ESC-302

Stress Testing Embedded Software Applications

T. Adrian Hill
Johns Hopkins University Applied Physics Laboratory
Adrian.Hill@jhuapl.edu

ABSTRACT

This paper describes techniques to design stress tests, classifies the types of problems found during these types of tests, and analyzes why these problems are not discovered with traditional unit testing or acceptance testing. The findings are supported by citing examples from three recent embedded software development programs performed by the Johns Hopkins University Applied Physics Laboratory where formal stress testing was employed. It also defines what is meant by the robustness and elasticity of a software system. These findings will encourage software professionals to incorporate stress testing into their formal software development process.

OUTLINE

1. INTRODUCTON
   1.1 What can be learned by "breaking" the software
2. DESIGNING A STRESS TEST
   2.1 What is a reasonable target CPU load?
   2.2 Other ways to stress the system
   2.3 Characteristics of a Stress Test
3. REAL WORLD RESULTS
   3.1 Case #1: Software Missed Receiving Some Commands When CPU Was Heavily Loaded
   3.2 Case #2: Processor Reset when Available Memory Buffers Were Exhausted
   3.3 Case #3: Unexpected Command Rejection When CPU Was Heavily Loaded
   3.4 Case #4: Processor Reset When RAM Disk Was Nearly Full
   3.5 Synopsis of All Problems Found During Stress Testing
4. SUMMARY

1. INTRODUCTON

Traditional Software Acceptance Testing is a standard phase in nearly every software development methodology. Test engineers develop and execute tests that are defined to validate software requirements. The tests tend to be rigid with specific initial conditions and well-defined expected results. The tests typically execute software within the limits prescribed by the software design. While these tests are often complemented with System Tests or Use Case Tests, these higher level scenarios still conform within the design bounds of the software.

Software Stress Testing, however, runs counter to these traditional approaches as demonstrated in Table 1. Stress testing involves intentionally subjecting software to unrealistic loads while denying it critical system resources. The software is intentionally exercised "outside the box" and known weaknesses and vulnerabilities in the software design may be specifically exploited. Degraded performance of a system under stress may be deemed perfectly acceptable, thus, the interpretation of test results and definition of pass / fail criteria is more subjective. Furthermore, a test that stresses one aspect of the software may lead to undesirable side effects in another area of the software thus the entire system behavior as a whole must be evaluated to properly analyze the results.

Software Acceptance Testing	Software Stress Testing
Black Box Testing. No need to understand software internals	White Box Testing. Tests target weak spots in the software design
Tests are designed to verify that software meets requirements	Tests attempt to "break" the software
Tests exercise software within acceptable bounds	Tests intentionally violate constraints to stress software
Pass / Fail criteria are clearly defined	Pass / Fail criteria are subjective

Table 1 Comparison of Acceptance Testing versus Stress Testing

1.1 What can be learned by "breaking" the software

For critical embedded software applications, the user has an expectation that the software is robust. The dictionary defines **robustness** as *the property of being powerfully built or sturdy; one step below bulletproof*. For software, this means that the implementation should demonstrate resiliency. It is permissible for the software to operate in a degraded fashion (e.g., dropping commands or missing data) while under stress but the degradation should be graceful and recoverable. A lack of robustness would lead to unacceptable behavior such as writing corrupted data or resetting the processor.

Secondarily, the embedded software should demonstrate elasticity. The dictionary defines **elasticity** as the *property of returning to an initial form or state following deformation*. In a software context this means that when the stress is removed the system should return back into its normal operating state and have full functionality restored. A lack of elasticity would leave the software operating in a degraded mode after the stress has been relieved.

While stress testing can validate the robustness and elasticity of the software system, that validation may not be enough to justify the time and effort required to develop and execute such tests. However, there is an additional benefit to stress testing that may not be as readily apparent. Stress testing exposes design flaws and implementation bugs that are difficult or impossible to discover under traditional testing approaches. Furthermore, these flaws and bugs are often present even when the system is not under stress. In such cases, the stress test magnifies these inherent problems allowing them to be more easily detected. Stress testing allows these defects to be corrected before the software is deployed into the user community.

