The best way to debug something is to make it crash immediately and deterministically.
— JOHN CARMACK

Beware of bugs in the above code; I have only proved it correct, not tried it.
— DONALD E. KNUTH

If debugging is the process of removing software bugs, then programming must be the process of putting them in.
— EDSGER W. DIJKSTRA

A colleague submitted a questionnaire concerning a proposed class on debugging. This paper is a response answering his direct questions In place, I've included different perspectives on the course to bring the students closer to understanding other concepts important for practicing the art of debugging embedded systems.

While I'm not a stranger to melting the lead, and home-brewing the widget, this author is probably better described as a real-time embedded software engineer for spacecraft. (With a dose of System Engineering DNA and general *Architecture follows Requirements* mind-set).

> *An obvious truism is that any program must do what its specifications say that it is supposed to do. This is no less true in the case of concurrent programs though the specifications are radically different from those of sequential programs.(Ben-Ari, 1982)*

I quote from *Principles of Concurrent Programming* to illustrate that concurrent programming has a lot in common with the nature of embedded systems and embedded software. Contention and resolution for resources must occur without failure, side-effect, or halting. Debugging for embedded systems is then truly awesome work to do.

Debugging is

- Hacking, in the purest form.
- Fixing the broken parts.
- Learning how the good parts work.
- Required every-time new development is implemented.
- Vital for testing.
- Is no-holds-barred development – you use whatever you got. Scopes, instrumented source, logging, debugger probes, source code. There is nothing off limits.
- What makes good hardware work well.
- Finds out what makes bad hardware broken.
- Proves the project is ready to ship.
- Separates the wheat from the chaff. I'll hire a good debugger over a prolific software developer. Up until now (hopefully) people go to school to write software. But they don't go to school to learn how to debug systems, hence the bias.

Terms like "you" and "your", etc… are used when this author is referring to the hypothetical **student.**

First some defined terms to avoid confusion.

- **firmware** We're going to split hairs on this. For those environments where FPGA, etc…are used we're going to use the term firmware to describe implementations in VHDL, etc…. For those environments where the most complex electronic system is a microprocessor (micro-controller unit (MCU)) and not FPGA, then firmware will mean compiled implementation in C (or C++) that does not rely on an embedded real time operating system (RTOS). To put it simply then:

- firmware is fabric, implemented in VHDL, etc...OR
- firmware is embedded software akin to BSP, facing inward towards the hardware

Unless otherwise noted we will use the latter definition. Firmware is low level C or C++ software written to facilitate control of underlying hardware interface4, providing at most (and no more than) an abstraction between the application software (if any) and the underlying hardware. Some like to refer to this as simply device drivers, and that is acceptable. But then again, we are careful to realize that in the spirit of the question firmware is really a structure built with three intents:

1. Provide low-level hardware abstraction. Example: The C code that initializes a GIO, provides set/get methods on pin-muxed GIO. Example: Establishment of a heart-beat interrupt or heart-beat GIO.
2. Provide interface to hardware. Example: The C header which defines the contract between the low-level driver and the high-side facing behavioral logic. Example: Macro or functions that generalize identification of I/O subsystems. Interface is leveraged by low-level firmware and behavioral capabilities.
3. Provide behavioral capabilities. Example: Routines much less (if at all) aware of the specific hardware and focus on the heuristics of functional behavior. Example: A communications routine which assembles MIL-1553 header and payload data marshalled across a UART. Example: Configuring ISR that reacts to heart-beat.

If the idea of firmware falls into the three issues above, then we are on the same page.

If the idea of firmware falls instead on the fabric of FPGA, then we are talking about something else. Fabric (firmware) on FPGA is out of scope for this paper, however we will describe some debugging examples where it is a desirable to have awareness.

