Pseudo-Reliable Code Development
- Achieving Higher Reliability at Low Cost.

Introduction

Arguably the most important software development problem is, building software to customer demands that is more reliable, faster and cheaper. Success in meeting these demands affects the market share and profitability of a product. These demands conflict, due to the very fact that the cost and time required to implement reliability is extremely high. While these high costs can be justified in developing complex, expensive systems for use in aircrafts, medical devices or mission critical applications, they cannot be in commercial products.

This article describes a reactive methodology, which focuses on detecting an error and correcting it, rather than using traditional proactive methodology to achieve reliability. In the traditional methods, a great deal of time and effort is spent to prevent such errors from occurring in the first place. This involves considerable amount of effort and resources to improve quality. Unfortunately, it has been repeatedly validated that software can never be without defects. The reactive methodology takes this as a fact and designs around detecting and handling errors when they occur. This new methodology for implementing embedded systems will assure low cost.

Traditional Prescriptive Reliability

Traditionally we have been trained to achieve reliability by doing a careful design. The prescribed method is to use a modular design while developing our embedded systems. Testing and designing smaller, simpler parts build big complex systems. Use of standards, especially in specific market segments, such as the Motor Industry Software Reliability Association (MISRA) intended for embedded automotive applications has shown to improve reliability by increasing the overall quality of the product. Another time-honored method to achieve reliability is by doing design reviews and code reviews at each step of the development cycle.

One of the major problems in using the traditional methods to achieve reliability is the high cost associated with them. NASA's space shuttle software is probably the best code ever written, with an average error rate of about one per 400,000 lines. The average cost of developing software with such high reliability is around $1,000 per line. Typical commercial shrink-wrapped software, with an average bug count of around 1 bug per 1000 lines of code, costs somewhere between $10 to $100 per line of code

Trying to achieve reliability also consumes a huge amount of another significant resource: time. Software companies face a tough choice of deciding between highly reliable software that is shipped late and unproven, lower reliability software that is shipped on schedule. This time to market in embedded systems, especially in consumer electronics tends to directly affect the bottom line of various companies.

Even though it is possible to write great code, but despite paying gigantic sums, perfection is still elusive. Outside of academia, software cannot be "proven" correct. Indeed, even after writing the best code, by employing the best development teams and using formal mechanisms to achieve reliability, perilous failures have been seen. The failures of Mars Pathfinder, Patriot Missile Defense Failure at Dhahran, Saudi Arabia and the Therac-25 incident give an expressive illustration of the fact. [1][2]

The increasing complexity of embedded systems has led to a corresponding increase in the dependence on 3rd party software. Most of the times this software is distributed in pre-compiled format with limited access to the source code. This means some of the not so obvious bugs in the 3rd party software could cause reliability concerns. Since most of the time access to the source code is restricted, achieving reliability by conventional methods is not likely in this case.

Note: 3rd party software in general is no more reliable than homegrown software and may even be less so due to poorly understood interfaces and inner mechanisms, which are often hidden from the purchaser. Obviously 3rd party software is also developed under the same development time constraints as home grown software and even more stringent cost constraints.

What else can we do?

Achieving reliability need not be costly or time consuming; indeed there are various ways in which reliability can be increased by using reactive methods. By reactive methods, I mean recovery or corrective methods, which are used after the system has performed an unreliable behavior.

One of the most common examples of a reactive method already in use is the watchdog. A watchdog timer is a combination of hardware and software that is used to prevent system hangs (often by resetting the processor if any occurs). Generally, watchdog timer is based on a down counter, which starts from a given value and resets the system when it reaches zero. The embedded software is responsible for periodically restarting (kicking) the watchdog. If the counter ever reaches zero it is assumed that the software is malfunctioning and the system is reset.

While using the watchdog, we should make sure that the following conditions are met:

- Watchdog timer interval should be carefully so chosen that the longest possible execution path between watchdogs updates will be less than one watchdog-timer interval.
- The kicking of the watchdog must depend on the correct sequencing of critical software in the system

The first condition can be met by having the path through the code between watchdog updates are well defined and deterministic enough to guarantee an execution time less than one watchdog update period. To meet the second condition, in a system without an operating system, we can kick the watchdog at the end of each execution of the main

loop. In a multitasking operating system, we can have a separate task to periodically kick the watchdog timer. The priority at which this task runs needs to be fine-tuned so that we run at a priority lower than the priority of all other tasks which we need to protect and also consider the worst case scenario, for the time delay this task gets a chance to run. We also need to consider the worst-case scenario for the time delay before this task gets a chance to run

For the sake of clarity, let's take an example where we have a preemptive OS**,** running the following tasks (as shown in Figure 1). The priority of each task can range from 0-255 with 0 being the highest priority.