2. DESIGNING A STRESS TEST

Although there is no single approach to designing a stress test, a recommended approach is shown in Figure 1. With this method, the CPU loading is increased at the start of the overall test. Next, a series of subtests are run against the heavily loaded processor. Each subtest exercises some type of targeted stress. The type of stress is highly dependent upon the system under test and could include items such as overflowing input buffers, simulating rapid keystrokes, etc. It is perfectly valid to inspect the software design and inject stress that targets specific areas such as input queues, interrupt service routines and critical resources. As each subtest executes, a qualitative measure of performance should be validated (i.e., robustness). This requires establishing an acceptable level of degraded behavior for each subtest. Finally, after the series of subtests completes, the CPU is returned to nominal levels and it is verified that the software system returns to its nominal operating state to confirm the elasticity of the system.

To simplify development, the subtests can often be developed, debugged and executed “standalone” (while the system is not under stress) to validate the test scripts and establish a baseline behavior before they are incorporated into the larger Software Stress Test. This provides a level of modularity to the overall stress test.

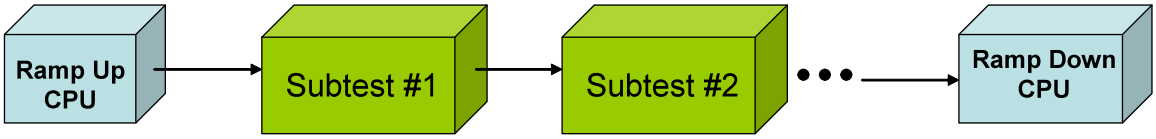


Figure 1 Structure of a Software Stress Test

2.1 What is a reasonable target CPU load?

One obvious way to place the software under stress is to perform steps to increase the CPU load. Increasing the CPU load is especially important when

stress testing pre-emptive multitasking systems. Software designers spend considerable effort assigning task priorities in a pre-emptive system to ensure that all real-time deadlines are met. It has been shown that even if task priorities are poorly assigned, task deadlines will usually be met when the overall CPU load is less than about 85%. Said another way, having 15% overall free CPU load can provide enough margin to mask flaws regarding task priority assignments [LEHOCZKY01]. Since one purpose of the stress test is to expose these flaws, the test should have a target CPU load of 90% or more, if possible.

2.2 Other ways to stress the system

Increasing CPU load is just one of the ways to stress a software system. Other stress methods, which are typically employed in the subtests, include:

- Maximizing I/O data rates
- Maximizing data bus usage
- Maximizing interrupt rate
- Exhausting available memory
- Overflowing queues

2.3 Characteristics of a Stress Test

Ideally, stress tests should be scripted and repeatable. While stress testing may seem less precise than traditional testing because of the subjectivity in evaluating system performance and defining acceptable behavior, it does not mean that the test process is any less formal. One possibility with stress testing is that a problem may be observed one time but cannot be repeated. While not a fool-proof solution, using scripts that can repeat a test will greatly increase the likelihood that intermittent problems can be recreated improving the probability that engineers can isolate the root cause.

Additionally, post-test analysis should always be performed to identify any unexpected anomalies. Checklists can ensure that all necessary data points are verified. If possible, scripting can to verify checklist items during the stress test. Frequently there is a side effect when stressing the system in one area that may have an unexpected effect on another area. Without a rigorous approach to reviewing all data, these side effects may not be observed which undermines a key benefit of Software Stress Testing.

3. REAL WORLD RESULTS

The Johns Hopkins University Applied Physics Laboratory (JHU/APL), located in Laurel, Maryland, has developed spacecraft flight software for three recently launched NASA missions. The embedded software is highly critical (a serious fault can result in loss of mission) and is expected to operate continuously for long periods of time (i.e., years) as shown in Table 2.

Mission	Launch	Duration
MErcury Surface, Space ENvironment, GEochemistry, and Ranging (MESSENGER)	August 2004	8 years
New Horizons (Pluto-Kuiper Belt Mission)	January 2006	9+ years
Solar TErrestrial RElations Observatory (STEREO)	October 2006	2+ years

Table 2 Recent JHU/APL Software Development Programs

All three software development efforts followed a formal software development life cycle including requirements analysis, design, implementation, unit and integration testing, requirements-based testing and system-level acceptance testing. Formal Software Stress Testing was performed on the software near the end of the software life cycle using the techniques described in this paper. The effort required to develop and execute a stress test was far greater than that required for any other individual software acceptance test. Overall, the stress tests represented about 10% of the entire software acceptance test effort.

