# RTES Assignemnt 5

Gerritt Luoma

## Problem 1

> Research, identify and briefly describe the 3 worst software and hardware computer engineering design flaws and/or software defects of all time

> A: Some options for product failures are Blackberry Storm, Windows Genuine Advantage, Windows 8, Windows ME, Apple Lisa, Pentium FPU bug

- Not picking any of these. Section is handled in part C.

> B: Some options for mission failures include: NORAD false alarms, Mars Express Beagle 2, Challenger and Columbia Shuttle Loss, and Boeing MCAS design flaws more recently corrected summarized in this documentary and fully analyzed.

- Picking NORAD, Mars Beagle 2, and Shuttle losses. Section is handled in part C.

> C: State why the 3 you have selected are the worst in terms of negative impact, negligence, and bad decisions made, but why the failure was not really a real-time or interactive systems design flaw. Rank them from worst to least-worst.

Chosen mission failures:

### NORAD False Alarms

- NORAD False Alarms

The first issue I chose to explore was the set of NORAD False Alarms that were issued on multiple occasions in 1979 and 1980. The North American Aerospace Defense Command, or NORAD, is a joint organization between the United States and Canada that monitors and protects the air sovreignty of North America to provide warnings of potential airborn threats. In November of 1979, an operator intending to test the functionality of the system loaded a test tape while failing to switch the system into a "test mode". This caused a constant stream of false warnings two underground "continuity of government" bunkers as well as international command posts. Twice in June of 1980 a failing communications device caused warnings to flash internationally at US Air Force commands signaling an impending nuclear attack.

In the first incident, the operator failed to follow proper protocl causing the false alarms to be broadcast. In the second incident, the failure was induced by faulty communication hardware leading to the failure. Due to these reasons, the failures aren't due to real-time design flaws or due to interactivity design flaws with the root cause instead being improper training and faulty hardware.

### Mars Beagle 2

- Mars Beagle 2

The second issue that I am investigating is the Mars Beagle 2 mission. The UK-based mission was a Mars astrobiology lander meant to search for signs of past life on the red planet. The lander successfully separated from its transporter and was meant to establish communications with Earth on December 25, 2003 but when communications were attempted, the lander never responded. The true fate of the lander was not known until the Mars Reconnaissance Orbiter, MRO, captured images of the Beagle 2 on the surface. The Beagle had successfully landed but when deploying the solar panels, two of the four panels stuck in their stowed position and never deployed. Unfortunately, these panels obscured the landers communication antennas meaning it could never receive or transmit data successfully. This issue is a hardware problem since the solar panels were unable to deploy meaning it was not a real time or interface problem from a software perspective.

### Challenger and Columbia Disasters

- Space Shuttle Challenger Disaster
- Space Shuttle Columbia Disaster

The final issue that I am reading into is the set of Space Shuttle disasters which were the Challenger breakup during launch in 1986 and the Columbia breakup during reentry in 2003.

The Shuttle Challenger was performing its 10th flight in January of 1986 carrying a crew of 7 individuals consisting of 6 professional astronauts and one teacher. The planned primary objective of the mission was to deploy a satellite that would join a constellation enabling constant communication with orbiting spacecraft. The launch took plae in the morning of an extremely cold, below freezing, morning which was the coldest shuttle launch temperature ever. In previous flights, NASA had noticed that the forward O-Rings of the Solid Rocket Boosters (SRBs) were not providing a perfect seal which was allowing some hot gasses to escape to the cavity between the first and second O-Rings. Deemed a non-critical issue by management but not the engineers, no resesign was made before subsequent launches. Due to the extreme cold on the day of the launch, the O-Rings were providing even less of a seal leading to hot gasses escaping and burning a hole through the SRB. Eventually, the SRB broke away intoducing unexpected torques on the vehicle as well as colliding with the liquid hydrogen tank leading to an explosion causing a complete vehicle breakup. Unfortunately, this lead to an over 2.5 minute free fall of the crew compartment down to the surface of the ocean where the crew most likely passed away.

