Flight Software Stress Testing

5th Annual Spacecraft Flight Software Workshop
October 19 – 21, 2011

T. Adrian Hill

Johns Hopkins University Applied Physics Laboratory

Adrian.Hill@jhuapl.edu

What Is Software Stress Testing?

- Software Stress Testing is intentionally subjecting software to unrealistic loads while denying it critical system resources
  - Testing should target known weaknesses and vulnerabilities in the design

- Software performance and behavior are measured during the test
  - Degraded software operation is acceptable during a stress test.
    - The software should handle the degraded operation gracefully and should fully recover when the unrealistic load is removed.

How Is Stress Testing different from Traditional Acceptance Testing?

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

Why Is Stress Testing important?

- Stress Testing will validate the **robustness** and **elasticity** of the design
  - **ro·bust·ness** – *noun*, the property of being powerfully built or sturdy. One step below bulletproof.
  - **e·las·tic·i·ty** – *noun*, the property of returning to an initial form or state following deformation.

- Stress Testing will expose design and implementation flaws that often remain hidden under traditional testing.

JHU/APL Spacecraft Missions Included In This Study

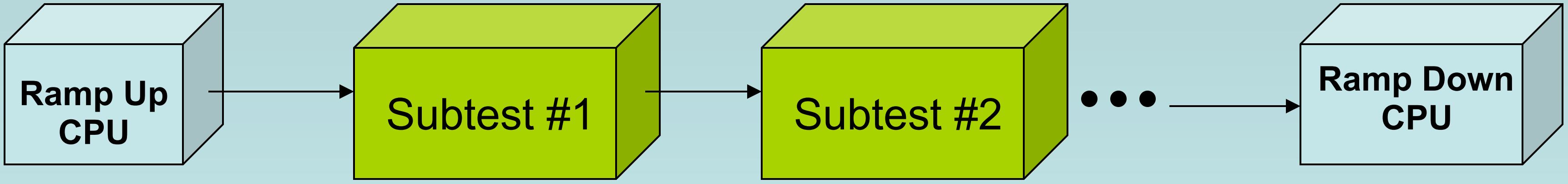
- **<u>ME</u>rcury <u>S</u>urface, <u>S</u>pace <u>EN</u>vironment, <u>GE</u>ochemistry, and <u>R</u>anging (MESSENGER)**
  - Launched: August 2004
  - Duration: 8 years

- **New Horizons (Pluto-Kuiper Belt Mission)**
  - Launched: January 2006
  - Duration: 9+ years

- **<u>S</u>olar <u>TE</u>rrestrial <u>RE</u>lations <u>O</u>bservatory (STEREO)**
  - Launched: October 2006
  - Duration: 2+ years

Approach Used for Stress Tests Executed on the Spacecraft Software

- Stress Tests typically had fairly long duration (12 to 72 hours)
- A baseline "steady state" CPU loading of around 90% was established at the start of the test
- A number of subtests were executed against the Flight Software while the CPU was loaded to further stress the system. Sub-tests included:
  - Maximizing I/O data rates
  - Maximizing data bus usage
  - Maximizing interrupt rates
  - Exhausting available memory
  - Overflowing queues



Characteristics of the Stress Test

- Stress Tests were scripted and generally automated
  - This allowed tests to be repeatable

- Post-test analysis was performed to identify unexpected anomalies that may have occurred during a test
  - Checklists were used to identify and ensure all necessary data points were verified.

- All problems were formally reported into a Problem Tracking System

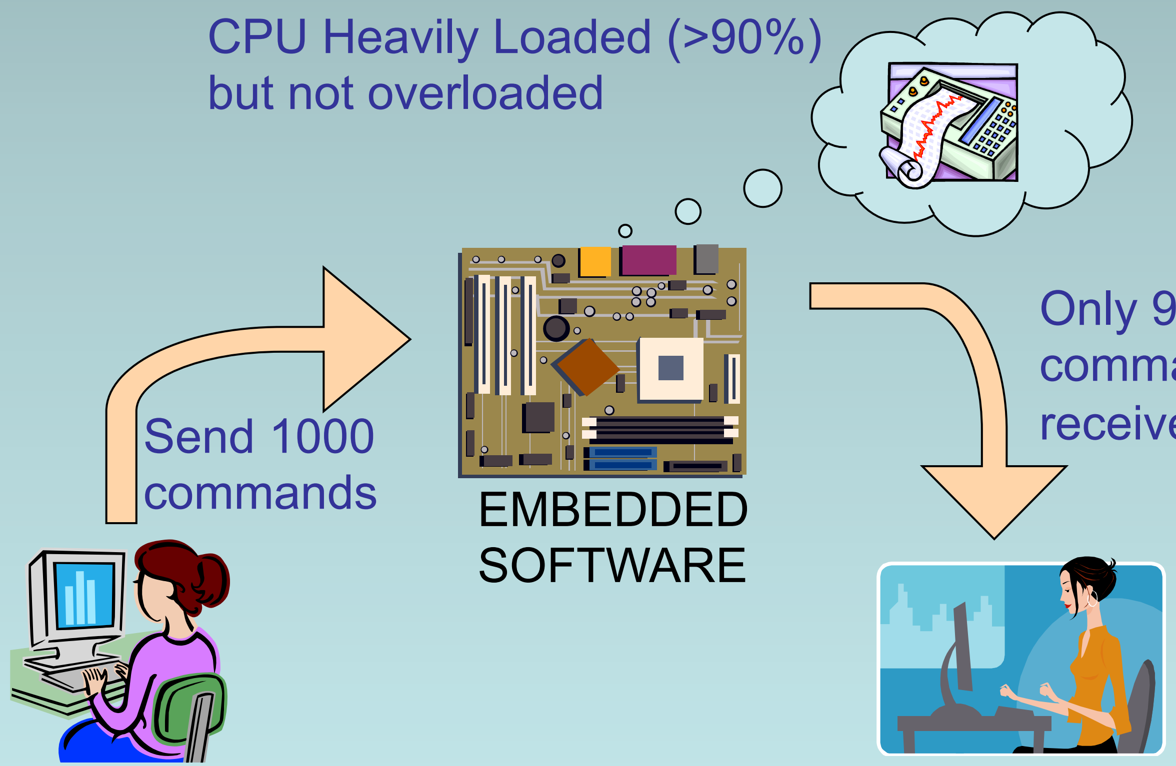
Real Problems Found During Stress Testing

