Watchdog Timers

To keep a watchdog timer from resetting your system, you've got to kick it regularly. But that's not all there is to watchdog science. We will examine the use and testing of a watchdog, as well as the integration of a watchdog into a multitasking environment.

Making proper use of a watchdog timer is not as simple as restarting a counter. If you have a watchdog timer in your system, you must choose the timeout period carefully, ensure that the watchdog timer is tested regularly, and, if you are multitasking, monitor all of the tasks. In addition, the recovery actions you implement can have a big impact on overall system reliability.

A watchdog timer is a piece of hardware, often built into a microcontroller that can cause a processor reset when it judges that the system has hung, or is no longer executing the correct sequence of code. This article will discuss exactly the sort of failures a watchdog can detect, and the decisions that must be made in the design of your watchdog system. The first half of the article will assume that there is no RTOS present. The second half covers a scheme for making use of a watchdog in a multi-tasking system.

The hardware component of a watchdog is a counter that is set to a certain value and then counts down towards zero. It is the responsibility of the software to set the count to its original value often enough to ensure that it never reaches zero. If it does reach zero, it is assumed that the software has failed in some manner and the CPU is reset.

In other texts you will see various terms for restarting the timer: strobing, stroking or updating the watchdog. However, in this article we will use the more visual metaphor of a man kicking the dog periodically—with apologies to animal lovers. If the man stops kicking the dog, the dog will take advantage of the hesitation and bite the man.

The watchdog is kicked by writing to an I/O line or a specific memory address. In some implementations a combination of addresses must be accessed in successive bus cycles. This reduces the chance that an errant program will accidentally kick the dog on a regular basis, preventing a bite.

It is also possible to design the hardware so that a kick that occurs too soon will cause a bite, but in order to use such a system, very precise knowledge of the timing characteristics of the main loop of your program is required.

What errors are caught

A properly designed watchdog mechanism should, at the very least, catch events that hang the system. In electrically noisy environments, a power glitch may corrupt the program counter, stack pointer, or data in RAM.

If the man stops kicking the dog, the dog will take advantage of the hesitation and bite the man.

The software would crash almost immediately, even if the code is completely bug free. This is exactly the sort of transient failure that watchdogs will catch.

Bugs in software can also cause the system to hang, if they lead to an infinite loop, an accidental jump out of the code area of memory, or a deadlock condition (in multitasking situations). Obviously, it is preferable to fix the root cause, rather than getting the watchdog to pick up the pieces. In a complex embedded system it may not be possible to guarantee that there are no bugs, but by using a watchdog you can guarantee that none of those bugs will hang the system indefinitely.

First aid

Once your watchdog has bitten, you have to decide what action to take. The hardware will usually assert the processor's reset line, but other actions are also possible. For example, when the watchdog bites it may directly disable a motor, engage an interlock, or sound an alarm until the software recovers. Such actions are especially important to leave the system in a safe state if, for some reason, the system's software is unable to run at all (perhaps due to chip death) after the failure.

A microcontroller with an internal watchdog will almost always contain a status bit that gets set when a bite occurs. By examining this bit after emerging from a watchdog-induced reset, we can decide whether to continue running, switch to a fail-safe state, and/or display an error message. At the very least, you should count such events, so that a persistently errant application won't be restarted indefinitely. A reasonable approach might be to shut the system down if three watchdog bites occur in one day.

If we want the system to recover quickly, the initialization after a watchdog reset should be much shorter than power-on initialization. A possible shortcut is to skip some of the device's self-tests. On the other hand, in some systems it is better to do a full set of self-tests since the root cause of the watchdog timeout might be identified by such a test.

In terms of the outside world, the recovery may be instantaneous, and the user may not even know a reset occurred. The recovery time will be the length of the watchdog timeout plus the time it takes the system to reset and perform its initialization. How well the device recovers depends on how much persistent data the device requires, and whether that data is stored regularly and read after the system resets.

Sanity checks

Kicking the dog on a regular interval proves that the software is running. It is often a good idea to kick the dog only if the system passes some sanity check, as shown in Figure 1. Stack depth, number of buffers allocated, or the status of some mechanical component may be checked before deciding to kick the dog. Good design of such checks will increase the family of errors that the watchdog will detect.