The Shuttle Columbia disaster was the 113th Space Shuttle mission and the 28th flight for this orbiter. This mission was to deliver the SpaceHab Research Double Module to orbit as well as perform some additional experiments. This mission was originally supposed to fly in January of 2001 but was delayed multiple times until it launched in January of 2003. During the launch, a piece of foam broke away from the External Tank (ET) and struck the left wing of the Columbia at around 500 MPH causing damage to the heat shield of that wing. At the time, neither ground control or the astronauts of the obiter noticed this had even occurred until the second day of the mission. After gathering evidence of the damage to the wing and performing simulations to model the possible damage, engineers noted that the damage could be severe enough to leave the aluminum frame under the heat shield exposed during reentry posing a risk to the safety of the astronauts. This fell on deaf ears as similar models and examples of foam strikes showed that there would be no risk. During reentry, the shuttle began shedding debris starting from the left wing which expanded deventually leading to a complete breakup of the orbiter. It is expected that none of the astronauts were conscious and able to regain consciousness due to a depressurization event around the time of the breakup and all 7 tragically lost their lives throughout the breakup of the orbiter.

Both of these disasters are the result of warnings from engineers/subject matter expers falling on deaf ears, being ignored. With both of these being hardware issues, they are not due to real-time or interface based software issues. Reading through the accounts of both of these disasters opened my eyes to the amount of risk that NASA was open to even just 20 years ago. Risks are necessary for human advancement but there become a point where more caution is needed.

#### Order of Severity

I will order the severity of these issues in two separate ways, first in the order of how they actually occurred, and second in the way that the would have been in the worst-case scenario.

First, in the order that the failures actually occurred, I will rank them as such:

1. Challenger and Columbia Disasters
2. Beagle 2
3. NORAD False Alarms

I have picked this ranking because the two shuttle disasters resulted in the tragic loss of 14 lives, the Beagle 2 mission resulted in the loss of property and scientific data, and the NORAD False Alarms led to the loss of tax payer dollars due to nuclear bombers being scrambled and deployed.

Second, in the order that the worst case of the failures could have occurred, I will rank them as follows:

1. NORAD False Alarms
2. Challenger and Columbia Disasters
3. Beagle 2

The NORAD False Alarms are being moved from lest significant to most significant because it could have led to a true nuclear war and the possible extinction of humanity. While it was historically a set of harmless false alarms that probably cause a little anxiety, if the responding bombers hadn't been called off in time and actually dropped a payload, it would have led to an all out nuclear war leading to mutually assured distruciton between all nuclear powers. The only way the shuttle disasters could have been worse was if the debris led to further loss of life to innocent bystanders, which thankfully didn't happen. The Beagle 2 failure was also essentailly the worst case for the mission because it led to a mission failure. Even if it failed on landing or failed to reach Mars entirely it would have led to the same result.

## Problem 2

> Research, identify and briefly describe the 3 worst real-time mission critical designs errors (and/or implementation errors) of all time.

> A: Some candidates are Three Mile Island, Mars Observer, Ariane 5-501, Cluster spacecraft, Mars Climate Orbiter, ATT 4ESS Upgrade, Therac-25, Toyota ABS Software.

Handled in part C.

> B: Note that Apollo 11 and Mars Pathfinder had anomalies while on mission, but quick thinking and good design helped save those missions and we discuss these in class, so please don't pick these two near failures.

Handled in part C.

> C: State why the systems failed in terms of real-time requirements (deterministic or predictable response requirements) and if a real-time design error can be considered the root cause.

### Toyota ABS Recall

- Toyota ABS Recall

In 2010 multiple reports were submitted to Toyota stating that the braking system of that years' Prius model was not performing as expected. Some even reported that the brakes were behaving in a manner that was causing the user to crash. It was found that in conditions when the ABS system needed to transition from its regenerative braking mode to its hydraulic traditional braking mode, a software logic issue caused there to be a lag of 0.4 seconds before the hydraulics actually kicked in. This caused the vehicle to have no braking at all for the full 0.4 second duration leading to the car not stopping as expected leading to potentially negative outcomes. This violated real time design because it did not meet a deterministic timing guarantee for its state transitions. Instead of there being a logic error in the code, the code simply didn't reach its desired state within a reasonable/responsible amount of time leading to property damage and even some people getting injured.