- Five examples are provided identifying real problems found during Software Stress Testing

- For each example, the following is provided
  - **Test Case**
    - Brief description of the test and report of the problem
  - **Investigation Results**
    - Results of the investigation
    - Identification of root cause
  - **Resolution**
    - Summary of how problem was addressed

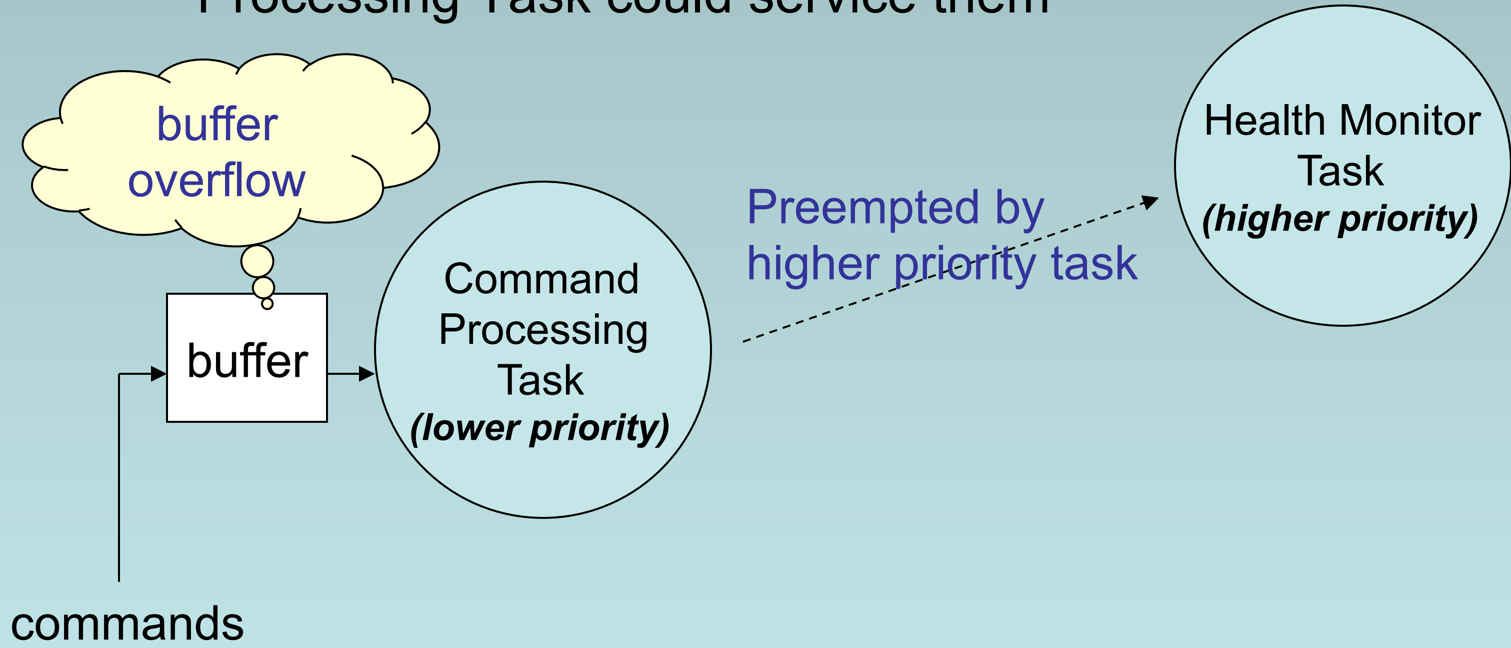
Example #1
Software Missed Receiving Commands when CPU Was Heavily Loaded

- Test Case
  - A series of commands was sent while the CPU was heavily loaded
  - A few of the commands were unexpectedly dropped by the software