The stress testing uncovered a total of 32 problems across the three software development efforts. The next subsections provide a detailed analysis of four of the failures to demonstrate typical deficiencies discovered in stress testing. That is followed by a synopsis which classifies all 32 problems by type and analyzes trends and tendencies regarding the problems found while stress testing.

3.1 Case #1: Software Missed Receiving Some Commands When CPU Was Heavily Loaded

A stress subtest was designed to issue a stream of 1000 commands at the highest possible rate to the software while the CPU was heavily loaded (> 90%) but not overloaded. The commands are buffered in hardware and the software must service the buffer before it is overwritten. During the stress subtest, five of the 1000 commands were dropped even though the CPU load during the subtest never reached 100% (see Figure 2). Had the CPU usage peaked at 100% while the commands were received, then dropped commands may have been an acceptable degraded behavior. But when commands were dropped even though there was still free CPU (albeit less than 10%), deeper investigation was warranted.

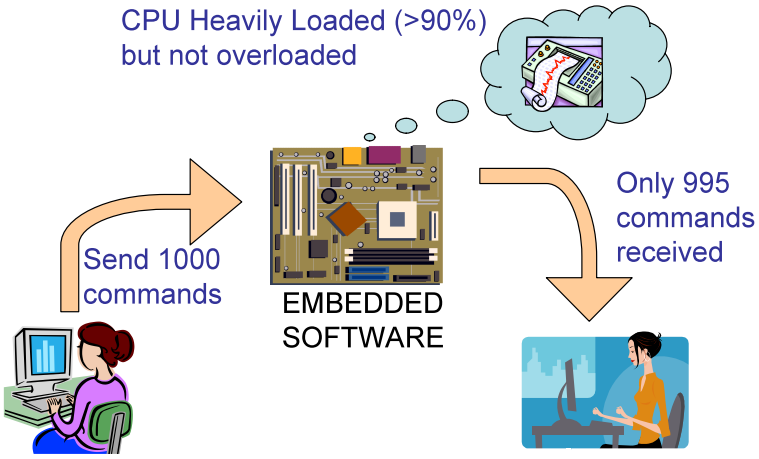


Figure 2 Software missed receiving some commands

The investigation revealed a problem with task priorities (see Figure 3). There was a Command Processing Task designed to read the commands from the hardware buffer. The buffer had to be serviced every 32 milliseconds by the task to avoid an overrun, thus, this was a hard deadline. A second task, the Health Monitor Task, was running at a higher priority and periodically preempting the first task. This higher priority task was responsible for performing all of the autonomous health and safety monitoring on the spacecraft and initiating corrective actions in the event of a fault. However, the Health Monitor Task ran only once per second and had a longer real-time deadline (one second); the task simply had to finish execution before it was time for that task to run again.

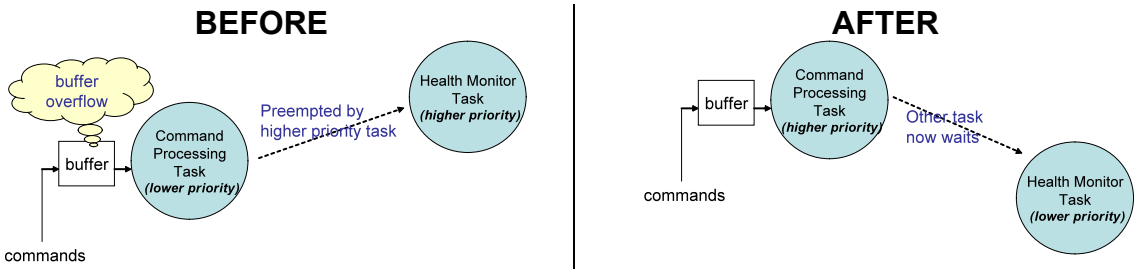


Figure 3 Interaction between Command Processing and Health Monitor Tasks

Investigation revealed that the software designers fell into a common trap of assigning a higher priority to an “important” task (like monitoring spacecraft health) rather than assigning the higher priority to the task with the shortest real-time deadline. This allowed the Command Processing Task to be starved and it occasionally missed servicing the buffer resulting in dropped commands.

