MOST COMMON ERRORS MADE in ECEN 5623 PROJECTS

1. Failure to cover and Check Return Codes - code that calls any API or library function must cover ALL return codes or at a very minimum check for SUCCESS or ERROR and call "perror" on the error path.
2. Lack of Verification as You Go - verify all code written at a unit test level (driver to call and verify functions), task level, and covering all possible return codes for API/library calls - likewise with hardware, unit test all components and fully characterize (e.g. test servos and controllers through full range, pre-test cameras and focus, etc.) Any transport protocols should include sequence numbers, timestamps, and headers so that dropped packets in UDP or lost TCP segments are easily detected.
3. Stack overflow & Heap issues - stack in Linux does dynamically grow via the "SBRK" system call you will see if you run strace on stack intensive applications - large array declarations within functions rather than as globals along with recursive functions consume lots of stack. Likewise, passing arrays/structs by copy, and large local variables/structs of any type may cause significant stack consumption and Linux does have an upper bound on how much a process/thread may use. It is a common mistake in Video processing and Image processing code to declare frame buffers on stack, which most often uses large frame arrays and can be a problem - likewise use of C++ "new" can be an even larger problem in terms of memory leaks and time required to dynamically allocate memory. Try moving array declarations out of functions, beware of "new", and consider one-time allocations in initialization if you do use malloc.
4. Re-entrancy - functions called by more than one task must be coded to be re-entrant so that each task executing the code has either its own stack, unique task indexed variables, or semaphore protected access to global data and resources.
5. Access out of Array Bounds - this will cause spamming of code, data, or stack segments and very erratic behavior - often a mistake made in large arrays with computed indices and with pointer arithmetic in C. Carefully verify all pointers and all indices during unit testing. This is a common mistake in Video and Image Processing code.
6. CPU Overload - often with video processing, image processing, encryption, compression, and other CPU intensive algorithms, CPUs can be overloaded and results in erratic behavior - this can easily be checked using KernelShark (similar to VxWorks Windview or Systemviewer). This is a common mistake in video and image processing code. If you do not want to learn ftrace or KernelShark (due to lack of time), the make sure you use syslog rather than printfs!!!
7. Not Reading the Manual - make sure you read all manual pages for API function calls carefully and understand if they can be called in task, ISR, or kernel context, all ERROR

and success return codes, and any limitations on use or pre-conditions that must be met prior to calling.