If nothing else, the deciding factor to succeed debugging embedded systems relies on a few traits and resources based on nothing less than plumbing the depth of the software execution to the hardware. [1,2]

- First – did you use requirements to architect and design your solution? Just as *No man is an island.*[3] neither is a requirement standalone – it was derived from somewhere. If we look high enough up the chain we find the golden BWI (Boss Wants It) level of requirement. Looking down along the left side of the requirements "V" we find things like SSS, SRS and so on. Are these terms still mysterious to the students? Do they have any awareness at all of what Systems Engineering is and how it affects architecture and design? A crash-course (one week) on the difference between "REQ-1: The Widget-2000 shall make coffee." vs. "REQ-1001: The Widget-2000 Thermal Subsystem shall provide the capability to close the Heater Switch upon detection of Thermal Sensor having values subject to the Widget-2000 Thermal-Switch-Sensor Ratio ICD subject to Policy within TBR 100ms."
  Was the word "shall" uttered a few thousand times in the process of settling on **what** the system is going to do before anyone attempted to describe **how** the system works? After your *System Engineering* efforts codified your Requirements did you then contemplate the few design solution preferences, finally settling on one preference of design solution? Were trades made?
- Complete and comprehensive knowledge of the programming language involved. For instance, if the user is developing in C, then there is a set of *gotta-have* skills and awareness. There may not be space here to run a discourse on what those skills are. Let us be frank. Awareness of `const`, `static`, `volatile` come to mind first. Aside from that: bit-twiddling, ISR's, function pointers, vector tables, stack vs. heap are critical.
- Complete awareness of the machine. The 2000-page technical reference manual that is provided by the silicon vendor and the errata sent after. This author jests with new hire engineers: "tell me exactly what happens when I put the batteries in." – meaning as soon as Vdd is applied, what happens in the MCU. At what point does the MCU begin executing instructions?
- What is the plan to get meaningful telemetry from the system – what can you learn about current execution state, and does that information help you understand what the system is doing?
- Before the hardware team announces the delivery of their new fabricated boards (Fab-A, EV0), can you as an embedded system developer claim to already know with high confidence your embedded software design will work/run close to specification? Hint: did

[1] My mentor Bob Hayes made this point a few times early in our engagement during Hardware Systems courses. We used Motorola 68010 processor mother boards, serial port, not much else.
[2] The **BUFFALO** (**B**it **U**ser **F**ast **F**riendly **A**id to **L**ogical **O**perations) monitor debugger was a built-in firmware for Motorola's M68HC11 micro-controllers, residing in the chip's ROM, used with evaluation boards
[3] John Donne, Devotions Upon Emergent Occasions (1624)

you use evaluation boards prior to receiving the Fab-A spins from HW Dev? If so, you get a gold star.

Answers to the Questionnaire are below with a copy of the question and my thoughts that come to mind. Let's get into it.

1. **Q:** *Do you think this class would be worthwhile, or basically a waste of time. Please elaborate.*
   Yes, the class would be worthwhile if the course covered the aspects of debugging that are expected from an early career engineer. In industry we are disappointed if the new engineer cannot have JTAG or SWD or similar debugger interface cooperating with their debugger utilities and show us stepping through instructions what occurs on the system just prior to the defective behavior.
2. **Q:** *Assuming you feel there is merit to the class, should this author would focus more on software hardware, SW/HW integration, use of tools, processes, etc. In other words, where would you put the emphasis?*
   This author has a bias. The focus should be on the SW/HW integration. The most valuable HW person on the team is the HW person who also can wield the compiler, write embedded software, load it, debug it, and prove beyond a doubt (based on the Requirements of course) that the sub-system, or system of systems is working as expected (happy path test case). Rinse-repeat for the Fault test case, Hazard test case, etc.... (We're going to get back to Test later because there are some important issues to sort out between design for test and testability.)
   As far as focus, this author is dismayed at the use of visual integrated development environments (IDEs) in classes for development. IDE's assume a great deal about the hardware and system that was designed to solve the requirement. Often, the IDE is not permitted because there is no support for any IDE (either technically not permitted, or the cost-basis for procuring the development system is prohibitive)[4]
   As much as we try to avoid the truth - the open source initiatives have produced some of the better flavors of debugging systems out there. gdb, openocd, and so on. If this author were back in the industry again, a candidate isn't hired unless they show a complete use-case of debugging an embedded system that covers gdb in the shell without an IDE.
   The class should teach the fundamentals before assuming use-cases that involve popular IDE software.
3. **Q:** *I was considering using the approach of the Harvard Business School. This would be case studies as the basis for examining defects, their causes and how to fix them. Do you think this approach has merit, or should I take a different approach?*

[4] Green Hills Software debuggers, for example, usually cost between $10,000 and $20,000 per seat. Compared to SEGGER J-Link at $600 per seat.