When the priorities were corrected and the same subtest repeated, all 1000 commands were successfully received and both tasks met their real-time deadlines despite no overall change in CPU loading.

3.2 Case #2: Processor Reset when Available Memory Buffers Were Exhausted

A subtest was created which maximized data input rates so that the software would exhaust all available memory. The test targeted a software implementation that included a pool of fixed size memory buffers that were allocated and freed by tasks in the system. Generally, tasks allocated buffers to store and process input data and then released the buffers when processing was complete. When uncharacteristically high inputs were driven into the system, the software allocated buffers more quickly than it could free them, eventually driving to a state with no free buffers (see Figure 4). It was expected that the software would then drop input data (since buffers were not available) but continue to execute in a degraded mode of operation. Instead, what was observed was that when the buffers were exhausted, the processor reset.

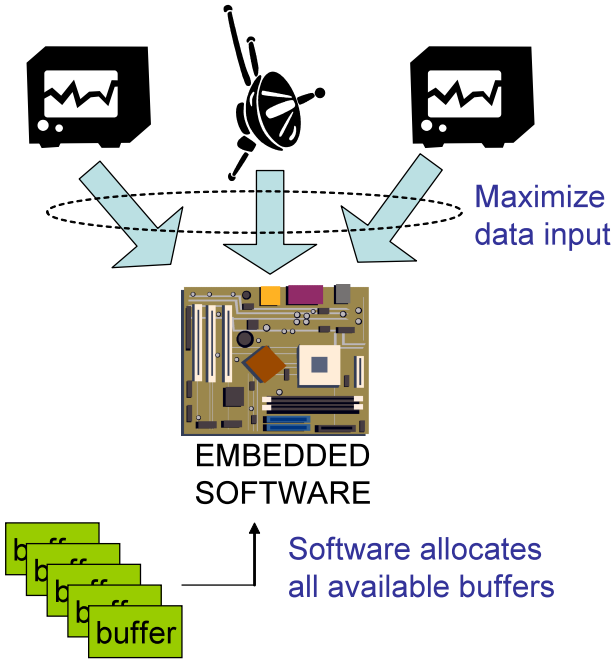


Figure 4 Maximize input data rate to exhaust memory

Investigation uncovered a classic programming error where a return value was not being checked (see Figure 5). A utility **GetBuffer()** function was used throughout the software to allocate a buffer from the fixed size pool. The function provided a pointer to the buffer as well as a return code to indicate whether a buffer was available. In one instance in the software, the return code was not checked and the resulting "null" buffer was used causing arbitrary memory to be overwritten leading to a watchdog reset. Once the problem was isolated, the remedy was to check the return code before using the buffer.