One approach is to clear a number of flags before each loop is started, as shown in Figure 2. Each flag is set at a certain point in the loop. At the bottom of the loop the dog is kicked, but first the flags are checked to see that all of the important points in the loop have been visited. The multitasking approach discussed later is based on a similar set of sanity flags.

For a specific failure, it is often a good idea to try to record the cause (possibly in NVRAM), since it may be difficult to establish the cause after the reset. If the watchdog bite is due to a bug (would that be a bug bite?) then any other information you can record about the state of the system, or the currently active task will be valuable when trying to diagnose the problem.

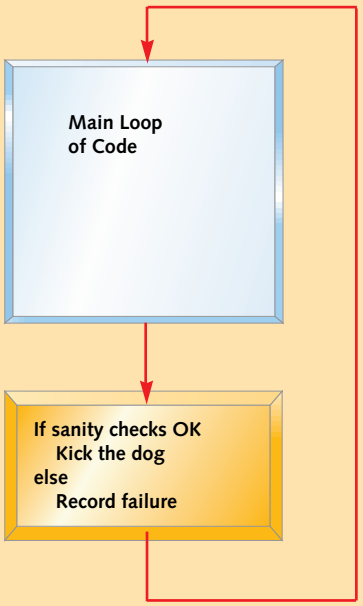
Choosing the timeout interval

Any safety chain is only as good as its weakest link, and if the software policy used to decide when to kick the dog is not good, then using watchdog hardware can make your system less reliable. If you do not fully understand the timing characteristics of your program, you might pick a timeout interval that is too short. This could lead to occasional resets of the system, which may be difficult to diagnose. The inputs to the system, and the frequency of interrupts, can affect the length of a single loop.

One approach is to pick an interval which is several seconds long. Use this approach when you are only trying to reset a system that has definitely hung, but you do not want to do a detailed study of the timing of the system. This is a robust approach. Some systems require fast recovery, but for others, the only requirement is that the system is not left in a hung state indefinitely. For these more sluggish systems, there is no need to do precise measurements of the worst case time of the program's main loop to the nearest millisecond.

When picking the timeout you may also want to consider the greatest amount of damage the device can do between the original failure and the watchdog biting. With a slowly responding system, such as a large thermal mass, it may be acceptable to wait 10 seconds before resetting. Such a long time can guarantee that there

FIGURE 1 At the end of each execution of the main loop, the dog is kicked before starting over



will be no false watchdog resets. On a medical ventilator, 10 seconds would have been far too long to leave the patient unassisted, but if the device can recover within a second then the failure will have minimal impact, so a choice of a 500ms timeout might be appropriate. When making such calculations be sure to include the time taken for the device to start up as well as the timeout time of the watchdog itself.

One real-life example is the Space Shuttle's main engine controller.[1] The watchdog timeout is set at 18ms, which is shorter than one major control cycle. The response to the watchdog biting is to switch over to the backup computer. This mechanism allows control to pass from a failed computer to the backup before the engine has time to perform any irreversible actions.

While on the subject of timeouts, it is worth pointing out that some watchdog circuits allow the very first timeout to be considerably longer than the timeout used for the rest of the periodic checks. This allows the processor time to initialize, without having to worry about the watchdog biting.

While the watchdog can often respond fast enough to halt mechanical systems, it offers little protection for damage that can be done by software alone. Consider an area of non-volatile RAM which may be overwritten with rubbish data if some loop goes out of control. It is likely that such an overwrite would occur far faster than a watchdog could detect the fault. For those situations you need some other protection such as a checksum. The watchdog is really just one layer of protection, and should form part of a comprehensive safety net.

Multiplying the interval

If you are not building the watchdog hardware yourself, then you may have little say in determining the longest interval available. On some microcontrollers the built-in watchdog has a maximum timeout on the order of a few hundred milliseconds. It you decide that you want more time, you need to multiply that in software.