The trouble with case-studies is that they are usually obsolete by the time your book is past the acquisition editor phase and you're getting your first dozen *gratis* copies.

Then again, what this author would prefer to see is driving factors – early detection of faults, undefined-behavior, and defects – the way the IEEE[5]System Engineering papers (Software Requirements Engineering (Thayer and Dorfman, 1997) is a good example. Biasing towards IEEE papers(Shankar, 2025) for worst-of-the-worst situations has two benefits: It re-introduces the students back to the IEEE as a bottomless well of new and innovative tactics for solving technical problems, and second it reminds them through their research that all of these problems are not new. They've been solved a few times. Maybe there's room for a new solution (that's where they come in).

A good week 1 homework would be: Go to the library, and find three white papers from the IEEE that discuss debugging tactics in embedded systems, pick out from those three papers the most interesting example cited (in the paper or in the references). Then, write 1000 words why the example is interesting, what questions you have concerning the approach/issue.

4. **Q:** *Do you have a favorite bug story that I could use? If so, please describe the bug, what caused it, and how did you find it, and fix it. Is there a take-away lesson that you can share?*

   Some of the more interesting stories involve the development of the Xbox One console.

   Microsoft certainly was maligned in the press over the RRoD *(Red Ring of Death).* For the non-gamer, this was the problem the earlier versions of Xbox had whereby the console locked up, and the error display among other indications was the red LED circle illuminating to indicate serious failure.

   The problem was eventually root-caused by a manufacturing process defect of the GPU/CPU components, solder paste wasn't flowed when the BGA parts were laid in.

   Anyway, as a result of RRoD, the Mountain View office of Microsoft (which did most of the silicon design) constructed a new laboratory. In this laboratory test-vehicles (boards designed with specific variants of the silicon and micro-controllers attached) were deployed by the hundreds in an array. On each board was a GPU/CPU package that was identical to the one used in the console. However the circuitry in the GPU was all together different – it wasn't a processor, but a matrix of heater elements and sensors in a x-y array, each controlled by custom firmware on the MCU attached. With this setup, the team could create a thermal floor-plan that matched varying grades of usage. We could "heat up" regions

[5] IEEE articles detail several specialized techniques for resource-constrained environments:
- Non-Invasive Debugging: Techniques that avoid the "probe effect," where the act of debugging alters the timing and behavior of the system, are critical for real-time and concurrent systems.
- Strategic Serial Output: Using serial output for real-time state monitoring is a highly accessible method that can significantly reduce diagnostic time, especially for security-related flaws.
- Instrumentation and Profiling: Adding small amounts of code (instrumentation) can provide deep visibility into system performance and timing with minimal impact on execution.

of the GPU as if it were a real GPU running real games. And as a result, we could measure thermal properties in the same package. That was stage one.
Stage two of the experiment was design of the mitigation factors – how to cool the die. We experimented with different heat-sink configurations, thermal paste compounds, and so on until we could arrive at a proven solution to keep the system healthy, reduce the cost to minimum, and have reliable performance to reach the CTx – Critical To Customer, Critical To Function, etc….
The process of designing the wall of test-vehicles, running 24/7, specialized firmware, special hardware test-vehicles and analysis of the results was not trivial. The goal was to sell 30-40 million units of Xbox. So, this level of effort was not unreasonable.
This goes to a broader point that is sometimes lost on early career embedded system developers: **Big prizes await you if your software can help find bugs in hardware.**
Getting to the point where software is able to detect hardware failures within hours after the first articles of hardware are fabricated is the main reason why HW development teams are coupled closely with Software teams. We're going to come back to this idea in the second part of the paper regarding aspects of the class proposal to be considered.