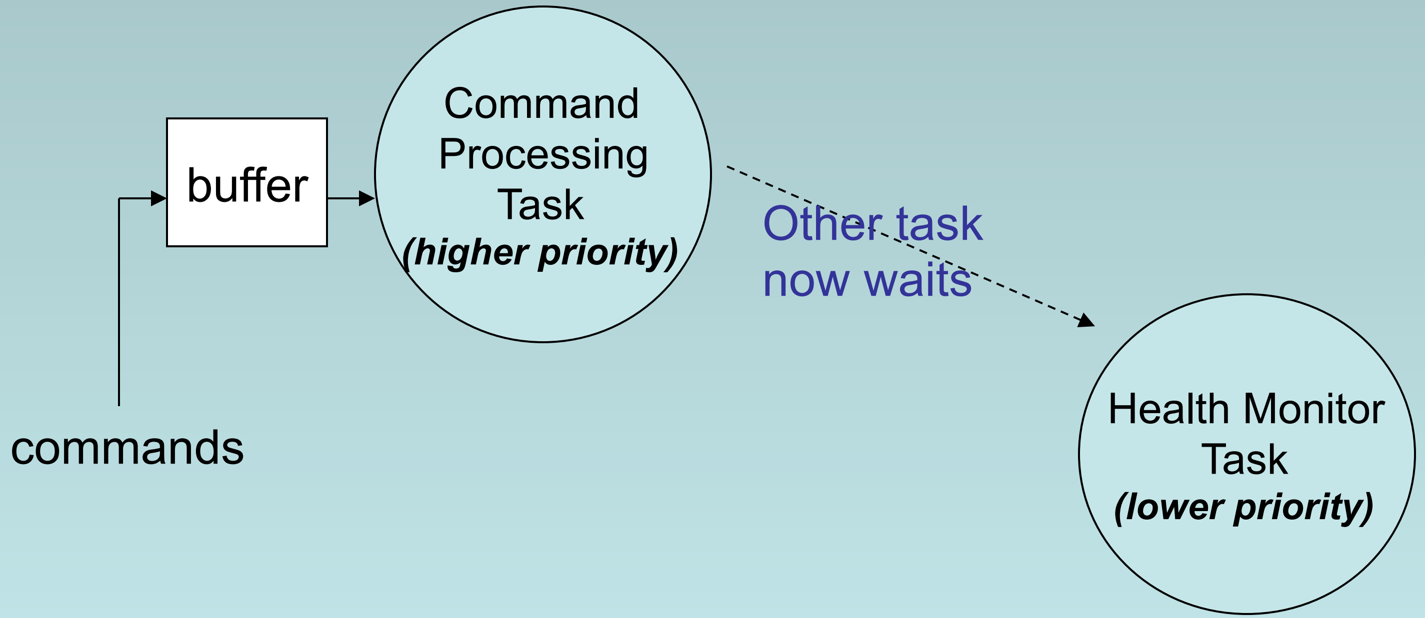
Example #1
Software Missed Receiving Commands when CPU Was Heavily Loaded

- Investigation Results
  - A task with real time deadlines was assigned a lower priority than a less time-critical task
  - This caused real time buffers to overrun before Uplink Processing Task could service them



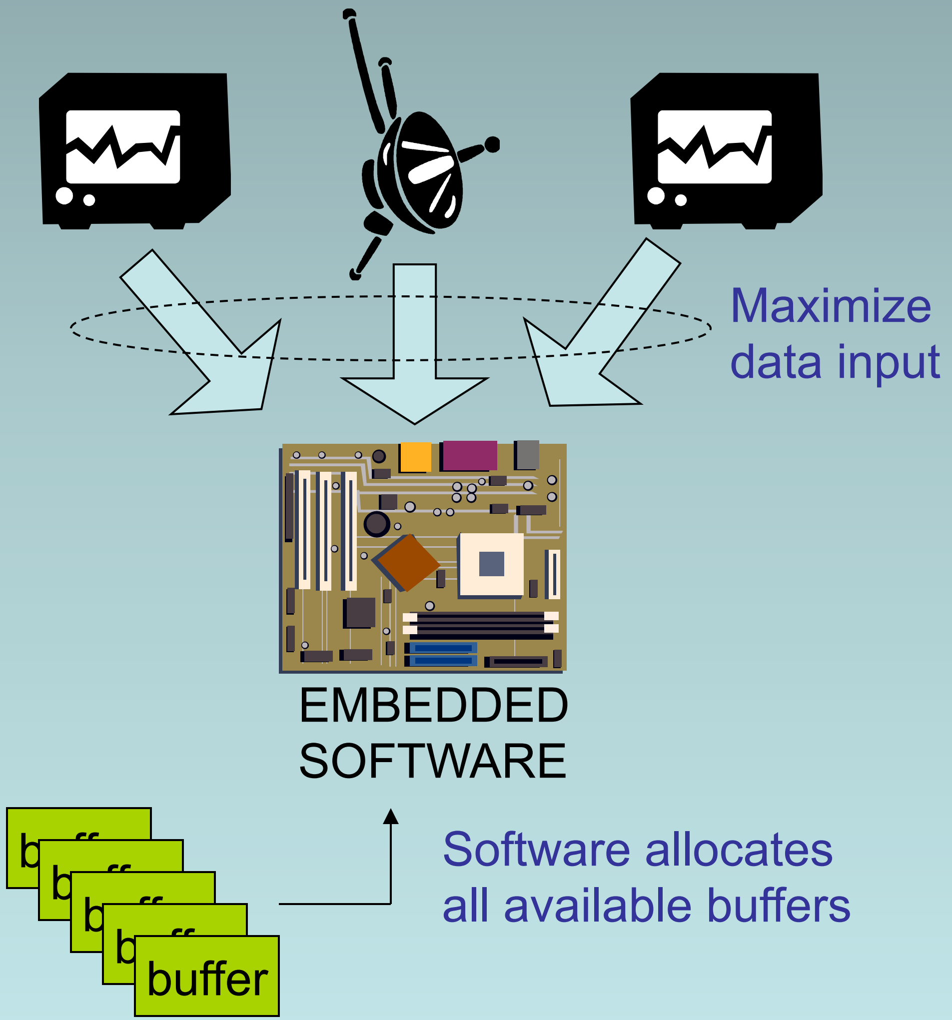
Example #1
Software Missed Receiving Commands when CPU Was Heavily Loaded

