's messaging resources.

As the program runs the number of counts for each task advances. Infrequently but at regular intervals the watchdog task runs. Perhaps once a second, or maybe once a msec – it's all a function of your paranoia and the implications of a failure.

The watchdog task scans the structure, checking that the count stored for each task is reasonable. One that runs often should have a high count; another which executes infrequently will produce a smaller value. Part of the trick is determining what's reasonable for each task; stick with me - we'll look at that shortly.

If the counts are unreasonable, halt and let the watchdog timeout. If everything is OK, set all of the counts to zero and exit.

Why is this robust? Obviously, the watchdog monitors every task in the system. But it's also impossible for code that's running amok to stumble into the WDT task and errantly tickle the dog; by zeroing the array we guarantee it's in a "bad" state.

I skipped over a critical step – how do we decide what's a reasonable count for each task? It might be possible to determine this analytically. If the WDT task runs once a second, and one of the monitored tasks starts every 50 msec, then surely a count of around 20 is reasonable.

Other activities are much harder to ascertain. What about a task that responds to asynchronous inputs from other computers, say data packets that come at irregular intervals? Even in cases of periodic events, if these drive a low-priority task they maybe suspended for rather long intervals by higher-priority problems.

The solution is to broaden the data structure that maintains count information. Add min and max fields to each entry. Each task must run at least min, but no more than max times.

Now redesign the watchdog task to run in one of two modes. The first is the one already described, and is used during normal system operation.

The second mode is a debug environment enabled by a compile-time switch that collects min and max data. Each time the WDT task runs it looks at the incremented counts and sets new min and max values as needed. It tickles the watchdog each time it executes.

Run the product's full test suite with this mode enabled. Maybe the system needs to operate for a day or a week to get a decent profile of the min/max values. When you're satisfied that the tests are representative of the system's real operation, manually examine the collected data and adjust the parameters as seems necessary to give adequate margins to the data.

What a pain! But by taking this step you'll get a great watchdog – and a deep look into your system's timing. I've observed that few developers have much sense how their creations perform in the time domain. "It seems to work" tells us little. Looking at the data acquired by this profiling, though might tell a lot. Is it a surprise that task A runs 400 times a second? That might explain a previously-unknown performance bottleneck.

In a real time system we must manage and measure time; it's every bit as important as procedural issues, yet is oft ignored till a nagging problem turns into an unacceptable symptom. This watchdog scheme forces you to think in the time domain, and by its nature profiles – admittedly with coarse granularity – the time-operation of your system.

There's yet one more kink, though. Some tasks run so infrequently or erratically that any sort of automated profiling will fail. A watchdog that runs once a second will miss tasks that start only hourly. It's not unreasonable to exclude these from watchdog monitoring. Or, we can add a bit of complexity to the code to initiate a watchdog timeout if, say, the slow tasks don't start even after a number of hours elapse.

## Summary and Other Thoughts

I remain troubled by the fan failure described earlier. It's easy to dismiss this as a glitch, an unexplained failure caused by a hardware or software bug, cosmic rays, or meddling by aliens. But others have written about identical situations with their vent fans, all apparently made by the same vendor.

When we blow off a failure, calling it a "glitch" as if that name explains something, we're basically professing our ignorance. There are no glitches in our macroscopically deterministic world. Things happen for a reason.

The fan failures didn't make the evening news and hurt no one. So why worry?

Surely the customers were irritated, and the possible future sales of that company at least somewhat diminished. The company escalated the general rudeness level of the world, and thus the sum total incipient anger level, by treating their customers with contempt. Maybe a couple more Valiums were popped, a few spouses yelled at, some kids cowered till dad calmed down. In the grand scheme of things perhaps these are insignificant blips.

Yet we must remember the purpose of embedded control is to help people, to improve lives, not to help therapists garner new patients.

What concerns me is that if we cannot even build reliable fan controllers, what hope is there for more mission-critical applications?

I don't know what went wrong with those fan controllers, and I have no idea if a WDT – well designed or not – is part of the system. I do know, though, that the failures are unacceptable and avoidable. But maybe not avoidable by the use of a conventional watchdog. A WDT tells us the code is running. A windowing WDT tells us it's running with pretty much the right timing. No watchdog, though, flags software executing with corrupt data structures, unless the data is so bad it grossly affects the execution stream.

Why would a data structure become corrupt? Bugs, surely. Strange conditions the designers never anticipated will also create problems, like the never-ending flood of buffer overflow conditions that plague the net, or unexpected user inputs (“we never thought the user would press all 4 buttons at the same time!”).

Is another layer of self-defense, beyond watchdogs, wise? Safety critical apps, where the cost of a failure is frighteningly high, should definitely include integrity checks on the data. Low threat equipment – like this oven fan – can and should have at least a minimal amount of code for trapping possible failure conditions.

Some might argue it makes no sense to “waste” time writing defensive code for a dumb fan application. Yet the simpler the system, the easier and quicker it is to plug in a bit of code to look for program and data errors.