5. **Q:** *Any course in software engineering will go into great detail about how to avoid bugs in the first place, i.e., teaching the best practices in software design. Hardware design is less rigidly structured, but I can certainly come up with best practices. Should I also spend course time on best practices for designing hardware and software, or just focus on the debugging process?*
   Short answer:
   - Yes.
   - Best practices that are compile-time practices, not run-time.
   - Hammer the point of `const`(Saks, 1998)

   Long answer:
   This may be true for some situations, but not true often with respect to embedded system development. The phrase "how to avoid bugs in the first place" is a loaded statement that assumes there's a finite list of best practices and where following them alleviates the risk of bugs. Perhaps in some situations, especially when the software is high-level (application software). This author hasn't found that to be the case in the triumvirate of embedded software (low level, interface and application). Here are the exceptions:
   - O.K. It's time for a bit of tough love. Understanding `const`. Engineers overlook just how important it is to use `const` in their software design. What most students are taught is wrong. How many instructors are telling their students that `const` means

constant?.[6]

`const` means read-only. Processor datasheets are explicit about registers that are read-only, or write-only, or read-write. And, there are instances where mistakes following the specification can lead to run-time failures. A garage-door opener circuit doesn't have a Design Assurance Level, DAL rating to be too concerned about. But a traffic system, or medical device, or transportation system, aerospace application, satellite? These have more severe DAL objectives. Depending solely on the test team to traverse all ranges of motion is not enough.

Read-only is not the same as constant.[7]

It shows that the data pointed to by `pREG` is `const` and therefore read-only. It is not constant. We rely on the underlying hardware register to change based on circumstances. The `const`-ness of the value means that at Compile-time.

It won't compile, therefore the code won't be loaded on the system and run. Therefore the bug of trying to set a value on a register that the data-sheet marks as READ ONLY cannot occur. Some hardware react badly when trying to write READ ONLY registers. (See earlier point about being aware of the technical manual/data-sheet of the silicon involved).

The "best practices" that was referred to in the question didn't differentiate between practices that are used for compiled software versus practices that are used to stop the wrong software from being compiled.

`const`-ness is far more effective in dealing with the problem of writing (updating, changing) data that should not be updated at run-time.

Compile-time policy (versus run-time checking) is far superior lesson to learn.

- Similarly, awareness of `static`, and `volatile` for different reasons is key to preventing bad software from passing the compiler.
- Understanding what the purpose is for Interrupt Service Routines – why they are written as they are. What you're allowed to do inside of them, and what is strictly forbidden.
- Understanding the Machine instructions available when certain kinds of operators are used in the software.

  Suppose the operation was some simple arithmetic. But, these arguments are unsigned long. (Usually 64 bit wide values). Your processor could be a 32-bit processor. What happens? The compiler does the best thing it can do and emulates the instructions to do the arithmetic using 32-bit values, shifting and moving the data around so that the result is accurate. However the key there is the compiler decides to leverage built-in stanzas of as-

[6] If so, they need their credentials revoked. When candidates came to me for embedded systems jobs and they told me that `const` means *constant* the interview is over. There is a trajectory for pursing embedded system development and regardless of the EE vs. CS bias they choose, one thing is certain – fluency in the core languages used to write embedded software is an absolute. There two: C and C++. Every other language, despite the protest, will never replace them.

[7] This shows **Policy** and Policy in software (frankly anywhere) is the lint-trap catching bugs before the compilation finishes.

```
// Correct
#define MASK (1U << OFFSET)

const uint32_t* pREG = ADDR;
// ...
while (*pREG & SOME_MASK) {
    do_something_quickly();
}

// Even more correct
volatile const
   uint32_t* pREG = ADDR;
// ...
while (*pREG & SOME_MASK) {
    do_something_quickly();
}

// Incorrect
uint32_t* pREG = ADDR;
// ...
while (*pREG & SOME_MASK) {
    do_something_quickly();
}
```

```
// This will not compile
// Stops the bug before
// it is deployed.
const uint32_t* pREG = ADDR;
// ...
*pREG = VALUE;
```

```
// Case in point:
unsigned long x = VALUE;
unsigned long y = OTHER;
unsigned long z
 = chefs_choice(x, y);
```

sembly – instructions you did not write but the compiler inserts as assembly.