- Resolution
  - Priorities were reassigned so that task with real time deadline ran at a higher priority
  - Software now receives all commands when CPU is heavily loaded with no degradation in performance



Example #2
Processor Reset when Available Memory Buffers Were Exhausted

- Test Case
  - Data input was maximized so that software would exhaust all available buffers used to store data
  - Test caused the processor to unexpectedly reset (watchdog timeout) when memory was exhausted



Example #2
Processor Reset when Available Memory Buffers Were Exhausted

- Investigation Results
  - There was one instance in the software where a buffer was used without verifying a return code
    - This caused the software to overwrite random memory leading to a watchdog reset
    - This was the only instance (out of 65 calls to **GetBuffer**) that did not check the return code
  - This error would not be detected in unit testing or “nominal” testing

Code Snippet

```
// Get buffer
return_code = GetBuffer(&buff_ptr);

// Write to buffer
*buff_ptr = ...
```

buffer is written without checking return code!

Example #2
Processor Reset when Available Memory Buffers Were Exhausted

- Resolution
  - Simple software change was made to add error check
  - Software now continues execution (degraded) when buffers are exhausted and recovers when buffers are again available

Code Snippet

```
// Get buffer
return_code = GetBuffer(&buff_ptr);

// Check return code
if (return_code == SUCCESS)
{
   // Write to buffer
   *buff_ptr = ...
```

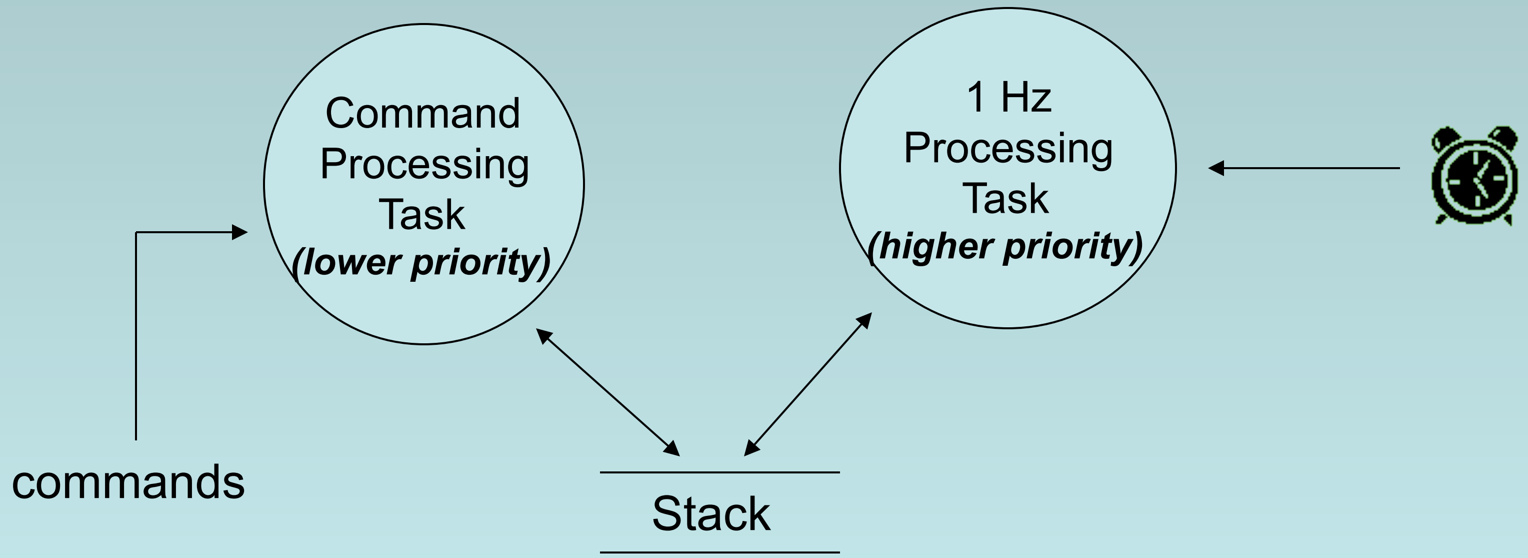
Return code is verified before writing to buffer