Say the hardware provides a 100ms timeout, but your policy says that you only want to check the system for sanity every 300ms. You will have to kick the dog at an interval shorter than 100ms, but only do the sanity check every third time the kick function is called. This approach may not be suitable for a single loop design if the main loop could take longer than 100ms to execute.

One possibility is to move the sanity check out to an interrupt. The interrupt would be called every 100ms, and would then kick the dog. On every third interrupt the interrupt function would check a flag that indicates that the main loop is still spinning. This flag is set at the end of the main loop, and cleared by the interrupt as soon as it has read it.

If you take the approach of kicking the watchdog from an interrupt, it is vital to have a check on the main loop, such as the one described in the previous paragraph. Otherwise it is possible to get into a situation where the main loop has hung, but the interrupt continues to kick the dog, and the watchdog never gets a chance to reset the system.

Self-test

Assume that the watchdog hardware fails in such a way that it never bites. How would you ever know? When the system works, such a fault is not apparent. The fault would only be discovered when some failure that normally leads to a reset, instead leads to a hung system. If such a failure was acceptable, you would never have bothered with the watchdog in the first place.

If you think watchdog failure is a rare thing, think again. Many systems contain a means to disable the watchdog, like a jumper that connects the watchdog output to the reset line. This is necessary for some test modes, and for debugging with any tool that can halt the program. If the jumper falls out, or a service engineer who removed the jumper for a test forgets

to replace it, the watchdog will be rendered toothless.

The simplest way for a device to do a start-up self-test is to allow the watchdog to timeout, causing a processor reset. To avoid looping infinitely in this way, it is necessary to distinguish the power-on case from the watchdog reset case. If the reset was due to a power-on, then perform this test, but if the reset was due to a watchdog bite, then we may already be running the test. Usually you will want to write a value in RAM that will be preserved through a reset, so you can check if the reset was due to a watchdog test or to a real failure. A counter should be incremented while waiting for the reset. After the reset, check the counter to see how long you had to wait for the timeout, so you are sure that the watchdog bit after the correct interval.

If you are counting the number of watchdog resets in order to decide if the system should give up trying, then be sure that you do not inadvertently count the watchdog test reset as one of those.

Multitasking

A watchdog strategy has four objectives in a multitasking system:

- To detect an operating system failure that is preventing some or all of the tasks from running
- To detect an infinite loop in any of the tasks
- To detect deadlock involving two or more tasks
- To detect if some lower priority tasks are not getting to run because higher priority tasks are hogging the CPU

Typically, not enough timing information is available on the possible paths of any given task to check for a minimum execution time or to set the time limit on a task to be exactly the time taken for the longest path. Therefore, while all infinite loops are detected, an error that causes a loop to execute a number of extra iterations may go undetected by the watchdog mechanism.

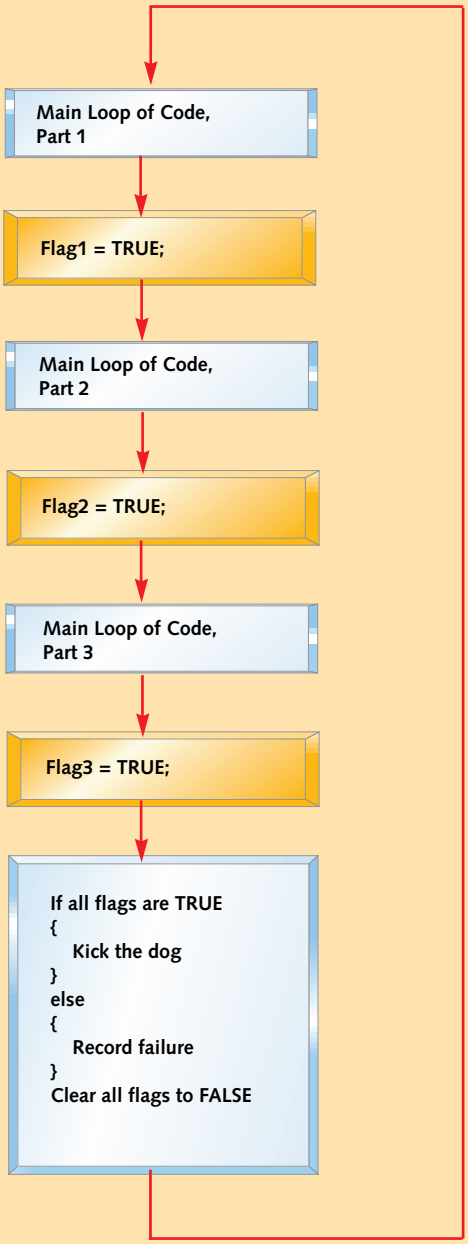
A number of other considerations have to be taken into account to make any scheme feasible:

- The extra code added to the normal tasks (as distinct from a task created for monitoring tasks) must be small, to reduce the likelihood of becoming prone to errors itself
- The amount of system resources used, especially CPU cycles, must be reasonable

The solution I will describe was used on a medical ventilator running on the RTXC real-time operating system. The idea was loosely influenced by Agustus P. Lowell's article "The Care and Feeding of Watchdogs," which describes a way to build the watchdog scheme into the RTOS itself.[2] Unlike Lowell's scheme, however, this scheme can run on top of any RTOS, without requiring changes to the RTOS code.

This scheme uses a task dedicated to the watchdog. This task wakes up at a regular interval and checks the sanity of all other tasks in the system. If all tasks pass the test, the watchdog is kicked. The watchdog monitor task runs at a higher priority than the tasks it is monitoring.

FIGURE 2 Use three flags to check that certain points within the main loop have been visited



The nature of the tasks

Most tasks have some minimum period during which they are required to run. A task may run in reaction to a timer that occurs at a regular interval. These tasks have a start point through which they pass in each execution loop. These tasks are referred to as *regular tasks*. Other tasks respond to outside events, the frequency of which cannot be predicted. These tasks are referred to as *waiting tasks*. First we will discuss how the scheme will work if all tasks are regular and then we will explain what extra work has to be done for waiting tasks.

The watchdog timeout can be chosen to be the maximum time during which all regular tasks have had a chance to run from their start point through one full loop back to their start point again. Each task has a flag which can have two values, ALIVE and UNKNOWN. The flag is later read and written by the monitor.

The monitor's job is to wake up before the watchdog timeout expires and check the status of each flag. If all flags contain the value ALIVE, every task got its turn to execute and the watchdog may be kicked. Some tasks may have executed several loops and set their flag to ALIVE several times, which is acceptable. After kicking the watchdog, the monitor sets all of the flags to UNKNOWN. By the time the monitor task executes again, all of the UNKNOWN flags should have been overwritten with ALIVE. Figure 3 shows an example with three tasks.

Waiting tasks

Waiting tasks can't be guaranteed to pass through their start point within any finite amount of time. These tasks normally have one or more points at which they are waiting on an external event, such as a user key action or communication from another processor. At those points, the flags are set to the value ASLEEP. After the wait, the

flag is set to ALIVE, and the process continues as described above. The monitor changes its scheme as follows: if the monitor checks the flags and sees the value ASLEEP, it considers that state to be valid. So, if all flags are either ASLEEP or ALIVE then the watchdog is kicked.

The disadvantage is that if a task sets a flag to ASLEEP and never changes it back, it always passes the test and any deadlock or infinite loops in that task go undetected. Therefore, one of our rules is that the line of code following the line where the flag is set to ASLEEP must perform the wait, normally using one of the blocking function calls from the operating system. The instruction which follows the wait must set the flag to ALIVE. For example:

```
myFlag = ASLEEP;
KS_wait(KEY_PRESS_HAPPENED);
myFlag = ALIVE;
```