Why is this a problem? Because the assembly it inserted may not be thread-safe. We see this a lot in use of floating point arithmetic also – stanzas of routines that ship with the compiler, used (blindly) by the developer and as a result the software is not thread-safe.

A developer could stare at the source code all week long and never realize the impact of the compiler revising the software without the knowledge of the user.

A homework-assignment would be to catalog a few instances of compiler toolchains offering stanzas of assembly in lieu of the desired instruction (that does not exist on the MCU). A second part of the homework is, using the language features how would you design your embedded software to detect at compile time the platform on which the software is targeting cannot reliably (safely) execute the instruction? (Hint: It involves use of `#error`).

The point is this – best practices in software engineering for embedded systems are most useful it it stops bad software from being compiled in the first place. Once the software is compiled, detecting flaws/defects is more difficult in embedded systems than fancy high level applications. More on that later.

6. **Q:** *I want to create debugging labs for the students. I have some ideas, but I would love to hear your thoughts on this. My major concern is how to create hardware faults that won't start fires.*

   A few examples that would drive this message clearly:

   - (INSTRUCTOR) On a target EVB, deploy some firmware (compiled C software flashed onto the target, begins execution upon POR (boot), Power on Reset). No RTOS (No operating system at all, actually). Vectors from POR into `main())` eventually and all that happens is the proverbial Blinky application.
   - Students are given:
     (a) The linker script (`.ld`)
     (b) The `.elf` with symbol-table
     (c) The EVB
     (d) The JTAG (or SWD) debugger (probe).
     (e) The instructions to setup gdb and openocd.
     (f) The Datasheet of the MCU (2000-page technical reference PDF)
     (g) A set of Test Cases (instructor written purposefully to be improper and incomplete – a set up for Phase 3).
   - (INSTRUCTOR) Beforehand, however, what you've done is instrument the software specifically: If one of several GPIO are

grounded before POR, then the Blinky application will perform the demonstration with different diagnostic patterns (fast, slow, wave, whatever). Test the GIO, branch to different Blinky behavior. But, you only **claimed this is the case – your instrumented GIO/ Blinky is intentionally broken. NONE of the different behaviors are working. No matter what GIO-chord is applied, the blinking is the same pattern.**.

- (STUDENT/INSTRUCTOR) Show them the requirements – it says (in some style of requirements you prefer) what GPIO-Chords are to cause certain Blinking behavior. Make the requirements distinct:
  - REQ-BLINKY-01: IF powered, UPON RESET cleared, WITHIN 500ms, the Blinky-2000 SHALL illuminate GPIO LED-1 (Subject to ICD) for continuous pattern: OFF 1000ms, ON 1000ms.
  - REQ-BLINKY-02: IF powered, UPON RESET cleared, WITHIN 500ms, the Blinky-2000 SHALL illuminate GPIO LED-2 (Subject to ICD) for continuous pattern: OFF 2500ms, ON 2000ms.
  - etc...

  The ICD is just a table of which GIO are active low, to dictate which GPIO LED-n are in blink mode.
- (STUDENT) The job of the student is this. Phase 1 (blind debugging)
  - Write up the report – Which "functions" (i.e., routines with names) are executed BEFORE `main`.
  - At what address is the behavioral software loaded into non-volatile memory?
- Phase 2. (STUDENT) Now give them the source code, and the makefiles.
  - Debugging – what is the instruction that sets the limits for how long a target LED is on or off?
  - Debugging – what changes to the source code will affect which blink pattern will occur when detecting GIO are active low?
  - Debugging – what function would you add (instrumentation) to the embedded software to detect the BSP (Board Support Package) is capable to read if a GIO is asserted?
- Phase 3. (STUDENT) Debugging – Your "project manager" doesn't have o-scope awareness, so what can you do for them with software that demonstrates the Blinky-2000. (We want them edged closer to understanding that telemetry from the system is crucial in design verification. Does the board "beep and squeak?" Does the board log to UART, or even Ethernet (UDP) telemetry about activity instrumented in the software?