Example #3
Unexpected Command Rejection when CPU Was Heavily Loaded

- Test Case
  - A series of commands were sent to the software when the CPU was heavily loaded (but not overloaded)
  - The software intermittently rejected some of the commands reporting that the command was garbled
    - The commands were failing an internal consistency check performed by the software
  - The exact same commands would be later be accepted by the software (problem appeared to be random)

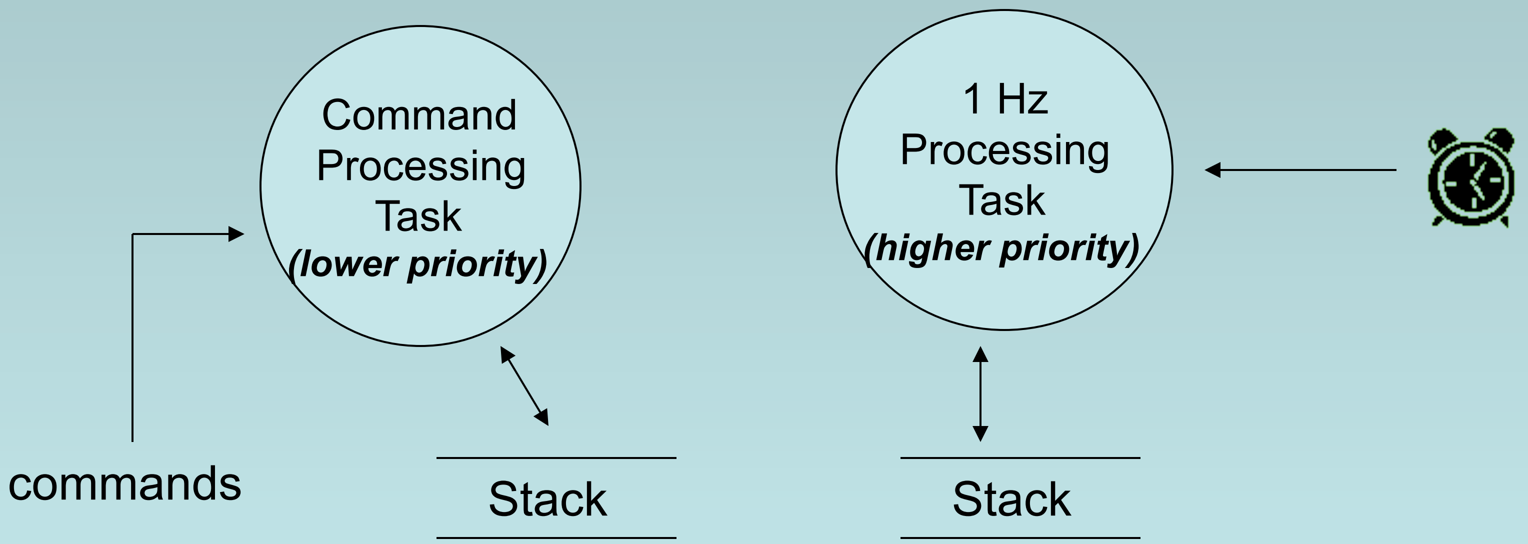
Example #3
Unexpected Command Rejection when CPU Was Heavily Loaded

- Investigation Results
  - Two tasks using same unprotected shared memory resource
  - 1 Hz Processing Task was higher priority and could preempt Command Processing Task and corrupt the stack
    - Caused Command Processing Task to occasionally compute a "garbled" result
  - Unlikely that this problem would be caught if CPU was not heavily loaded



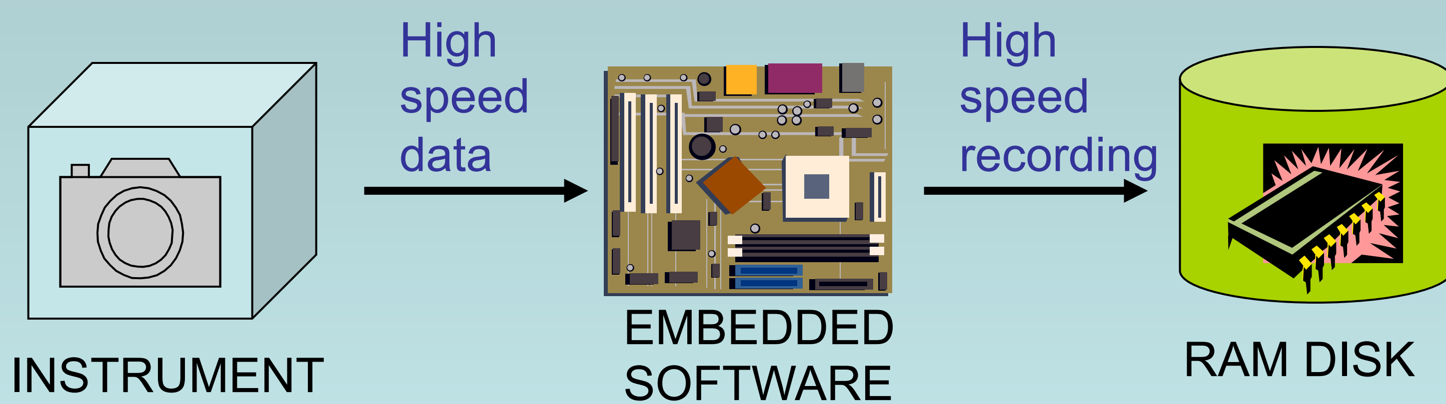
Example #3
Unexpected Command Rejection when CPU Was Heavily Loaded