### Ariane 5-501 Failure

- Ariane 5-501 Failure

The Ariane 5-501 flight failure was the complete loss of the launch vehicle at T+0:39 due to the self desintegration system being trigged caused by high aerodynamic loads on the vehicle. These loads were introduced by an aggressive angle of attack of over 20 degrees caused in part by faulty Intertial Reference Systems (SRIs). For each Onboard Computer (OBC) there are two SRIs running in parallel with one as the active and the other in hot standby. If the OBC detects that the active SRI is in a fault, it will switch to the redundant SRI given it is also operational. This redundant design and the software supporting it is largely the same as the Ariane 4 design and software which led to some reused code between the vehicles, particularly in this area. The aggressively commanded nozzle deflections were commanded due to faulty diagnostic data originating from SRI2 being interpreted as valid data. SRI2 had experienced a software fault converting from a floating point to a 16 bit integer leading to an exception and a stop of its onboard program. The OBC was unable to swap over to SRI1 because it had already experienced the exact same fault during its previous processing cycle (72 millisecond period). The reason this fault occurred was because of an alignment function, which wasn't even used during flight operations, had read values that were out of the regular bounds that the alignment function needed to handle while on the pad. This alignment sequence needed to be active for 50 seconds after launch on the Ariane 4 vehicle but was not necessary for the Ariane 5 but was left in the software to reduce the differences between the two software builds essentially citing the "if it isn't broken, don't fix it" philosophy. Unfortunately, neither SRI was able to be recovered from the fault which led to the eventual loss of the vehicle with multiple real-time deadlines being missed over a long period of time.

This exception caused both indeterministic and unpredictable results breaking rate-monotonic theory and realtime policy.

### Mars Climate Orbiter

- Mars Climate Orbiter

This is one of my favorite famous software falures that I have studied not because of the fact the orbiter might still be somewhere in tthe solar system if it skipped off the atmosphere just right but because of how silly the actual problem was. The nagivation software and the ground software were produced by two separate companies that were developing based off an Interface Control Document (ICD). Somewhere along the way, the company producing the ground software deviated from the ICD accidentally by calculating all measurements and commands in Emperical Units (pound-seconds) while the navigation software remained with the ICD calculating in Metric units (Newton-seconds). This caused a difference with a factor of ~4.45 to occur between what was projected on the ground and what occurred on the orbiter. When it came time to perform the insertion burn navigators began to notice the orbiter was actually way lower than initially inspected with an altitude of ~57km instead of the expected 140km above the surface. At the expected altitude of 140km the orbiter would be able to use the thin Martian atmosphere to perform aerobraking to slow its velocity before ending up in orbit. At the lower altitude, however, the atmosphere was much thicker than expected leading to the orbiter either burning up due to heat generated by the friction or the orbiter skipped off the atmosphere like a pebble on water launching it out of the Mars gravity well leading it to forever drift through space.

Since this is a logic/unit conversion error this does not qualify as a realtime software issue. This instead is an issue of improper testing and validation of the software requirements.

The best part of this issue is that two navigators actually noticed that the altitude might be lower than expected pretty early on but their concerns were dismissed because they hadn't filled out the proper form to document their concerns.

## Problem 3

> The USS Yorktown is reported to have had a real-time interactive system failure after an upgrade to use a distributed Windows NT mission operations support system. Papers on the incident that left the USS Yorktown disabled at sea have been uploaded to Canvas for your review, but please also do your own research on the topic.

> A: Provide a summary of key findings by Gregory Slabodkin as reported in GCN (https://gcn.com/1998/07/software-glitches-leave-navy-smart-ship-dead-inthewater/290995/)

Within Slabodkin's article he explains the September 1997 incident where the USS Yorktown experienced a mass software fault of its Smart Ship technology leading to it losing propulsion making it essentially "dead in the water". The root cause of the incident was an operator writing a "0" into an input field within the database management system which triggered a divide by zero error leading to a buffer overflow. This overflow caused the entire system to go down entirely instead of just being located to the one device the error occurred on. This issue was caused by improper input validation, improper training, and vulnerabilities in their software robustness. This also raised flags of whether Windows NT was the correct tool for the job when something like Unix was readily available.