- (STUDENT) Debugging – Compare and contrast the Test cases that were used and what you did during debugging and investigation. Compare the requirements to the test cases.
  - Is this a case of the Test Case being incorrect (The test is not explaining a process to test Blinky-2000 based on requirements, but rather just "good ideas").
  - Is the test process you used different than the test process explained in the test case? Why or why not?
  - Explain the benefit and features of telemetry gained during development which inform the user/developer about behavior of the system?
  - Categorize the problem: Is this a case of the test case being incorrect/improper or is this a case of the requirement not being correctly implemented? Or both? How does this challenge affect debugging?

7. **Q:** *When I wrote my debugging book in 2018-2020, Applications Notes were a great source of material. Do you have any favorites that you could share with me? I will always cite my references.*
   Within embedded software environments/problems, the use of Application Notes isn't as prolific as in the HW realm. This author has seen a few interesting cases in Application Notes (and even found bugs in the Application Notes) – much to the dismay of our development team.
   A case example was when New Glenn designed the Power Distribution boards with Microsemi FPGA (IGLOO2) and Texas Instruments TMS-570 MCU processors.
   The challenge was to provide the capability to flash new bitstream of FPGA fabric onto the FPGA from external sources. The FPGA had a Cortex-M3 core (MSS) connected by SPI bus with SPI-Flash memory accessible by the FPGA. The solution that was preferred was a TFTP server running on the Cortex-M3 (the FPGA had its own MAC/PHY facilitating TFTP).
   Some of the analysis that was found through debugging as follows:
   The original note to the team:

   > Through the course of the development, the author found the boiler-plate startup-code (assembly) had a small defect.
   >
   > Here is the fix applied. It might be worthwhile to take a last look at this with the team to be absolutely sure of the fix. So far it has been fine, but I wouldn't want to ship the software without some discussion about it.

File: `startup_m2sxxx.S`

```
/*----------------------------------------
 * block copy.
 *
 * r0: source address
 * r1: target address
 * r2: end target address
 *
 * note: Most efficient if memory aligned.
 * Linker ALIGN(16) command should be used
 * as per example linker scripts.
 * Note 1: If the memory address in r0 or r1,
 *   byte copy routine is used
 * Note 2: If r1 < r2, will loop indefinitely
 * to highlight linker issue.
 ----------------------------------------*/
block_copy:
 push {r3, r4, r5, r6, r7, r8, lr}
 cmp r0, r1
 /* Exit early if source and destination */
 /* are the same. */
 beq block_copy_exit
 cmp r2, r1      /* JDW - BUGBUG -- */
           /*  If compare r1 to r2.
           /*   This difference is supposed */
           /*   to be used in R2 for counting the */
           /*   number of bytes to copy. */
           /*  When the value is zero */
           /*   (when R2 == R1 is true), then */
           /*   R2 is zero by result of the SUBS.W */
           /*   instruction (original) below. */
           /*  When that happens the copy routine */
           /*   incorrectly copies far more bytes than */
           /*   are intended and runs off the end of */
           /*   the viable buffer. */
 beq block_copy_exit  /* JDW - BUGBUG -- */
           /* Thus, if the target */
           /*   address equals end target address, */
           /*   we exit early! */
 /* Calculate number of bytes to move */
 subs.w r2, r2, r1
 bpl  block_copy_address_ok   /* check: */
          /*  (end target address) > */
          /*   (target address) => continue */
 b .      /* halt as critical error- */
          /*    memory map not OK- make it easy  */
          /* to catch in debugger */
```

The lesson here is that despite what vendors say about their SDK, it is always possible that the SDK (which includes software template/examples) is wrong. Why else do they provide version control of their SDK? In this instance the author had to make several phone calls with the Microsemi technical contact to prove to them their SDK was buggy (at fault) and gave them the corrected version of the driver for the SPI related software (Direct-C API).
While it would be easier to just wave hands and say that the issue

is third party software (Microsemi) and then wait for fixes – or dig into the software itself at the assembly and root out the problem.
Which kind of embedded software engineer do you want to educate?
What kind of embedded software engineer do we want to hire?

