COMMON ERRORS MADE Writing System PROGRAMS

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 with a functional unit test or example input (test C/C++ driver to call and verify functions and/or test input). Design input and test cases so that they cover expected inputs and parameter ranges as well as boundary cases like zero, negative values, maximum supported values, and unexpected input.

3. **Stack overflow** - if recursive functions are defined (a function that calls itself), if this type of function does not have a termination condition, it may recurse indefinitely until stack space for calling arguments, local variables, and return addresses are exhausted.

4. **Re-entrancy** - This is only an issue if your function can be called by more than one thread, but if so, all global data that is shared and can be updated should be protected with a mutual exclusion semaphore. Alternatively, functions can avoid use of global data and only use stack or use globals that are however thread indexed and therefore unique to each execution context.

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** - Algorithms that process large amounts of data or include iteration or search with large ranges can take more CPU than you might expect, so if a program is not responding quickly, check CPU use with "top" in Linux or the Task Manager (Ctrl-Alt-Delte) in Windows if this is the case.

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. **Improper thread or process priorities** - remember that processes and threads should be assigned a priority based on fairness and response latency expectations - for common Linux and Windows applications the default priority is often acceptable, but for interactive systems, real-time, or any low-latency application, non-default priority may need to be set.
9. **Excessive or Improper use of sleep functions** - delays should not be used as a general mechanism to time programs - instead consider using thread or process scheduling features, semaphores, timeouts and timers to sequence applications.

10. **Relying too Much on Console Debug compared to Debugging Tools** - Single step debugging, adding variable watches and setting breakpoints in a visual debugger is normally the best way to find bugs in code compared to adding "cout" or "printf" calls in code. If at all possible, try to use a debugger included in your development environment. In some cases this may not be possible (e.g. Linux kernel programming), but other options like use of log files may still be better than "printk" and standard output.

11. **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.

12. **Semaphore Initial Conditions** - make sure semaphores are initially FULL or EMPTY by design and carefully select not only these initial conditions, but also any options (e.g. binary, mutual exclusion, or counting semaphore semantics).

13. **Failure to Release Memory and Resources** - In general your code should release all resources allocated during initialization or runtime - e.g. if you call malloc, then you should call free, or if you call open for a device or file, you should call close.

14. **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.

15. **Endian Errors for Transported Data** - use ntoh and hton when transporting data over networks, esp. between architectures that may have different endianess (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.

16. **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.

17. **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.

18. **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, manuay pages, and testing - likewise all Off-the-Shelf hardware components must be fully understood based on documentation and in terms of unit characterization.

19. **Failure to Use Verification Tools** - tools such as the single step debugger, Wireshark (TCP/UDP/IP trace), Linux top or Windows task manager, or for me embedded solutions, digital multi-meters, oscilloscopes, or logic analyzers should be used to verify hardware assumptions.

20. **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.

21. **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.

22. **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.

23. **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. In general, do not suppress or turn off warnings, turn them all on and pay careful attention to any compiler or linker warnings during build.

24. **Suffering in Silence** - not asking for help from the instructor, your classmates or TA.

25. **Get Another Pair of Eyes** - if you can't 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.

26. **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.

27. **Not Doing Background Reading** - in addition to textbooks and manuals, read the Programmer's Guide materials for your development environment and use the web to search for errors you encounter, documentation and read comments in example code or open source code carefully. With some research on the web you can learn not only the hard way from your own mistakes, but from commonly made mistakes by other programmers.

28. **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 YOUR DEVELOPMENT EXPERIENCE

TOP tips to get better grades on programming assignments and to avoid frustrating bugs.

1. Be clear on grading criteria and how you spend your time. Most exercises have point values associated with each requirement in the assignment (grading rubric), so spend your time in proportion to the value.

2. Clearly indicate any assumptions you make and restate your understanding of problems in your own words.

3. Spend your time wisely - if specific code or hardware is not working, verify and debug carefully documenting what you have done (this allows for partial credit and better grader feedback) and then either get help and/or move on to other aspects of your project. Circle back later to see if you can make progress with fresh eyes on a trouble spot or with the help of your instructor or a classmate. Do not suffer in silence or rat-hole in repeating tests you know won't work - avoid running the same tests over and over and looking at ONE suspected problem.

4. Go for maximum partial credit! - if a portion of your program does not work, stick with it, but if in the end it never works, demonstrate what does work, and describe in your report the challenges you encountered, how you approached them, what you did to verify/debug, any work-arounds you designed - show your grader that it would only be a matter of more time to fully complete your program if you run short of time ... or, admit
that your design has short-comings and tell your grader how you might improve and what you learned.

5. Comment your code well and avoid use of "magic numbers" or hard-coded values that have no explanation.

6. Document and clearly communicate - always provide an easy to follow report on your program design and how you tested it, with example input and output so that it's clear your program works.

7. Avoid last minute design changes - programs are often broken by "one last change to clean up" code that is not tested. After any code change you should re-test your code (regression test) to make sure there are not unintended consequences of your simple change.

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 your instructor that you considered the top mistakes most often made by reviewing help like this.