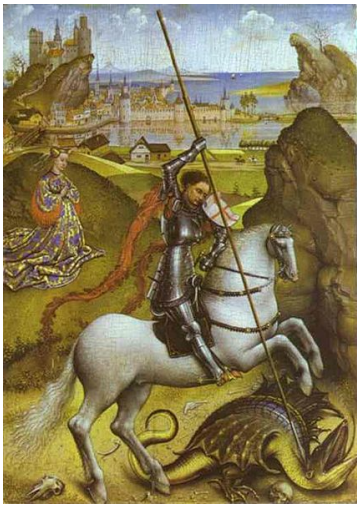
8. Use of printf - remember that printf can significantly affect the timing of code execution and cause blocking to task progress and is NOT allowed in kernel context (use printk in kernel context). It is best to use syslog instead of printf or printk, which simply queues messages output by a syslogd or klogd daemon in slack time.
9. Improper kernel task or kernel/user thread priorities - remember that kernel tasks should be assigned rate monotonic priority and elevated above priority of all non-real-time tasks - use POSIX threads and the FIFO scheduling class for user-space threads and be sure to assign priorities between RTMIN and RTMAX for all RT services. Pre-spawn processes or kernel tasks and POSIX threads so that process/task/thread context creation latency is not added for each real-time service release - pre-create in a driver and wait for service request on a semaphore.
10. Excessive or Improper use of sleep functions like "sleep", "usleep" or "nanosleep" - delays should not be used as a general mechanism to time services - instead, consider tasks, semaphores, and POSIX timers to sequence services. Ideally a sequencer approach should be taken with use of nanosleep in no more than one spot in your real-time micro-services design. Or better yet, if possible, do not use sleep at all and make your design interrupt driven with synchronization between ISRs and threads/tasks using binary semaphores or message queues.
11. Not using tracing and profiling tools - Use of ftrace, strace, kernel shark, systemtap, gcc gprof and gcov and/or sysprof will help you immensely - these are powerful real-time, timing, and event debug tools and should be part of standard verification steps for all threads (or kernel tasks) and integrated multi-service systems.
12. Not using Single-Step Debugger - use the graphical debugger, either with code that does not have optimization on, or in mixed mode ASM/C-source with optimization on, to verify code control flow, data values, and proper functionality.
13. Not making use of high-resolution timers and time-stamps - use of jiffies may be sufficient for coarse timing, but for highly accurate timing and timeouts, use the high resolution timestamp and timer features for Linux.
14. Semaphore Initial Conditions - make sure semaphores are initially FULL or EMPTY by design.
15. Failure to Destroy Resources on Shutdown - write a shutdown for all services and make sure to quiesce all hardware, delete kernel resources (e.g. semaphores, thread join/exit, message queues, etc.), and to undo all that was allocated in your service initialization.
16. Failure to Read ALL data from Sockets - calls to read must check actual bytes read and loop if return does not match expected to read all data from sockets and other types of network and byte-stream interfaces.
17. Endian Errors for Transported Data - use ntoh and hton when transporting data over networks, esp. between architectures that may have different endianness (e.g. Sun SPARC and Intel x86) and ALWAYS for multi-byte complex data types like "float", "double", or any type that is more than one byte at a time.
18. Data Corruption from Lack of Synchronization - semaphores must protect global data and resources accessed by more than one task unless data is read-only with no writers or ATOMIC update.

19. Data Corruption - Cache Coherency - drivers must flush cache before starting outbound DMAs and invalidate before reading when in-bound DMAs complete or use memory that is non-cacheable.
20. Failure to Take Ownership and to Understand Re-used Code or OTS Components - All re-used code must be fully understood by reviewing source and/or carefully reading associated documentation, manually pages, and testing - likewise all Off-the-Shelf hardware components must be fully understood based on documentation and in terms of unit characterization.
21. Failure to Use Verification Tools - tools such as Wireshark (TCP/UDP/IP trace), Kernel Shark (kernel event trace), debuggers, digital multi-meters to buzz out hardware and check voltages, scopes to view signal output (e.g. TTL PWM for servos), and logic analyzers should be used to verify assumptions.
22. Improper Handling of Hardware - Failure to insulate, use stand-offs, handle ASICs with proper ESD pre-cautions, failure to pre-check voltage levels, to power-off PCs before plugging PCI cards, and to carefully use lab equipment will lead to failures, lost time, and potentially not being able to demo projects if spares are not available.
23. Pre-testing Hardware - Any hardware that can be pre-checked for proper operation prior to integration with projects should be tested to greatest extent possible - e.g. cameras should be directly hooked up to LCD monitors / TVs, serial controllers tested with manufacturer provided code, servos tested one at a time, and cards tested with PCI probing. Check to make sure that hardware has Linux driver support.
24. Un-initialized Data - remember that while ANSI C should zero out global (BSS) data, it is best to statically initialize or explicitly initialize all global data prior to use.
25. Coding Bugs - to avoid simple coding mistakes in C, consider turning on "-Wall" for code builds and make sure all code is warning and error free. Use of "-Wall" will catch questionable code that can lead to run-time bugs - for example unclear implicit type conversions, unused or un-initialized variables, etc.
26. Suffering in Silence - not asking for help from other members of your group or outside your group (TA help, instructor debug session).
27. Get Another Pair of Eyes - if you cannot make progress on debugging, ask a team-mate to walk through your code with you and try to explain to them how it should work as you go.
28. Not Taking a Break - if you've been staring at a bug with no progress, print out code to take home and read later, go do something else for an hour and come back to it, or get some sleep and look at it in the morning.
29. Not Doing Background Reading - in addition to the API manuals, read the Programmer's Guides, use the web to search for errors and documentation and read comments in example code or open source code carefully. For Wind River manuals on google, use "wind river" and then the topic - e.g. "wind river network" or "wind river system viewer manual". Chip-set docs, manuals, and the course textbook all contain useful background information and resources. For example, an errata might hold the key to a documentation bug - e.g. wrong specification of a read-to-clear interrupt bit in a chip-set document cleared up in errata or in a Linux reference driver (this has happened to me). Read hardware specifications including devices like servos, NTSC cameras, etc. and leave nothing to doubt.

