

Guidelines for Creating Robust Embedded Systems

Part 7 – Creating Robust Watchdog Timers

By Bob Japenga

Scope

This is Part 7 of a white paper intended to provide a brief overview of lessons learned in creating robust systems over the past 35 years of embedded systems development (starting with the Intel 4040). This part will address the necessity and method of creating robust watchdog timers.

Introduction

Although we have learned a lot over the years creating robust watchdog timers in order to create robust embedded systems, Jack Ganssle has, in my opinion, written the best white paper on the subject. This white paper will reference Jack but for a more expansive discussion on creating robust watchdogs, read his white paper entitled "[Great Watchdogs.](#)" Niall Murphy also has a very nice article on watchdogs entitled "[Watchdog Timers.](#)"

Let me define the terms I will be using in this paper:

- Watchdog timer - There are a number of definitions for a watchdog timer used in industry. For our purposes we are defining it as follows: A watchdog timer is a combination of both hardware and software that contains a timing device and software to service the timing device. The timing device triggers a system reset or places the system in some fail safe or recovery mode (for this paper we will call this "reset the system.") if the system, due to some fault condition in hardware or software (such as a hang, a resource deadlock, a lock-up or an endless loop in software) fails to perform its specified functionality. The timing device is serviced by software on a regular basis based on indications it obtains from the system of proper operation.
- Tickle the Watchdog timer – this is when the software tells the watchdog timer to restart its timer.
- Service the Watchdog timer – same as "tickle" the watchdog timer.
- Watchdog timer timeout – this is when the watchdog timer has timed out and reset the system. This phrase can be used in conjunction with a "too early" tickle as well as a "too late" tickle even though technically "too early" is not a timeout.
- Watchdog timer timeout period – this is the required for the watchdog timer to time out and reset the system

Why do robust systems need a Watchdog Timer?

Probably the best answer to this question is a paraphrase of a crude axiom: “Problems happen.” Our software fails, our software has flaws, our hardware hiccups, our hardware has race conditions, our software has race conditions, our systems experience a cosmic ray hit that alters memory locations, and much worse. Jack makes the point that watchdog timers are not “emergency outs.” They need to be essential parts of our imperfect software.¹ We couldn’t agree more. One of our core values² at MicroTools is the recognition that “imperfection is the rule.” We have to admit to ourselves that we are hopelessly flawed.

To handle our hopelessly flawed software, the best solution is often a restart. Jack tells a great story about flying on a 737 one day that was experiencing problems before take-off. They solved the problems by restarting everything. And we are not shocked!!! Thanks to PC’s, the reboot and Control-Alt-Delete has become part of the common vernacular. We have found that starting everything all over fixes things. In the old days it was slap it on the side of the box – today’s intermittent failures are often fixed by rebooting. (most of the younger folk here at MicroTools did not even know what we meant by slapping the side of the box!).

The literature is full of examples of expensive systems that failed because they either did not have a watchdog timer or they did not have a robust one.³

Required Degrees of Robustness

These guidelines contain “must haves” and “should haves.” “Must haves” are those guidelines that we believe must be in all systems purporting to be robust. “Should haves” are optional depending upon the degree of robustness required. Systems that are installed in places that have technical support on-site need not be as robust as systems which have no technical support present. Systems that are not accessible by humans need to be more robust than systems that are accessible. It is up to us as system designers to make those tradeoffs. What is the cost (for service; for reputation; for future sales) should the system hang? Remember that good engineering is doing for one buck what any dang fool can do for ten bucks.

What is needed in the hardware portion of a robust Watchdog Timer?

1. The hardware watchdog must be independent of everything else in the system (except power). It should not rely upon anything else in the system. It should not share the clock or the processor (that includes internal watchdogs built into many microprocessors.⁴)
2. The hardware watchdog must completely distrust the software. If the software can do anything to cripple the hardware watchdog it will. If it can do anything to change the time of the watchdog timer timeout period – it will. This is really just re-stating item 1.