- Resolution
  - Changed software so that each task instantiated its own version of the stack
  - Command Processing Task no longer has its stack data corrupted



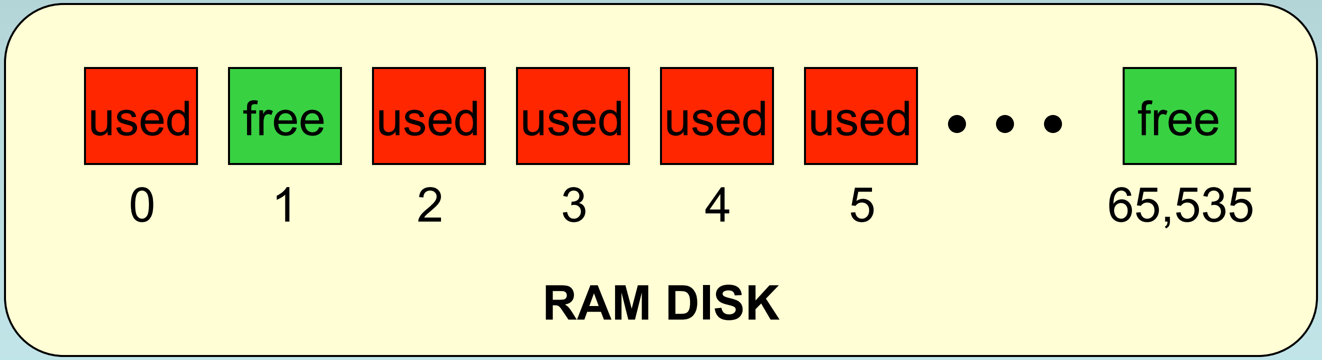
Example #4
Processor Reset when RAM Disk Was Nearly Full

- Test Case
  - System included a Solid State Recorder (RAM Disk) used to store scientific measurements
  - Software ingested high speed data from imaging instrument and stored it on the RAM Disk
  - Test caused the processor to unexpectedly reset when the RAM Disk usage was near capacity (~98%)



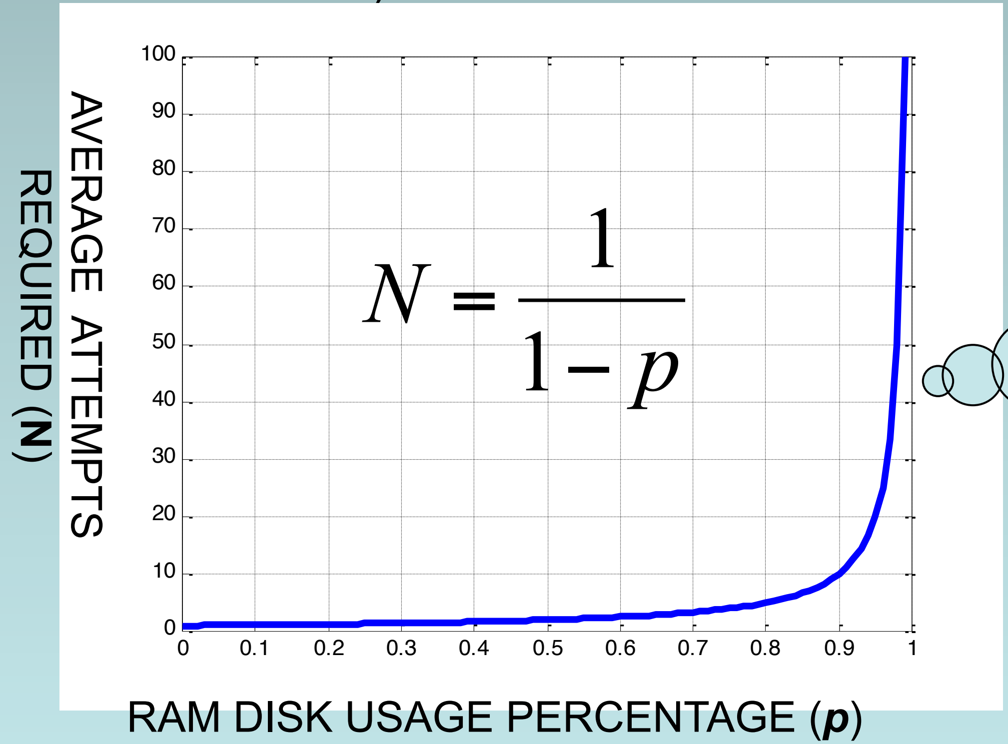
Example #4
Processor Reset when RAM Disk Was Nearly Full