30. Over-focus on What you Think is the Bug - get second opinions on bugs, your code, and be willing to explain your code, take suggestions, and test theories carefully to eliminate and isolate (ruling out what is proven to verify) as you chase down a bug. Ultimately, consider an alternate formulation, algorithm, or system call for extreme cases where a bug persists.

TIPS for OPTIMIZING GRADE AND EXPERIENCE gained form RTES PROJECTS

TOP tips to get the most out of your work in RTES in terms of learning experience and your grade:



1. Be clear on grading criteria and how you spend your time.
   50% - Completion of all proposal, analysis, design and all sections of final report
   30% - Completion and demonstration of working system that meets minimum goals
   10% - Completion and demonstration of working system that meets target goals
   10% - Completion and demonstration of working system that meets optimal goals
2. Be sure you meet all write-up requirements and stay on schedule with analysis, design, code, and testing plan reviews and walk-throughs.
3. Spend your time wisely - if specific code or hardware is not working, verify and debug carefully documenting what you do (to include in appendix or test plans for partial credit) and then either get help and/or move on to other aspects of your project, then circle back later to see if you can make progress with fresh eyes on a trouble spot or with the help of team member(s). Do not suffer in silence or rat-hole in repeating tests you know do not work, keep going deeper on verification and debug methods rather than running the same tests over and over and looking at ONE suspected problem.
4. Go for maximum partial credit! - if a portion of your project does not work, stick with it, but if in the end it never works, demo what does work, and describe in your report and during your demo the challenges, how you approached them, what you did to verify/debug, any work-arounds you designed, and show me you understand why that piece does not work, that you could get it working with more time, money, parts, ... and that you worked on creative and technically thorough approaches to fix that subsystem, component, code module - i.e. show me that it would only be a matter of more time and money and it would be working ... or, admit that it was not the best design, tell me why, and tell me what you learned.
5. Document and clearly communicate - your personal role in your group as well as supporting your team-mates both in writing and verbally during DEBUG and at your DEMO.
6. Plan your demo and debug sessions, film any early tests (in case something breaks), keep good lab notes, show me your test code, your test cases, your debug and instrumentation

methods and be prepared to "tell me the story" so I can come to know your effort, perhaps help you with last minute improvements, and maximize your grade and final experience.

7. Avoid last minute major design changes - I'd prefer you demo the "pieces that work and don't work" rather than making a major design change and claiming it now all works even though it is way different from your proposal - stick to good design and if it wasn't the best design after all, be honest and tell me what you would do differently next time.
8. Manage your code, documents, hardware, and lab environment in a professional and thorough manner and tell me how you did this and document it! This also helps me help you.
9. Show me that you applied RM theory and concepts we learned in the labs and in class - e.g. priority assignment based on? Often this also results in tuning breakthroughs and even dramatic bug fixes.
10. Show me that you considered the top mistakes made by other students over the past 10 years and leveraged and learned from documentation provided in the book, class web, from the TA, using the Workbench tools (debugger, system viewer), etc. - e.g. http://ecee.colorado.edu/~ecen5623/toperrs_rtlinux.html

Tons of progress is often made in the last 2 weeks, so do not give up, keep up the fight, and stay focused on good methods and constant progress. Sometimes I see outright miracles happen in the last few days and even hours or based on debug during the demonstrations, walk-throughs, and reviews.