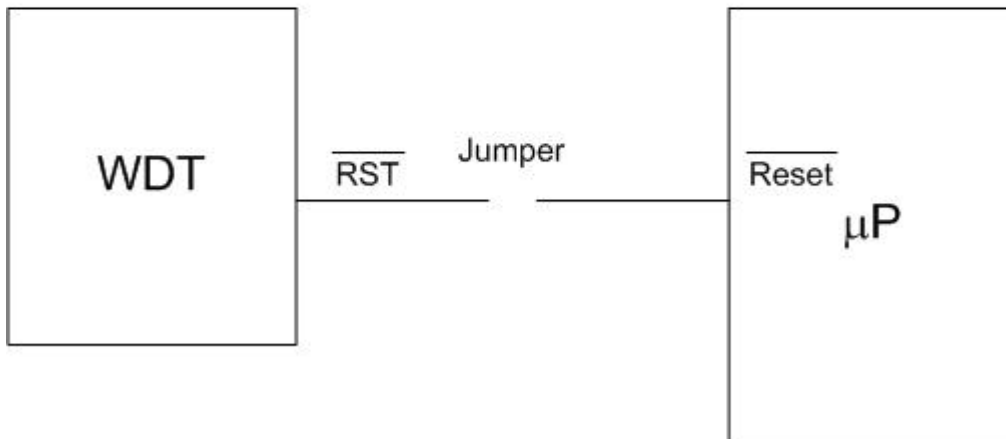
3. The hardware watchdog must have a watchdog timer timeout appropriate for system requirements. For example, if the software can malfunction for several minutes without harm, there is no need to have a very low watchdog timer timeout. In addition, very often system designers either cannot tickle the watchdog for a period of time at startup (for example when we use GreenHills Integrity RTOS with proprietary startup code, we don't have access for almost 500 ms after power on) or they choose not to modify their open source OS startup code to tickle the watchdog. The ideal solution is to choose a hardware watchdog has a long delay at startup. For example, on the Maxim web site, in their parametric search of hardware watchdog timers, there is a column for Watchdog Startup Delay.
4. The hardware watchdog must not be reset by a level but by an edge. Most off-the-shelf devices do this.
5. The hardware watchdog must drive an external reset so that external devices (assuming there are external devices) can be reset as well as the microprocessor. One of the watchdogs built into one of the ARM9 core's has a great watchdog that resets the processor but provides no way to reset external devices. Watch for this when you are selecting an internal watchdog. This is unacceptable and why we don't use that watchdog in any of our products. Jack makes a major point in his paper about how terrible most of the built-in watchdog timers are.
6. The hardware watchdog should not have a mechanism that allows the software to turn it off.⁵
7. The hardware watchdog should have a window of both a maximum **and** a minimum time period. This assures that your system is acting the way you designed it to. Should one of your clocks be running too fast and you start tickling the watchdog earlier than you expected, the watchdog will "timeout." The Texas Instrument TPS3813 and the Maxim 6323 both provide this kind of feature. This "should" needs to be traded off against keeping the software from becoming too complicated. If your software has hard deadlines that "must" be met and they are "windowed," then this "should" becomes a "must" even in face of the added complexity it adds.
8. The hardware watchdog should not be required to be enabled at startup.⁶
9. The hardware watchdog should provide a mechanism to indicate that it has reset. Having this feature allows the software to make intelligent decisions following a watchdog timer timeout. We do not know of a stand-alone chip that provides this. Certain microcontrollers (TMS-470 – an ARM7 core - has a System Reset Exception Status Register⁷) have it built in – but many do not. A software mechanism (albeit imperfect) can be provided whereby a 32 or 64 bit fence (some complicated combination of 1's and 0's like 0xAA5500FF) is written in RAM that is read prior to RAM initialization. On initialization, if this fence is present, the system has gone through a watchdog timer reset. A hardware solution could also be provided whereby a software readable flip-flop is cleared on POR and set on a Watchdog timer reset.
10. The hardware watchdog should have a way to be disabled with a hardware jumper (not enabled with a jumper). The software should have a way to detect