- Investigation Results
  - Software logically organized RAM Disk into 65,536 clusters of equal size
    - Each cluster was marked as **used** or **free**
  - When the software needed to find a new cluster, it sequentially searched the RAM Disk, cluster by cluster, until it found one marked **free**
  - When the RAM Disk was nearly full, this search took too long and starved other critical tasks (causing a watchdog reset)



Example #4
Processor Reset when RAM Disk Was Nearly Full

- Investigation Results (cont)
  - Time required to find a free cluster grows exponentially when RAM Disk is nearly full
    - This can be proven mathematically (assuming free clusters are randomly distributed)



$$N = \frac{1}{1-p}$$

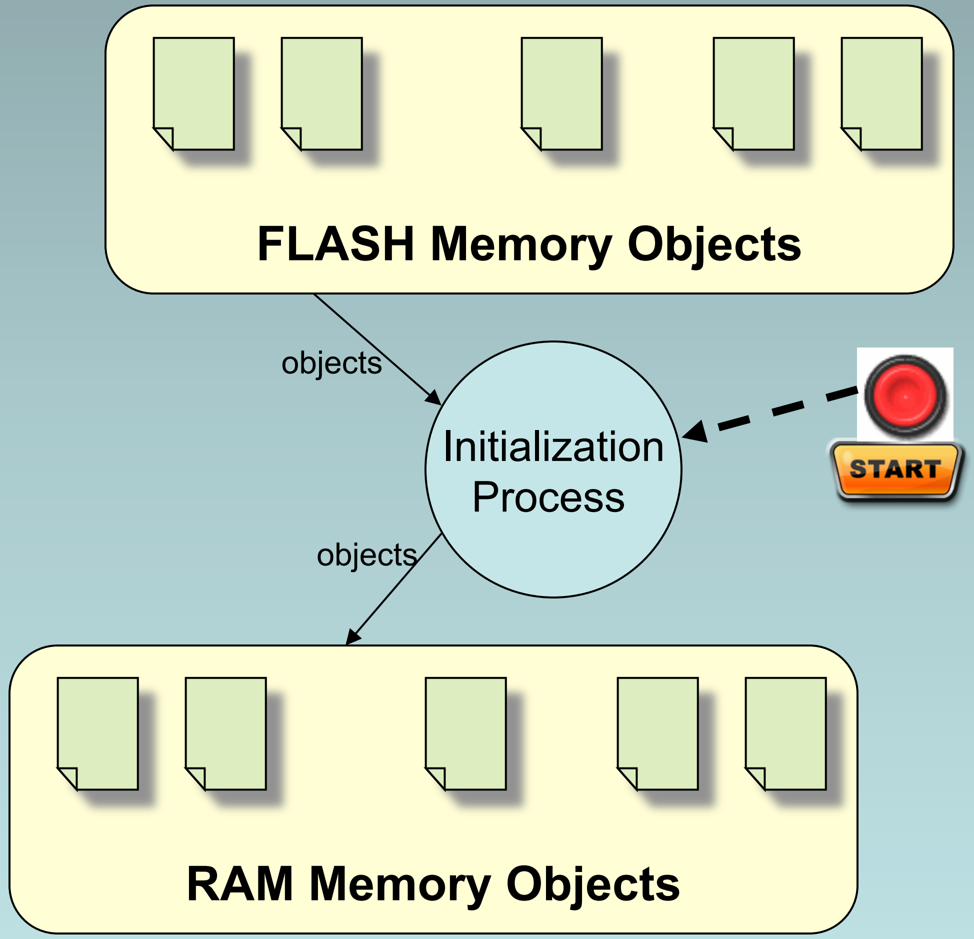
Software could not find **free** clusters quickly enough when RAM Disk Usage was ~98%

Example #4
Processor Reset when RAM Disk Was Nearly Full

- Resolution
  - No easy software solution was available
  - Operational Constraint was written to not exceed 95% of capacity on RAM Disk
  - No software change was made to enforce this constraint or improve the performance

Example #5
Processor Hung in Endless Reset Loop When FLASH Memory Was Fully Loaded

- Test Case
  - Non-volatile FLASH memory was preloaded with user-configurable objects
  - These objects allow the user to store custom configuration commands and data.
  - At initialization, the software copies the objects from FLASH to RAM
  - When all available FLASH objects were preloaded, the processor appeared to hang indefinitely



Example #5
Processor Hung in Endless Reset Loop When FLASH Memory Was Fully Loaded