Task Name	**Task Priority**
tExcTask	0
tShell	1
tRlogind	5
timerTask	10
tNetTask	50
tHttpdTask	150
tSnmpdTask	150
tLoggerTask	250

Figure 1

If we can predict the maximum time each of the tasks takes before giving up control, we can have the watchdogTask run at priority 255. Since here we are executing watchdogTask as the lowest priority task, if the watchdogTask does not get a chance to run within the watchdog period the watchdog will not be kicked and as a result the system will be reset.

In real life it is extremely difficult if not impossible to predict the maximum time taken by an entire task. Assuming that the code for tHttpdTask was obtained from a third party in binary format. According to the specs provided, the average time for tHttpdTask to serve a web page is low, with unpredictable worst times. That being the case, we will not be able to come up with reasonable values for the watchdog timer period.

One of the simplest things we could do here is to have the watchdog task run at priority 100. Assuming all the tasks with a priority of below 100 to be critical, it will make sure that all the critical tasks behave well. In the event of a critical task behaving inconsistently, the watchdog timer task will never get to run, thereby never kicking the timer. This will eventually cause the system to reset.

If used properly a watchdog timer can successfully detect all tasks frozen or hung because of deadlock within tasks and/or infinite looping of a task. Watchdog can also detect faults in scheduling algorithm, if some tasks are not getting scheduled because higher priority tasks are hogging the CPU

Even though the watchdog seems to be simple in concept, it is often difficult to implement reliably. It may not detect some task crashes or system corruption. Taking the above example if tRlogind task crashed without having significant side effects, the watchdog may never be able to find out the cause of the crash. In many cases, we could have critical tasks running as low priority threads. Using a simple watchdog in these cases, we will never be able to detect these tasks crash or hang, especially in the presence of non-deterministic tasks.

Policing the individual tasks [introducing Task Monitor]

Because of the reasons mentioned above, it seems that it would be better if we could have the task monitor each task and verify the sanity of the task before kicking the dog. The simple way to do it is for the task to update a monitor table, which has entries for each task that needs to be monitored. A monitor utility could be run periodically to verify if all the tasks have updated their states properly. In a multitasking operating system, all the tasks can be divided into 2 categories, some that execute periodically and the rest, that wait for an event to happen.

Monitoring a periodic task is easy because its frequency is deterministic, each of the periodic tasks could set alive flag whenever they are successfully run. The monitor task could check and reset the flag if the task had set its flag to alive. If any task has not set their flag as alive, within, the specified period, the monitor task does not kick the watchdog, causing the system to reset.

The other types of tasks wait for an event to happen and can potentially wait for unpredictable periods of time without executing. For these tasks, by using additional flags we can have the monitor utility that makes sure that the tasks complete their execution within a specified time after an event happens. The monitor ignores the task if it is waiting for an event.

The following example lists one possible implementation. In this implementation periodic functions are also treated as event based functions waiting for a timer event. A *monitorTable* stores the state of the tasks, the possible values for which are

- *taskRunning* – When the task is running and has not yet completed one loop in this period
- *taskEventWait* – The task is waiting on an event, which is yet to occur
- *taskEventOccur* – The event on which the task was waiting for has occurred

During initialization, an entry in the *monitorTable* is created for each task that needs to be monitored. The task state is initialized to *taskEventWait*. The tasks, upon receiving an event, it should set their state to *taskEventOccur*. After the task completes its execution loop and is waiting for the next event, it should set the state to *taskEventWait.*

The monitor task, which is invoked periodically, changes the state of each task, which is in state *taskEventWait* to *taskRunning*. If in the next invocation of the monitor task, if it finds a task entry, which is still in *taskRunning* state, it can conclude that the task has

stopped running and then it should not kick the watchdog. This would eventually result in the watchdog resetting the whole system and starting all over again.

This monitor task should be run at either a high priority or at timer ISR context to make sure that the watchdog is kicked in time, if all the tasks are verified to be in good condition.

Resetting the System – Getting up after the fall

Resetting a system seems to be one of the crudest methods of implementing reliability and yet it is the primary method employed by watchdogs. Although the system is recovered from an unknown state still there are tasks the system was doing before it went into this stage. In some of these cases some other perfectly good task could be performing a vital function, which also gets interrupted by this reset.

One of the critical issues we have to deal with when we are trying to reboot the system once a problem is detected is that we could end up with the system getting stuck in an infinite reset – boot loops. The error condition, which caused the reset, in the first place could potentially still exist even after the reset, and could possibly cause another reset as soon as the system finishes initialization. Making sure that after a predetermined number of reboots the system goes to a known safe state and stop resetting the system can easily prevent this. This is very easy to implement, as a watchdog reset will almost always cause a status bit to be set. By examining this bit and updating a counter at boot time, we can determine if we have crossed the threshold of rebooting the system (say a maximum of 1 reset per day) and go to safe mode.

Another issue we need to address is that the system should recover quickly after reset. A reset time shorter than the power on reset time can be achieved by skipping some of the device self-tests in case of a watchdog reset. The self-tests would only be executed if too many resets occurred and the safe-mode was entered. This gives the flexibility of quick recovering from run-time application errors, but also the fall back of detecting hardware errors if too many resets occur

Even after taking into considerations all the points mentioned above, resetting the system still might not be a very good idea. The system could have a perfectly valid task modifying some critical region could cause a system corruption when interrupted by a reset. Some of the other actions we could take if the watchdog bites is to sound an alarm, download and load an alternate image or wait for user intervention.