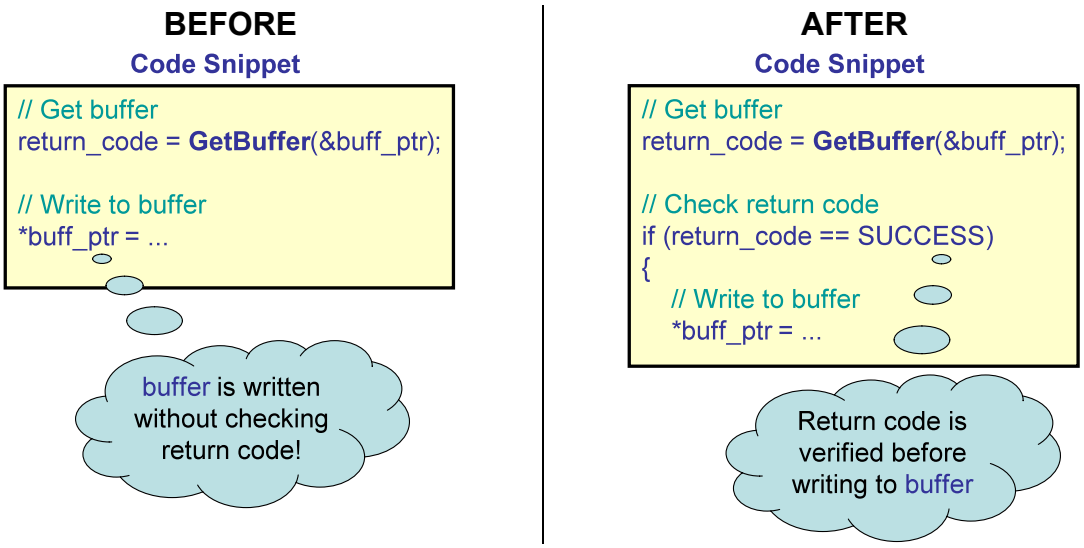


Figure 5 Checking return code of a call to GetBuffer()

At a first glance, one may wonder why this was not caught in unit testing. The problem is that the unit test focuses on paths of execution. If the developer forgets to implement the error checking logic, he will not test for it because there is no error path to test.

It should be noted that there were 65 instances of calls to **GetBuffer()** throughout the software and this was the only instance that did not have the proper error checking. By stressing the software, this single stress test validated the error handling of all 65 invocations of **GetBuffer()**.

Once the coding error was corrected and the test repeated, the desired graceful degradation was observed – some input data was lost but the software continued to operate (i.e., robustness). When input data rates were returned to nominal levels, full software operation was restored (i.e., elasticity).

3.3 Case #3: Unexpected Command Rejection When CPU Was Heavily Loaded

A stress subtest was devised that sent certain commands to the software while the CPU was heavily loaded but not overloaded. When the software receives these commands it performs an internal consistency check on the command fields before accepting it for further processing. Intermittently, the software was rejecting some of the commands because they failed this consistency check which indicated that the command was garbled. When the exact same command was later sent to the software, it was accepted by the software as valid. The problem appeared to be random.

The investigation into the behavior revealed that two tasks were using the same unprotected shared memory resource (see Figure 6). The Command Processing

Task used a Stack data structure in memory to validate the command data arguments. A higher priority Health Monitor Task used the same data structure to validate previously loaded health monitor checks. Occasionally, the higher priority Health Monitor Task would preempt the Command Processing Task while it was in the middle of validating a command. The higher priority task would "corrupt" the Stack, thus when the Command Processing Task resumed after preemption, it appeared that the command that it was processing was invalid.

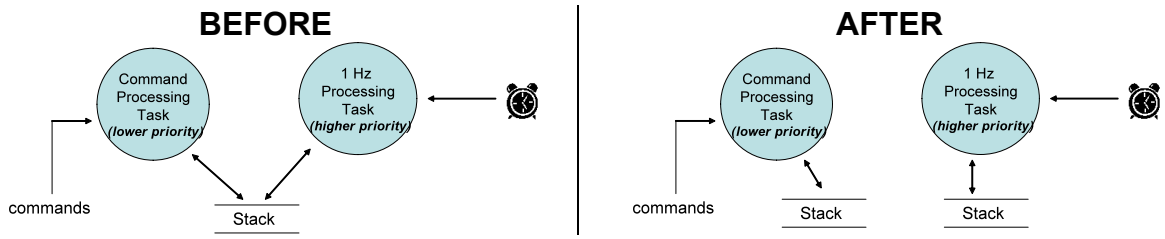


Figure 6 Tasks accessing an unprotected resource

The root cause was further masked by the fact that the two tasks actually called a common function that performed the validation check and did not manipulate the Stack data structure directly. This common function was not reentrant because it used a static Stack data structure.

The resolution was to separate the common function into two functions, each with its own Stack data structure. The subtest was repeated and the software worked as expected with no rejected commands.

This is an example of a flaw that was inherent in the implementation and could have struck at anytime regardless of CPU load. However, it is more likely to strike when the CPU is loaded because there are more opportunities for the two tasks to interact. That is why the problem presented itself so easily under stress conditions. This type of problem is rarely discovered in unit testing because unit tests are typically run in their own thread of execution and reentrancy problems will rarely reveal themselves in single-threaded environments. It's only in a multitasking environment with frequent task preemptions that a problem like this will rise to the surface.