8. **Q:** *Regarding debugging as a disciple, can you articulate any process that you followed, or would recommend?*
   Before there is *debugging*, we need to know about the *bug*.
   When do we find bugs?
   - We find bugs as we develop the software, before the software is checked in.
     - Edit
     - Compile
     - Flash (deploy on test hardware)
     - Unit-test*
     - Ascertain if the result is matching the expected outcome per requirement.

     After that, we might feel confident to check in the change to the development branch that is associated with the feature requested (per requirement).
   - On the other hand, suppose the reason why the developer is editing the software at all is because a bug (issue) was found during formal testing. Formal testing definitions vary – let's say for sake of argument that Formal Testing is defined as testing efforts which result in credit earned for proving the correct behavior of the system subject to requirement. The key-word is "credit". Dev testing (by the developer, in private, before even checking in the source code) is NOT formal testing.
     Still, we have a lot of cases where bugs (issues) are found during Formal Testing efforts. These are the classic types of bugs. A de-skilled staff member is following a recipe of steps to evaluate the correct behavior of the system. They follow a script (hopefully written and traced with Requirements), and find actual results differ from expected results.
     User/Developers do not like heavy process burden and so typically the preferred solution to handle this is the same process used to implement new features (see above). The only difference is to which branch the new (changed) work is made. The popular term used these days is *git-flow* – which describes the steps used in development to move a batch of changes from a developer's work branch into a branch that builds for Formal Testing and so on. The word *flow* refers to the path the change takes in the Configuration Management System (CMS).

- The branch with the bug fix is merged into a collecting branch for execution on Hardware-In-The-Loop (simulated Sources and Loads) *(HITL)*.

There was an asterisk (*) above to call attention.

The question you are asking – about preferred discipline when debugging is a much larger question. Knowing full well that the software will have bugs – bugs (issues) that are found in Formal Testing, bugs that are found during informal-testing, bugs that are found during very serious flight-qualification testing, and so on. (As the level of rigor of the test develops, so does the level of rigor of the process used to apply corrections to the system.)

What is the discipline? This author never approaches an embedded software problem until two things are figured out in advance:

(a) What is the simplest and fastest way for me to get telemetry from the device under test (DUT). In a word: Communication. The board with my software must "beep and squeak" from day 1. How do I make communication the first feature of the embedded system. Not every software engineer has a scope or hardware development accessories – but what they usually have is a JTAG (or SWD) debugger probe, some software they wrote, and other inexpensive debug accessories (FTDI USB-Serial expansion, etc…).

(b) Whatever the name of the system is, prefix it with the word **tiny** and develop the Blinky/Hello-world version of the software for the target as long as the HW team has provided:
   i. A JTAG/SWD controller pad/header
   ii. A couple GPIO's
   iii. (and if I'm fortunate) either a MAC/PHY for Ethernet (UDP) spew, or
   iv. A UART that is not already devoted to inter-board communication or other purpose for the stated mission.

So the discipline ?

- Write a Linux/Windows based tool that comprehends simple C2 (Command and Control) with the DUT.
- Write message handling communication in the embedded system capable of emitting telemetry and accepting monitor-like commands (via C2) to peek and poke registers and data held by the embedded target.[8] In the case where some silicon was not directly connected to embedded software, a mailbox was used to pass messages between firmware and remote silicon on the board. Nonetheless, the user application was essential to instrument tests (developer, or otherwise) to investigate behavior as the embedded system was being developed. Without `dsmcdbg`, there would be no Xbox.

[8] Xbox was developed in such a way. A very versatile tool called `dsmcdbg` was continually developed to allow access to any register and memory location on the target device MCU(s) subject to an ICD.