Another strategy we could follow is to reset only the individual task, which has failed. This could be achieved by some simple modifications made in the taskMonitor routine discussed above. We could further improve the implementation by giving up in case we have reset the task a predetermined number of times and it has still crashed. At this point of time, depending on the criticality of the task and depending on the nature of the system, we could reset the system, ignore the task or raise an alarm.

To implement this along with our above example, the monitorTable should now be expanded to include at least the following values for each task:

- resetFoo - Reset function for each task.
- maxResetNo - Number of max resets allowed for each task.

The Users Perspective

If we use all the techniques mentioned above we can in many cases hide the recovery mechanism from the end user. In terms of the user experience, the recovery may be instantaneous, and the user may not even be even aware that a reset occurred. The recovery time we have is 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.

Looking from the outside, the system can behave like a pseudo-reliable system, which never seems to be in a bad state. As far as a user is concerned, if the user does not see it then the problem does not affect him and therefore to him, does not exist.

Examples from the real world

The reactive methods described above could have a wide variety of applications. They can be used to make critical systems more reliable without significantly increasing the cost of the system. On the other end of the horizon they can also be used in low cost consumer electronic devices where the pressure for time to market and lower per unit cost is demanded. In these quick development cycles, speedy design and low time to test results in oddities, which sometimes can be really annoying to users.

NASA has historically used watchdog timers for Spaceflight, in the Space Shuttle's main engine controller. The controllers are redundant but were not actively synchronize, to quote from **Computers in Spaceflight: The NASA Experience** [3]**, "***Marshall Space Flight Center studied active synchronization, but the additional hardware and software overhead seemed too expensive.*" The controller had a major cycle of 20 microseconds, which was divided into 4 minor cycles of 5 microseconds each. Two watchdog timers with a timeout period of 18 ms were used, which had to be kicked by the software. In case of failure, the engine would be "uncontrollable," for less than one major cycle before the redundant computer takes over.

In one company I know, it was making a network access device which developed a "strange" problem weeks before it was to be released. While testing the final product it was found that the system would freeze after around 3 days of pumping data through it. Debugging the systems seemed to indicate a memory leak, with no idea as to the source of the leak. The company then decided to implement a reactive methodology to deal with this issue. It ran the watchdog timer as a low priority thread with a timeout of around 200 ms. The system (without the power on test) would take around 10 to 15 seconds to completely initialize and start all over again. Since the device was used primarily for browsing, from the users perspective when the device hangs and reset itself the user's browser would time out. By the time the user did a refresh the system would be back

online and would serve the user with the web page. Thus the typical user working on windows OS would never notice the difference.

What else can we do?

These reactive methods, which I have described above, are only a few of the techniques that can be used to achieve reliability; many others can exist and work as well. These methods are simple methods, which can be implemented at low costs, since cost has always been (and more so now) one of the driving forces, which decides the fate of a product.

Monitor the Monitor Task: To further strengthen the reliability of the system we could implement another monitor task (taskMonitorController) which would watch over the monitor task and restart this task monitor if it messes up. To make sure that this taskMonotorController is reliable this task should be responsible for only monitoring the taskMonitor task.

In our quest for higher reliability and to minimize the impact on the system, we could periodically preserve states at system level or task level. On system or task reset we could revert back to the known good state, thereby limiting the impact on the system.

Although these techniques can give a feeling of reliability to the user, they do not guarantee the correct operation of the device. Hardware has long used Error Detection and Correction techniques like parity checking and complex retry mechanisms to give the feel of reliability over an unreliable medium. The author stresses the fact that these techniques cannot deal with a bad design or be responsible for detecting all errors in a system. Nevertheless, they could be very effective in hiding the last bug, which still lurks in our system, from the user.

Vivek Kumar *is an embedded systems consultant and has been writing and developing embedded software for over 8 years. Vivek is based in* ***San Jose****, CA and specializes in board bring-up, developing device drivers and optimizing codes for various systems. He welcomes feedback and can be reached at vivek@employees.org*

References

[1] Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia - http://www.fas.org/spp/starwars/gao/im92026.htm
[2] What really happened on Mars. - http://research.microsoft.com/~mbj/Mars_Pathfinder
[3] Computers in Spaceflight: The NASA Experience www.hq.nasa.gov/office/pao/History/computers/Ch4-7.html
[4] Niall Murphy, "Watchdog Timers," *Embedded Systems Programming,* Nov 2000.
[5] Niall Murphy and Michael Barr, "Introduction to Watchdog Timers," *Embedded Systems Programming,* Oct 2001
[6] Agustus P. Lowell, "The Care and Feeding of Watchdogs," *Embedded Systems Programming,* April 1992,