3.4 Case #4: Processor Reset When RAM Disk Was Nearly Full

The software supported a 1 Gigabyte (GB) RAM disk. The software ingested high speed scientific data from an imaging instrument and stored this data onto the disk. A stress subtest was designed to generate image data until the RAM disk was completely filled (see Figure 7). This is analogous to snapping pictures with a digital camera until the memory card is full. It was expected, once the disk was full, that the software would then drop subsequent image data but otherwise continue normal operations. However, when the test was executed, the

processor unexpectedly reset when the RAM disk reached about 98% of its capacity.

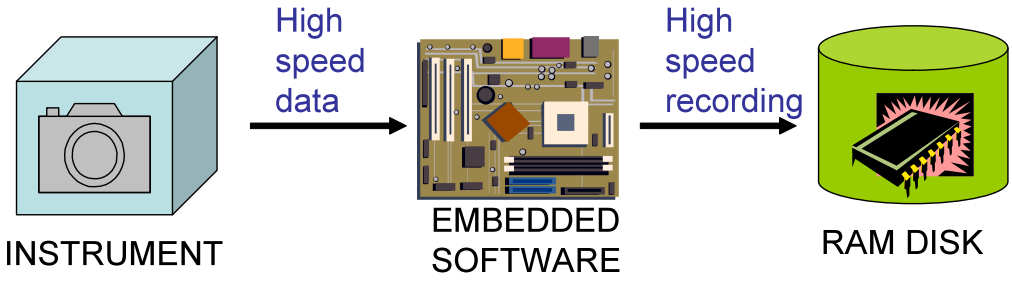


Figure 7 Recording to RAM disk until completely full

The software and underlying operating system organized the RAM disk into 65,536 clusters of equal size (16 Kilobytes per Cluster). Each cluster was marked by the operating system as *used* or *free* (See Figure 8). When the software needed to find a new cluster to store data it executed a system call that returned the address of a free cluster.

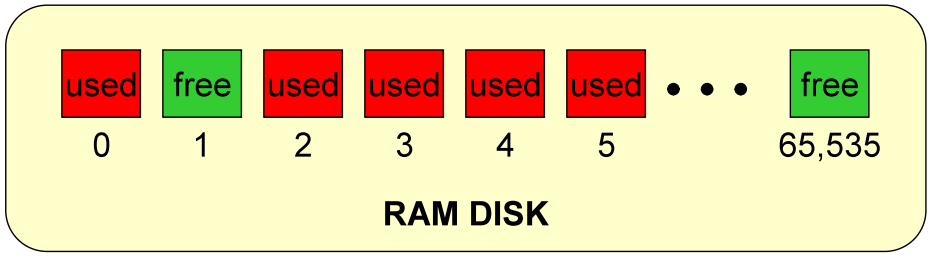


Figure 8 Organization of RAM Disk into clusters

Analysis showed that the search time to find a free cluster grew nearly exponentially as the RAM disk usage approached capacity. This was because the algorithm employed to find a free cluster performed a linear search checking cluster after cluster until it discovered one marked free. Figure 9 shows the average number of attempts (***N***) required to find a free cluster as a function of RAM disk percentage full (***p***) using this inefficient brute force search algorithm. The number of attempts, and thus the time required to find a free cluster, rises dramatically around the 90% usage level. It requires an average of 10 attempts to find a free cluster when the RAM disk is 90% used, 50 attempts at 98% used and a whopping 32,768 attempts when only one free cluster remains! Thus, search times that took hundreds of microseconds on a nearly empty RAM disk were now measured in seconds when the RAM disk was nearly full.