> B: Can you find any papers written by other authors that disagree with the key findings of “Gregory Slabodkin as reported in GCN” story above? If so, what are the alternate key findings?

Wikipedia states that a civilian Naval engineer stated that the outage was much worse than reported by the navy with the Yorktown requiring a tow back into port with a multiple day fix once there. Atlantic fleet officials claimed that this was one of many "LAN casualties" that occurred on this ship. The civilian engineer also went as far as saying "Using Windows NT, which is known to have some failure modes, on a warship is similar to hoping that luck will be in our favor".

> C: Based on your understanding, describe what you believe to be the root cause of the fatal and near fatal accidents involving this machine and whether the root cause at all involves the operator interface.

I believe that the root cause of this error is two different sets of errors. I believe that the first issue was improper input validation. In no way should data that is input by a human user be capable of crashing not just the current system but the entire LAN environment it is in. This leads to the second issue of how nearly every computer on the LAN went down as well during this failure mode. If you are running distributed systems like this, they need to be stateless and unreliant on the other nodes. This seems to have been exemplified by the use of Windows NT which was a consumer grade OS being used on a warship.

> D: Do you believe that upgrade of the Aegis systems to RedHawk Linux (https://concurrentrt.com/products/software/redhawk-linux/) or another variety of realtime Linux would help reduce operational anomalies and defects compared to adaptation of Windows or use of a traditional RTOS or Cyclic Executive? Please give at least 2 reasons why or why you would or would not recommend Linux.

I would recommend using Linux but not for the input validation issue presented previously. Input validation needs to be handled properly regardless of system and it was a failure on the Navy for allowing user input to create a database crash via divide by zero errors. The true issue lies in how all of the other computers on the network went down as well which could have been due to using Windows NT. I believe that using Linux, which was used more for the navy's usecase and had realtime options, would have helped the Yorktown prevent these errors by imposing more consistent realtime constraints and by providing robustness to the "LAN casualties" observed since it was designed for just that type of job.

## Problem 4

> Form a team of 1, 2, or at most 3 students and propose a final Real-Time prototype, experiment, or design exercise to do as an individual or a small team. Creative Projects are possible if they have real-time requirements and timing constraints but must have multiple services running on one core with clear RM constraints. However, for the summer RT-Systems class, I recommend that in the interest of time you complete the summer Standard Project which is a simple real-time machine vision synchronome (based loosely upon prior creative project requirements for time-lapse monitoring design). The Standard Project challenge details and goals and objectives are provided in these background notes to be graded according to this rubric. You must come up with specific requirements and design to meet the goals and pass the acceptance tests outline in the rubric as MIN, TARGET, and STRETCH goal tests.

> A: Based upon your Creative or Standard project goals and objectives and your understanding of drivers (e.g., the UVC driver and camera systems) used with embedded Linux on your Raspberry Pi, evaluate the services you expect to run in real-time first or any core with 2 or more real-time services. Then, once implemented, plan your runtime verification analysis including methods to test with tracing and profiling to determine how well your design should work for predictable response. Describe specific requirements for your design in this proposal in good detail (e.g., Si, Ti, Di, Ci and functionality of each service, how they share data, what core it runs on, etc. to meet your project objective – Standard or Creative).

Handled in C

> B: You will complete this effort in Lab Exercise #6, making this an extended lab exercise, and ultimately you will be graded using the standard rubric or creative rubric. Either way, you must submit a final report with Lab Exercise #6 as outlined here, but at this point, you are asked simply to propose your solution at this point. You may find the “Final Project” videos on various design options helpful as you come up with your proposal (along with Coursera Lightboard videos).

Handled in C

> C: Your Group should submit a proposal that outlines your response to the requirements, your design concept and methods of modeling and implementation analysis (Cheddar and tracing on Linux). This should include some research with citations (at least 3) or papers read, key methods to be used (from book references), and what you read and consulted.

