watchdog

For those embedded systems that can't be constantly watched by a human, watchdog timers may be the solution.

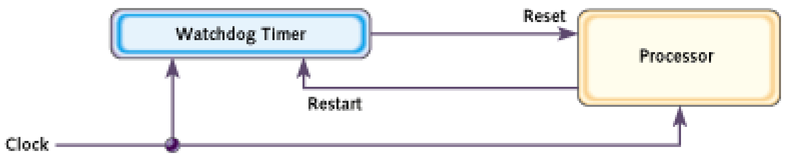
Most embedded systems need to be self-reliant. It's not usually possible to wait for someone to reboot them if the software hangs. Some embedded designs, such as space probes, are simply not accessible to human operators. If their software ever hangs, such systems are permanently disabled. In other cases, the speed with which a human operator might reset the system would be too slow to meet the uptime requirements of the product.

What is a watchdog timer?

A watchdog timer (WDT) is a piece of hardware that can be used to automatically detect software anomalies and reset the processor if any occur. Generally speaking, a watchdog timer is based on a counter that counts down from some initial value to zero. The embedded software selects the counter's initial value and periodically restarts it. If the counter ever reaches zero before the software restarts it, the software is presumed to be malfunctioning and the processor's reset signal is asserted. The processor (and the embedded software it's running) will be restarted as if a human operator had cycled the power.

Figure 1 shows a typical arrangement. As shown, the watchdog timer is a chip external to the processor. However, it could also be included within the same chip as the CPU. This is done in many microcontrollers. In either case, the output from the watchdog timer is tied directly to the processor's reset signal.

Figure 1: A typical watchdog setup



Kicking the dog

The process of restarting the watchdog timer's counter is sometimes called "kicking the dog." The appropriate visual metaphor is that of a man being attacked by a vicious dog. If he keeps kicking the dog, it can't ever bite him. But he must keep kicking the dog at regular intervals to avoid a bite. Similarly, the software must restart the watchdog timer at a regular rate, or risk being restarted.

A simple example is shown in Listing 1. Here we have a single infinite loop that controls the entire behaviour of the system. This software architecture is common in many embedded systems with low-end processors and behaviours based on a single operational frequency. The hardware implementation of this watchdog allows the counter value to be set via a memory-mapped register.

```
uint16 volatile * pWatchdog =
(uint16 volatile *) 0xFF0000;

main(void)
{
        hwinit();

        for (;;)
        {
                *pWatchdog = 10000;
                read_sensors();
                control_motor();
                display_status();
        }
}
```

Suppose that the loop must execute at least once every five milliseconds. (Say the motor must be fed new control parameters at least that often.) If the watchdog timer's counter is initialised to a value that corresponds to five milliseconds of elapsed time, say 10,000, and the software has no bugs, the watchdog timer will never expire; the software will always restart the counter before it reaches zero.

Watchdog software

A watchdog timer can get a system out of a lot of dangerous situations. However, if it is to be effective, resetting the watchdog timer must be considered within the overall software design. Designers must know what kinds of things could go wrong with their software, and ensure that the watchdog timer will detect them, if any occur.

Systems hang for any number of reasons. A logical fallacy resulting in the execution of an infinite loop is the simplest. Suppose such a condition occurred within the read_sensors() call in Listing 1. None of the other software (except ISRs, if interrupts are still enabled) would get a chance to run again.

Another possibility is that an unusual number of interrupts arrives during one pass of the loop. Any extra time spent in ISRs is time not spent executing the main loop. A dangerous delay in feeding the motor new control instructions could result.

When multitasking kernels are used, deadlocks can occur. For example, a group of tasks might get stuck waiting on each other and some external signal that one of them needs, leaving the whole set of tasks hung indefinitely.

If such faults are transient, the system may function perfectly for some length of time after each watchdog-induced reset. However, failed hardware could lead to a system that constantly resets. For this reason it may be wise to count the number of watchdog-induced resets, and give up trying after some fixed number of failures.

Karate lessons

An actual watchdog implementation would usually have an interface to the software that is more complex than the one in Listing 1. When the set of instructions required to reset the watchdog is very simple, it’s possible that buggy software could perform this action by accident. Consider a bug that causes the value 10,000 to be written to every location in memory, over and over again. This code would regularly restart the watchdog counter, and the watchdog might never bite. To prevent this, many watchdog implementations require that a complex sequence of two or more consecutive writes be used to restart the watchdog timer.

If the watchdog is built into your microcontroller, it may not be enabled automatically when the device resets. You must be sure to enable it during hardware initialization. To provide protection against a bug accidentally disabling the watchdog, the hardware design usually makes it impossible to disable the watchdog timer once it has been enabled.

If your software can do a complete loop faster than the watchdog period, the structure in Listing 1 may work fine for you. It gets more challenging if some part of your software takes a long time to complete. Say you have a loop that waits for an element to heat to a certain temperature before returning. Many watchdog timers have a maximum period of around two seconds. If you are going to delay for more than that length of time, you may have to kick the dog from within the waiting loop. If there are many such places in your software, control of the watchdog can become problematic.

System initialization is a part of the code that often takes longer than the watchdog timer's maximum period. Perhaps a memory test or ROM to RAM data transfer slows this down. For this reason, some watchdogs can wait longer for their first kick than they do for subsequent kicks.

As threads of control are added to software (in the form of ISRs and software tasks), it becomes ineffective to have just one place in the code where the watchdog is kicked.

Choosing a proper kick interval is also an important issue, one that can only be addressed in a system-specific manner. These and other issue of greater complexity are discussed in the references listed at the end of this article.

Conclusion

A watchdog timer is a useful tool in helping your system recover from transient failures. Since it is so common to find watchdogs built into modern microcontrollers, the technique is effectively free. If you are working on a mission-critical system, then either common sense or a regulatory body will insist that you use a watchdog. It's always a good idea to make your systems more self-reliant.

Niall Murphy has been writing software for user interfaces and medical systems for ten years. He is the author of Front Panel: Designing Software for Embedded User Interfaces, and contributing editor to ESP. Niall's training and consulting business is based in Galway, Ireland.

Michael Barr is the editor in chief of *ESP.* He is also the author of *Programming Embedded Systems in C and C++* and an adjunct faculty member at the University of Maryland College Park.

References:

1. Murphy, Niall. “Watchdog Timers,” *Embedded Systems Programming,* November 2000, p. 112.
2. Santic, John S. “Watchdog Timer Techniques,” *Embedded Systems Programming,* April 1995, p. 58.

See also Dale Lantrip and Larry Bruner’s “General Purpose Watchdog Timer Component for a Multitasking System,” *Embedded Systems Programming,* April 1997.

Related articles:

- Ultra-low power watchdogs
- *A quick introduction to instruction set architecture and extensibility*
- *An introduction to confidential edge computing for IoT security*