that this jumper is present. Figure 1 shows the traditional way of enabling the watchdog with a jumper. Figure 2 shows one possible way of disabling the watchdog with a jumper as well as providing a means to detect the jumper. If your system has not implemented a means to detect if the watchdog is enabled, as a minimum this should be tested as part of the manufacturing board level or system check. We do this on all such systems. Alternatively, some systems can periodically test the operation of the watchdog timer to verify proper operation and verify the presence of the jumper. Too often we have trouble shot systems that are hung and wonder if manufacturing forgot to put the jumper on.

11. The hardware watchdog should have the ability to have a longer timeout at startup since it is often difficult to tickle the watchdog early. Many stand-alone devices provide up to 60 seconds watchdog timer delays at power up.
12. The hardware watchdog should have a broad programmable range of timeouts. Some systems require very quick fault detection and reset. Most can live with relatively long timeouts. The software designer needs to determine what this timeout should be.

Figure 1

Traditional Watchdog Disable

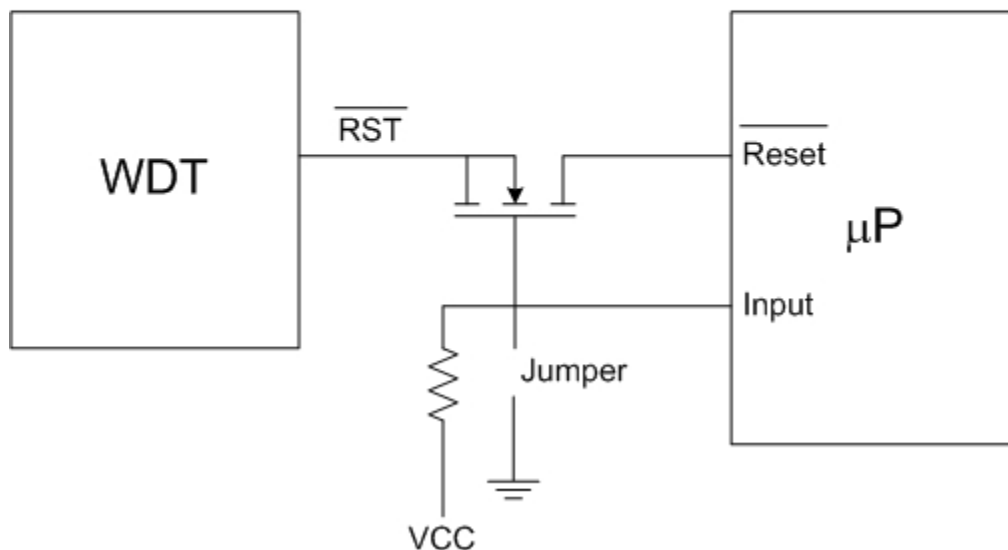


13. The hardware watchdog should have a mechanism for the software to know that it is operational. I am not aware of any devices that do this which necessitates the kind of built in testing described above. If you know of any such hardware devices, please let us know.
14. The hardware watchdog could consist of multiple processors saving the cost of a separate watchdog. If your system has multiple processors, tie the watchdog

timer software logic together and provide an interprocessor link to detect health. If one processor detects that the other is malfunctioning, it yanks the reset of both. If this is the route to go, care must be taken not to have a common mode failure (software or hardware).

Figure 2

More Robust Watchdog Disable



What is needed in the software portion of a robust Watchdog Timer?