- Requirements
  - Image requirements
    - Images must be taken from the camera in its native YUYV format
    - Images must be saved in either RGB (PPM) or Grayscale (PGM) formats when written to flash
    - Saved images must not be blurry, duplicated, missed, or otherwise corrupted
  - 1Hz Analog Clock Requirements
    - Program must save images of analog clock at 1Hz with its seconds hand not in motion
    - A total of 1801 images must be captured fulfilling all image requirements. This is a runtime of 00:30:01
    - Images must represent a unique second hand position on the analog clock without any blurring present
    - All images must be saved with a sequence number in the file name and a timestamp in the header
    - Frame capturing must occur in a dedicated service
    - Frame processing and detection must occur in a dedicated service on the same CPU core as the frame capturing service
    - Frame saving to flash must occur in a dedicated best effort service on a separate CPU core from the other two services
  - 10Hz Digital Clock Requirements
    - Program must save images of digital clock at 10Hz capturing each 1/10 second update of the digital clock
    - A total of 1801 images must be captured fulfilling all image requirements. This is a runtime of 00:03:00.1
    - Images must represent a unique 1/10th second on the digital clock without any blurring present on the minute, second, or 1/10th second of the clock
    - All images must be saved with a sequence number in the file name and a timestamp in the header
    - Frame capturing must occur in a dedicated service
    - Frame processing and detection must occur in a dedicated service on the same CPU core as the frame capturing service
    - Frame saving to flash must occur in a dedicated best effort service on a separate CPU core from the other two services

**General Design:**

I will be utilizing a signal-based sequencer running at a frequency resonant with both of the high priority services so that it can release the threads as accurately as possible. This sequencer will use a linux timer and a signal for calling the sequencer. This sequencer will run until all frames have been saved to flash and a stop flag has been raised. The required behavior of the 1Hz and 10Hz program will be selectable via command line input when running the same program. Rate-monotonic analysis will be performed using Cheddar to validate the feasibility of the software system running on the two cores. The `SCHED_FIFO` RT scheduling policy will be used to use standard static priority rate monotonic scheduling. For tracing the timing of the services running in the program, `syslog` will be utilized for outputting timestamps to the logs for retrieval and further timing analysis.

**Analysis and Verification**

I will be using general analysis of the program by using the Liu and Layland paper for calculating the LUB of my services. Aside from writing to flash, I believe my program will be able to easily achieve the requirements of 1Hz and 10Hz processing while remaining under the LUB for two services. I will need to generate the service periods, computation time, etc. to verify the total CPU usage is below the LUB to guarantee it is schedulable.

I will also utilize the Cheddar program to ensure that the two systems are schedulable and verifiable. Once again, will need to determine the deadline, WCET, period, etc. This will allow me to graphically analyze the system and visually inspect when the various services will complete.

The final analysis that I will utilize is post-run analyzation. By extracting the timestamps from my syslogs, I will be able to calculate the average, minimum, and maximum computation times to validate against the expected period/deadline to verify there were no missed deadlines and to calculate the jitter of the program.

> D: Each individual should turn in a paragraph on their role in the project and an outline of what they intend to contribute (specific feature, design, documentation, testing, analysis, coding, drivers, debugging, etc.) and team schedule. For groups of 2 or 3, it is paramount that you specify individual roles and contributions. For those of you working alone, just provide a basic outline of your schedule to complete the project.

I will be completing this project alone. It is due on August 3, 2025. This is 18 days for me to complete the entire project and writeup as well as record all of my demos. Given that I have a deadline for work this week I most likely will not be starting until Saturday, July 19, 2025. This will give me a total of 15 days to complete the project. I will use the first 5 days to complete my static analysis and generate my processing flow charts for the writeup. This will include finalizing service requirements, creating flow charts for my various processing services, and completing the introduction section of the official writeup. The second 5 days will be used for development of the program based off my design completed in the first 5 days. This will be spent implementing the design and validating the performance of the program. The final 5 days will be used analyzing the performance of the program and completing the writeup of the project before recording my final videos and submitting the assignment.