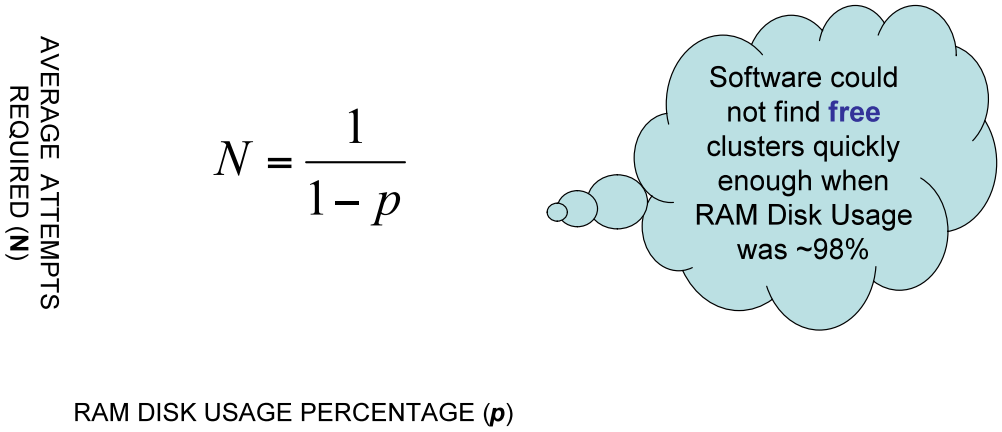


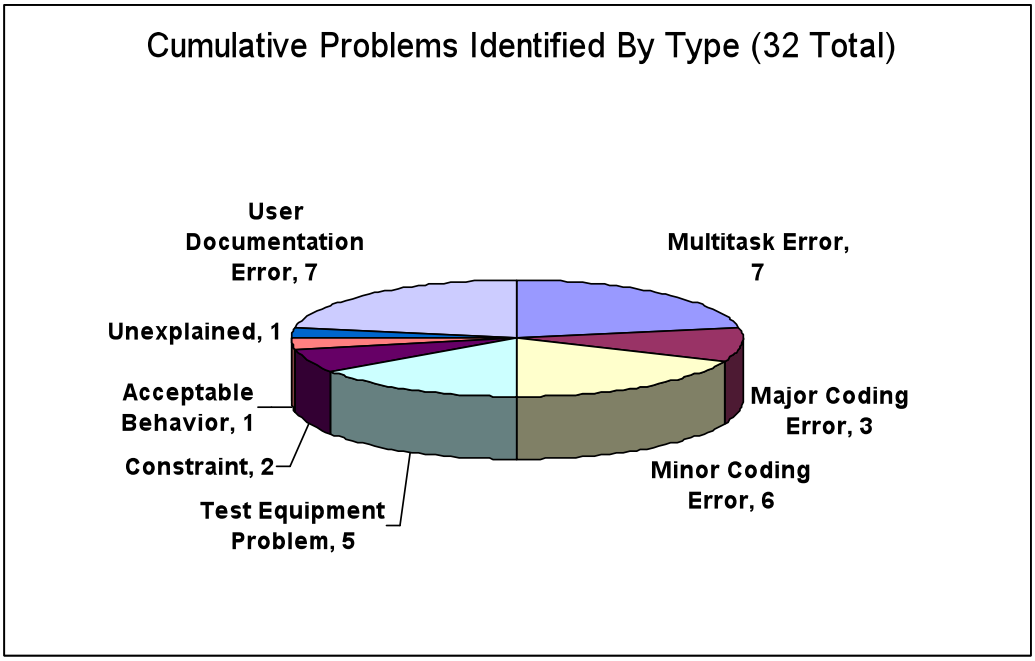
Figure 9 Number of attempts required to find a free cluster

Thus it follows that during the subtest as the RAM disk was filling, the search time to find a free cluster took longer and longer, eventually causing other tasks to be starved, leading to a watchdog reset.

This search algorithm, however, was in the operating system rather than the software application code. Although the team had access to the operating system source code as well as an approach for a much improved search algorithm, they were reluctant to change the operating system late in the software development cycle. Instead, an operational constraint was established to maintain the RAM disk at no more than 95% used. No change was made to the software.

3.5 Synopsis of All Problems Found During Stress Testing

As mentioned earlier, 32 problems were formally reported from stress testing across the three software development programs. Test engineers were encouraged to document problems regardless of root cause. As such, the total includes problems that were subsequently determined to be non-software issues. The 32 problems are organized by type and presented in Figure 10.