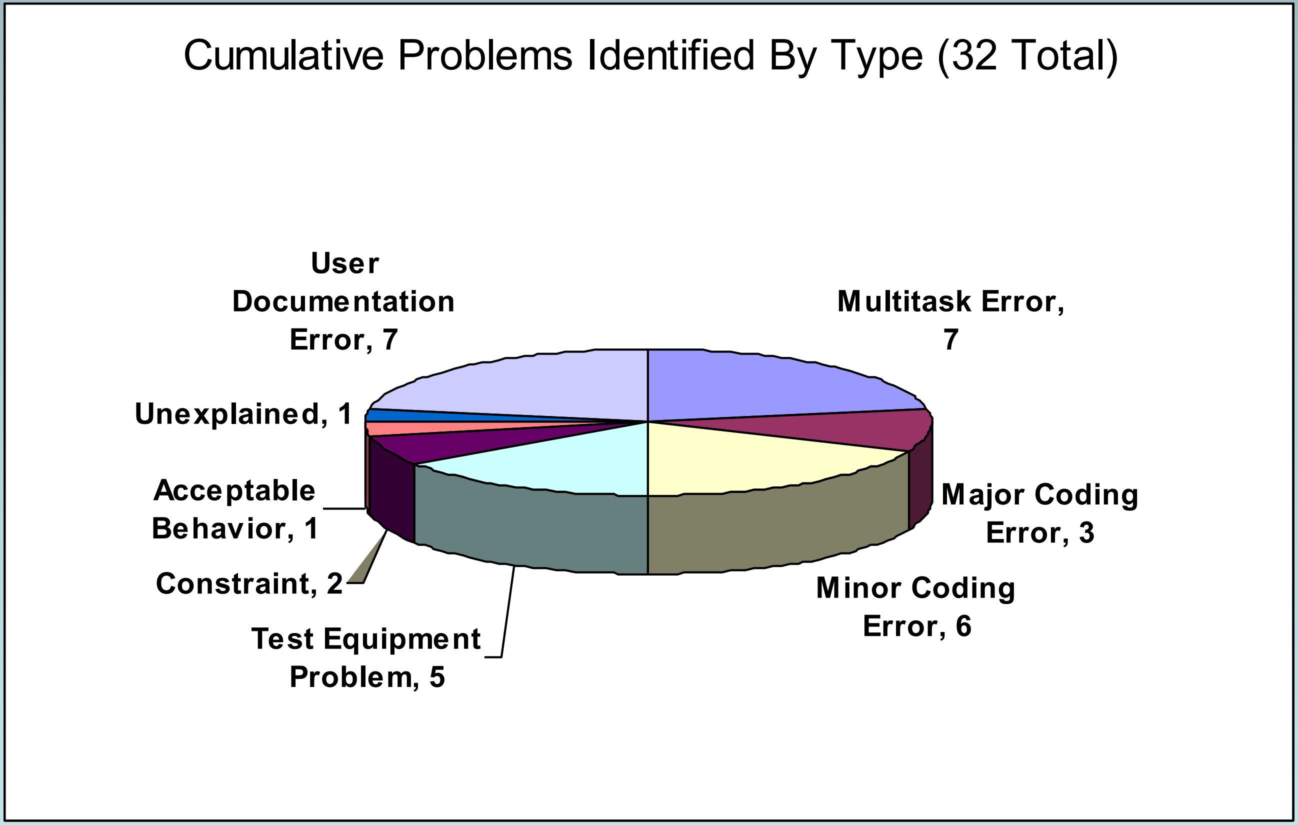
- Investigation Results
  - The software initialization process copies objects from FLASH to RAM
    - As it copies each object, it performs a consistency check to validate the object
  - The duration of the software initialization increased as a function of the number of objects in FLASH
  - When the maximum number of objects was loaded, a watchdog timer expired resetting the processor before the initialization was complete.
  - This caused the process to remain in an endless reset loop because after each reset the processor would re-attempt to copy and validate the objects from FLASH to RAM

Example #5
Processor Hung in Endless Reset Loop When FLASH Memory Was Fully Loaded

- Resolution
  - Additional calls to service the watchdog timer were strategically inserted in the software initialization processes
  - Once updated, the software initialization completed even when FLASH memory was fully loaded

Problems Identified by Stress Testing -- Cumulative Total --

- The Software Stress Testing across the three missions produced 32 Problem Reports. The 32 problems have been analyzed and are categorized as follows:



Problems Identified by Stress Testing
-- Types Explained --

- Multitask Errors
  - Software Errors attributed to complexities of a multitasking environment such as
    - Tasks that starve other tasks (causing missed realtime deadlines or watchdog resets)
    - Omitting semaphore protection around shared resources
    - Memory Leaks
    - Deadlocks
    - Priority Inversion
    - Race Conditions

Problems Identified by Stress Testing
-- Types Explained --

- 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
  - Software Implementation that does not meet requirements

- Minor Coding Errors
  - Software Bugs with minimal operational consequence such as:
    - Reporting wrong ID in an anomaly message
    - Incrementing wrong error counter

Problems Identified by Stress Testing
-- Types Explained --

- 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.

- Constraints
  - 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

Problems Identified by Stress Testing
-- Types Explained --

- User Documentation Errors
  - Discrepancies between user documentation and actual software implementation
- Test Equipment Problems
  - 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.

Summary

- Stress Testing is a vital component in the validation of any critical Embedded Software System.
- Stress Tests provide a measure of a software's robustness (ability to operate under stress) and elasticity (ability to recover after stress)
- Stress Tests uncover design and implementation errors that are not easily discovered through other testing mechanisms

Thank You