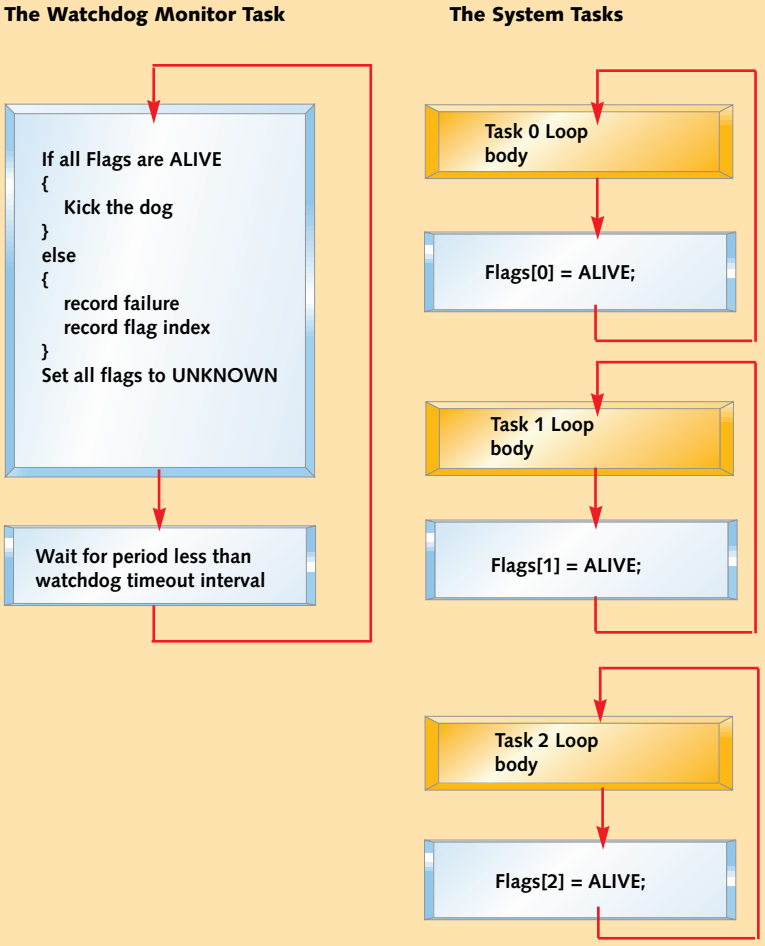
Because there are no conditions or branches in this sequence, no set of circumstances allow the task to continue with the flag in the ASLEEP state.

Once the flag has been set to ALIVE, the task must run to some point where the flag is again set to ALIVE or ASLEEP, before the monitor task has time to clear the flag to UNKNOWN and wait one timeout period. Many tasks have only one place where they wait on an external event and set the flag to ASLEEP. Those tasks must complete one full loop and be back at the three lines shown above in less time than the monitor's timeout.

Note that this mechanism is not used on all blocking calls to the operating system; it is only used for the waits that are dependent on events for which a finite return time cannot be guaranteed.

There are still some concerns with this scheme. If a deadlock occurs that involves waits in a number of waiting tasks while each of the waiting tasks has its flag set to ASLEEP, the monitor cannot detect the fault. In order to avoid this pitfall, a graph can be manually created to show each task with an arrow to the tasks it waits on (drawing arrows only for waits that set the task flag to ASLEEP). If there is a complete loop (for example, Task1 waits on Task2; Task2 waits on Task3; and Task3 waits on Task1), then these waits are not genuinely waiting for external events and you should consider whether the task flag should be set to ASLEEP at all of these points. If such a loop cannot be avoided, an extra timeout could be set on one of the waits (assuming that your RTOS supports timed waits), and this timeout would provide protection

FIGURE 3 The monitor task checks the sanity flags of each of the three system tasks



against a deadlock. This timeout could be far longer than the watchdog timeout period. In the case of this extra timer timing out, the system would be judged to be in deadlock.

In some cases, you may choose to assign two flags to one task. The flags could then be set to ALIVE at different points within the task's main loop. This would catch a problem where a task was stuck in a loop that reset one of the flags but skipped some vital part of its work. The monitor would only consider the task to be healthy if both flags are set to ALIVE within each period.

For waiting tasks, all of the tasks' flags are set to ASLEEP at the waiting point and all of them set to ALIVE immediately afterwards. For example if a task was allocated two flags called myFlag1 and myFlag2 then the sequence of calls when this task is waiting is as follows:

```
myFlag1 = ASLEEP;
myFlag2 = ASLEEP;
KS_wait(KEY_PRESS_HAPPENED);
myFlag1 = ALIVE;
myFlag2 = ALIVE;
```

Concurrent access

Since writes of a single byte are atomic, it is safe to use a single byte as a flag for a single task. No matter when the task switch occurs, it is impossible to get an illegal value written to the byte.