Problem Type	Definition
Multitask Errors	Software Errors attributed to complexities of a multitasking environment such as: • Tasks that starve other tasks (causing missed real-time deadlines or watchdog resets) • Omitting semaphore protection around shared resources • Non-reentrant procedures • Deadlocks • Priority Inversion • Race Conditions
Major Coding Errors	Software Bugs that result in unpredictable or undesirable execution such as: • Referencing uninitialized variables • Missing 'break' statement in a C-Language case statement
Minor Coding Errors	Software Bugs with minimal operational consequence such as: • Reporting wrong ID in an anomaly message • Incrementing wrong error counter
Acceptable Behavior	Observing degraded operation while a constraint is violated. Example: • Sending 15 commands per second (when software is designed to accept 10 commands per second) results in software missing some commands.
Constraint	Causing an unrecoverable problem when constraint is violated. Example: • Sending 15 commands per second (when software is designed to accept 10 commands per second) causes processor to reset
User Documentation Error	Discrepancies between user documentation and actual software implementation
Test Equipment Problem	Problems that originally appear to be software issues but are later attributed to failure of test equipment
Unexplained	Problems that occur whose root cause cannot be identified. Typically the problem is not reproducible.

Figure 10 Classification of all problems identified during Stress Testing

3.5.1 Software Problems Identified by Stress Testing

The types identified on the right hand side of the pie chart address 16 of the 32 problems and demonstrate those whose root cause is an error in the software design or implementation. All of these errors resulted in modifications to the software to correct the issue.

The most prevalent type of software error identified during stress testing was **multitask errors**. There were seven such errors across the three programs. These are errors attributed to the complexities of task interconnections in a multitasking environment. This result is not surprising given that stress tests "excite" the interactions between tasks and increase the likelihood of missed real-time deadlines, deadlocks, race conditions, reentrancy issues and other multitasking related errors Two of the four cases presented earlier (incorrect task priorities and non-reentrant code) fall into this category.

A second type of software error was **major coding errors** and three such errors were identified. These are logic errors in the code that result in unexpected or undesirable behavior and three such instances were found during the stress testing programs. The earlier case where a return value was not checked before a buffer was accessed is an example of major coding error.

Also, six **minor software errors** were found. These represent errors with minimal operational consequence such as reporting the wrong ID code in a log message or incrementing the wrong error counter. Since stress testing often executes off-nominal paths through the code, it follows that these types of problems are exposed in these tests.

3.5.2 Other Problems Identified by Stress Testing

The left hand side of the pie chart reveals problems whose root cause was attributed to something other than the software under test. This is not unusual since the root cause of an apparent test problem is not always immediately known.

There were seven instances of **user documentation errors** discovered during stress tests. These were inconsistencies between the user documentation and the actual software implementation that resulted in an update to the document.

Five **test equipment problems** were noted which covered problems that were isolated to the support equipment used to run the tests and inject faults.

There was one case of **acceptable behavior** where further analysis deemed that the degraded behavior of the system under stress was acceptable.

There were two cases where stress testing uncovered a previously unrealized **constraint** on the system. The case presented earlier where the processor reset

when the RAM disk was nearly full is an example of a problem identified as a constraint. Recall that no software change was made, however, an operational constraint was established to document the limitation.

Finally, there was one instance of an **unexplained** problem where the processor unexpectedly reset. The problem could not be duplicated despite repeating the same test multiple times. Based on available data it is believed that the problem was related to an anomaly with test equipment, however no definitive determination could be made.

4. SUMMARY

Software Stress Testing should be an essential component for any critical embedded software development program. While typically not written as a formal requirement, users have an expectation that the software demonstrates the characteristics of robustness and elasticity in response to any user actions. Furthermore, stress testing can expose design flaws and software bugs that are not easily uncovered using traditional testing methods. The problems uncovered in stress testing often involve the complex interactions between tasks such as missed real-time deadlines, deadlocks, race conditions, and reentrancy issues. A formal and rigorous approach to Software Stress Testing can uncover serious problems before the software is released into the user community.

REFERENCES

[LEHOCZKY01] J. Lehoczky, L. Sha and Y. Ding, The Rate monotonic scheduling algorithm: exact characterization and average case behavior, *IEEE Real-Time Systems Symposium*, pp. 166-171, December 1989.