Very simple systems tend to translate inputs to outputs. Their primary data structures are the I/O ports. Often several unrelated output bits get multiplexed to a single port. To change one bit means either reading the port's current status, or maintaining a copy of the port in RAM. Both approaches are problematic.

Computers are deterministic, so it's reasonable to expect that, in the absence of bugs, they'll produce correct results all the time. So it's apparently safe to read a port's current status, AND off the unwanted bits, OR in new ones, and output the result. This is a state machine; the outputs evolve over time to deal with changing inputs. But the process works only if the state machine never incorrectly flips a bit. Unfortunately, output ports are connected to the hostile environment of the real world. It's entirely possible that a bit of energy from starting the fan's highly inductive motor will alter the port's setting. I've seen this happen many times.

So maybe it's more reliable to maintain a memory image of the port. The downside is that a program bug might corrupt the image. Most of the time these are stored as global variables, so any bit of sloppy code can accidentally trash the location. Encapsulation solves that problem, but not the one of a wandering pointer walking over the data, or of a latent reentrancy issue corrupting things. You might argue that writing correct code

means we shouldn't worry about a location changing, but we added a WDT to, in part, deal with bugs. Similar concerns about our data are warranted.

In a simple system look for a design that resets data structures from time to time. In the case of the oven fan, whenever the user selects a fan speed reset all I/O ports and data structures. It's that simple.

In a more complicated system the best approach is the oldest trick in software engineering: check the parameters passed to functions for reasonableness. In the embedded world we chose not to do this for three reasons: speed, memory costs, and laziness. Of these, the third reason is the real culprit most of the time.

Cycling power is the oldest fix in the book; it usually means there's a lurking bug and a poor WDT implementation. Embedded developer Peter Putnam wrote:

Last November, I was sitting in one of a major airline's newer 737-900 aircraft on the ramp in Cancun, Mexico, waiting for departure when the pilot announced there would be a delay due to a computer problem. About twenty minutes later a group of maintenance personnel arrived. They poked around for a bit, apparently to no avail, as the captain made another announcement. "Ladies and Gentlemen," he said, "we're unable to solve the problem, so we're going to try turning off all aircraft power for thirty seconds and see if that fixes it."

Sure enough, after rebooting the Boeing 737, the captain announced that "All systems are up and running properly."

Nobody saw fit to leave the aircraft at that point, but I certainly considered it.

# Better Firmware... *Faster!*

## A One-Day Seminar

Presented at

## Your Company

Does your schedule prevent you from traveling?

*This doesn't mean you have to pass this great opportunity by.*

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

More information at [www.ganssle.com](http://www.ganssle.com)

The Ganssle Group  
PO Box 38346  
Baltimore, MD 21231  
(410) 504-6660  
fax: (647) 439-1454  
[info@ganssle.com](mailto:info@ganssle.com)  
[www.ganssle.com](http://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. *Contact us to learn how we can award each of the attendees 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 was this year's keynote speaker at the Embedded Systems Conference. 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

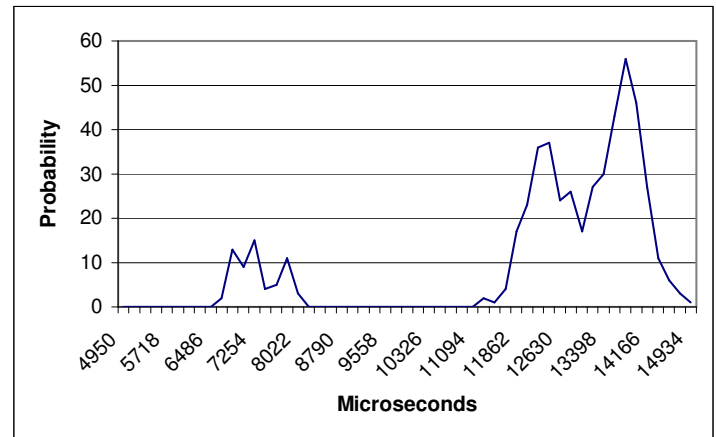
- Design *predictable* real-time code.
- Managing reentrancy.
- Troubleshooting and eliminating *erratic crashes*.
- Build better interrupt handlers.

### 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 those C/C++ runtime routines execute in a usec or a week? This trig function is all over the map, from 6 to 15 msec. You'll learn to rewrite real-time code proactively, anticipation timing issues before debugging.*

## 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.

***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.***

**Here is what some of our attendees have said:**

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 Bateman, INDesign LLC***

Here are just a few of the companies where Jack has presented this seminar:

**Sony-Ericsson, Northrup Grumman, Dell, Western Digital, Bayer, Seagate, Whirlpool, Cutler Hammer, Symbol, Visteon, Honeywell, Kodak and Western Digital.**

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

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?*

**Contact us** for info on how we can bring this seminar to your company.  
**e-mail:** [info@ganssle.com](mailto:info@ganssle.com) or call us at 410-504-6660.