These guidelines contain must haves and should haves:

1. The software watchdog timer must detect all software errors that can prevent the critical core functionality from operating correctly. Each system designer must identify what this critical core functionality is. For example, in several systems we have designed we have a browser based setup program. This software is not considered part of the critical core functionality because:
 - a. It is usually only run when first configuring the system and thus doesn't carry with it the need for super robust operation
 - b. A user is present who could power the unit down in case of a lock-up

When a critical software error is detected, the hardware watchdog timer must not be tickled thus allowing the watchdog timer to reset and the system to restart. This means that if there are 25 threads that run in the system and 15 periodic interrupt handlers, the software watchdog must detect failures in any of these concurrent processes. Does a thread abort? Is a periodic interrupt handler not getting called? There are a number of mechanisms for doing this. Jack recommends some in his article.

2. Non-periodic operations that are part of the critical core functionality (as needed interrupts or threads that are only run on demand) should be handled in a special manner. For example, if the system knows that it is in a certain mode and that it should be servicing certain interrupts, it should verify that such non-periodic interrupts have been serviced. For example we recently implemented an ftp server on an embedded system. This server only spawned certain threads when an ftp client logged in a user. Occasionally the server would lock up (prior to final test – thank heavens!). We needed to provide a means for detecting such lock ups and take appropriate actions.
3. The software watchdog must completely distrust the hardware watchdog timer. If hardware can do anything wrong – it will. For example, if the watchdog timer software determines that a thread has died and it is going to stop processing the watchdog timer, it should verify that the watchdog actually kicks in and log a failure and attempt a software restart if that fails.
4. The software watchdog should provide some means of indicating that a watchdog reset has occurred and why. For example, the memory that contains the fence could also contain a failure number of the last watchdog trip that was knowingly caused by the software. At power up, when the fence is not present, the location could be cleared. When the watchdog timer software detects the need for a reboot, it should write a failure code in that location (for example – User Interface Thread #4 aborted).
5. The software watchdog should provide some means of logging a timestamp as to when the software tickled or “kicked” the watchdog. This timestamp can be contained in that same RAM structure containing the fence and the fault code. This is useful for handling those cases where the system is generating false alarm watchdog resets.
6. The software watchdog should provide some means to detect a system that repeatedly experiences a watchdog timer timeout. At this point, in some systems that we have designed, we either execute a fail safe program or in extreme cases boot a completely different read only OS and application to allow the system to be diagnosed and/or re-loaded. Many times we have seen systems that just keep re-booting and do nothing to correct the problem. Our software, if it is going to be robust should provide a means to fall back to a known working state. Dare I say the word “Safe mode?” But a safe mode that has all of the necessary ingredients to recover or diagnose the system – but not necessarily operate it.

MicroTools provides a device that can be added to a PC system that easily and inexpensively adds a watchdog timer to any PC based system. It is called [KeyDog](#) and

is highly recommended for use on Kiosks and other PC based systems that must be robust.

In Part 8 we will discuss how to design data redundancy into your system. (This whitepaper is not yet released). It is expected to be available in July 2009.

¹ “Great Watchdogs” by Jack Ganssle (<http://ganssle.com/watchdogs.pdf>)

² See MicroTools core values at <http://microtoolsinc.com/values.php>

³ “Great Watchdogs” by Jack Ganssle (<http://ganssle.com/watchdogs.pdf>)

⁴ Jack provides some guidelines if you must use the processor’s internal watchdog. The biggest problems I have seen with internal watchdogs are that the software can modify them and that they often do not provide an external reset to restart external peripherals. The Atmel ARM-9 is a classic example of a worthless internal watchdog if you have any peripherals that can lock up. It resets the processor but doesn’t bring the reset line outside the chip.

⁵ Many will say that this is a must. For ultra high reliability, I would agree. But not all systems need absolute high reliability. The probability that the software disables the watchdog is very small (depending on your design).

⁶ Same comments as before – If you do this you need to make absolutely sure that nothing can lock up before you enable the watchdog. Do it as early as possible. Modify Linux initialization if you have to.

⁷ The SYSESR register has a PORRST and WDRST flag used to indicate what triggered the restart. We would love to find a stand-alone watchdog that has this feature.