In the case of the monitor, the byte is read and then written. Theoretically, a task switch between the read and the write could change the state of the byte, and then that change would be overwritten by the monitor. This can never happen if the monitor is a higher priority task than the tasks being monitored. The tasks being monitored never read the flag. They only write to it.

Monitor interval

As stated, the timeout interval must be enough for all of the tasks being monitored to complete at least one loop. If there is a big difference between the shortest task loop and the longest then the tasks with shorter execution times may only be getting checked after a few hundred loops. The list of flags can be divided into high frequency flags and low frequency flags. Each time the monitor is awakened, the high frequency tasks' flags are checked, but the low frequency tasks' flags are only checked on every *n*th iteration, where *n* is the ratio between the high and low frequency.

Debugging

When testing and debugging the system, it is a good idea to run the system with the watchdog timeout set tighter than it normally will be in the field. This will help identify any of the paths in the code that are borderline.

It is also a good idea to install the monitor task early in the development cycle, since that will show how the system reacts to the real bugs in the monitored tasks during development. During debugging, always place a breakpoint in the monitor task at the point where it detects a failed flag. Then a failed task is not only detected immediately, but you can also use the debugger to look at its state and figure out why it missed its deadline.

Priority of monitoring task

This watchdog scheme is designed on the assumption that the monitoring task is running at a higher priority than any of the tasks that it is monitoring. This has one drawback. It means that it may take up CPU cycles at a time when another task may be trying to meet some hard real-time target. If your monitoring task performs checks other than the flags described here, and if those checks consume a lot of CPU cycles, you may want to consider altering this scheme to one where the monitoring task runs at a lower priority. If you do this you will have to ensure that the watchdog task is scheduled to run more often so that it will not be deferred for so long by a high priority task that it does not strobe the hardware watchdog in time. For example you might schedule it to kick the dog every 25ms, even though the hardware watchdog only requires a kick every 50ms. It will then survive a 25ms delay caused at a time when a higher priority task is running.

In this case, two possibilities exist for a reset. One is that a task blocks and its flag does not get set to ALIVE. The monitor task will notice this and cause a reset. The second is that a task loops, preventing the monitoring task from running, and then the hardware watchdog will eventually bite.

Using a lower priority task will improve the ability of high priority tasks to meet their hard real-time targets. The disadvantage of such an approach is that you lose the opportunity to record the identity of the task that fails to set its flag to ALIVE, which is useful debugging information. I also believe that it is harder to ensure that there are no circumstances where a properly functioning system will lock out the monitoring task for long enough to get an unwanted kick.

When the lower priority task is the monitoring task, you will also have to address the possibility that another task may interrupt the monitaring task while the flags are being updated. The assumptions made in the "Concurrent access" section no longer hold, and the other task may update the flag that the monitoring task ahs already read, but before the monitoring task has a chance to write to it. One option is to use a resource lock on the set of flags. Another option is to ensure that examining and updating the flag in the monitoring task is performed as an atomic read-and-modify operation, which may be available as a single CPU opcode, or your RTOS may provide a facility to do this.

Conclusion

A good watchdog mechanism requires careful consideration of both software and hardware. It also requires careful consideration of what action to take when the failure is detected. When you design with watchdog hardware, make sure you decide early on exactly how you intend to make best use of it, and you will reap the benefits of a more robust system. **esp**

Niall Murphy has been writing software for user interfaces and medical systems for 10 years. He is the author of Front Panel: Designing Software for Embedded User Interfaces. *Murphy's training and consulting business is based in Galway, Ireland. He welcomes feedback and can be reached at nmurphy@panelsoft.com.*

References

1. *www.hq.nasa.gov/office/pao/History/computers/Ch4-7.html*
2. Lowell, Agustus P., "The Care and Feeding of Watchdogs," *Embedded Systems Programming*, April 1992, p. 38.

Acknowledgments

The author gives thanks to Bill Gatliff for assistance, encouragement, and ideas for this article.