- As early as possible, align the telemetry from the system to the official data-book telemetry ICD. Even if the DUT and the embedded software began is incapable of operation to requirement, at least it is emitting (bad, but correctly formatted ICD telemetry) so that *other parties in the team* that are preparing *their test cases* are able to use the *simulated* telemetry with some confidence.
  In slang, what we want to do is sort out as quickly as possible:
  The "gazintas" and "gazoutas."
  It's old-school engineering slang:
  gazinta = what goes into a system ("goes-into" – gazinta)
  gazouta = what comes out of a system ("goes-out-of" –gazouta)
  Resolving as quickly as possible how do this even before any significant architecture related to the intended behavior is ideal.
  This is not a mere hack. We try to make *Telemetry* as a service as abstract as we can up front, but doing so before much anything else is a discipline.
- Not so much a technical aspect of discipline, another important task to sort out in advance is the process the team will use to define bugs. It's difficult to attempt debugging when the bugs are malformed. An agreement has to be made with the teams who write bugs and teams that resolve bugs.
  At the bare minimum, a bug is as follows:

(a) A brief synopsis of the problem. Two or three sentences. Don't write a novel.

(b) A list of the requirements that are affected by the bug. This is absolutely fundamental. Just list the requirement ID numbers (or links) so the parties can compare the requirements affected with the bug report.

(c) A ordered list of stimuli used to demonstrate the bug. Exactly, in order, what preceded the bug demonstration? List all of the environmental factors such as the network topology, run-time environment, configuration files. Then very methodically list the steps (stimuli) that were used to provoke the system to reach a defective demonstration. Commands entered? Messages sent? External signals perturbed? Buttons pressed? Etc...

(d) A very brief and succinct list of EXPECTED RESULTS. What did the user expect to happen. Align the Expected Results with the STIMULI. There should be a 1:1 correspondence between Expected Result and Stimuli.

(e) A very brief and succinct list of ACTUAL RESULTS. Here's where we come to the identification of bug-like behavior. At each Stimulus, what was the actual result witnessed (logged, captured, etc...).

(f) OPTIONAL: If there is reason to offer it because of intrinsic knowledge of the test operator, provide a two or three sentence (brief) summary of what a proposed solution might be.

Other organizations are prime to prefix a bug with other metadata subject to their design policies: Like priority, reproducibility, severity, etc...
When bugs are triaged those factors may be used to re-order work to do for resolving the bugs.
If this is in place, then *Debugging* can occur.
We gotta know what the problem is before it can be fixed.

9. **Q:** *I plan on spending a lot of effort on design reviews, both formal and informal. Any comments or concerns?*
   For this class?
   I'll go back to some discipline that works:
   - *Know what you're going to do before you do it.*
   - *How do you measure success?*
   - *How do you know you're done?*

   For a review of design, ask the question – *What is the Approval Basis for the design?* This question will get a few side-long glances.
   What is the Approval Basis – what were the rules and processes that were set to determine if all of the Requirements levied onto the system are described in the Architecture? What were the trade-offs between competing design solutions? Why was the design solution ultimately preferred?
   Is the design biased on a functional block view of sub-systems or does the design identify *interface boundaries* between sub-systems?
   Does the Test effort anticipated align with the design approach?
   Let's be clear – **design for test** is fine to a point, but the more effective approach is **design for flight.**. A Requirement is the first place where testability is measured. Is it feasible to test this requirement? The imagination of the Test team is leveraged and the imagination of the Software/Firmware/Hardware teams should be conspicuous – In the architecture do we cover the system of systems and how they relate?
   If it was me, there would be some brainwashing performed:

   (a) Requirements specify what something does. Not how.
   Tell-tale words in requirements that will confuse developers and test teams – The words like *with, by, using* – those directly pierce the boundary between a What versus a How statement.

   (b) Architecture specifies HOW the system works. Stacks of papers would fill my workspace that discuss Architecture. IEEE has covered this exhaustively. The essence that I'd try to distill for the students is that we prefer Architecture bias towards explaining *How the requirements work together as a system.* Further, the trait of an Architecture describes this critical *how-statement* as defining *Interface Boundaries*, and/or defining *Components*. Why is so much ink being spilled over this? It goes back to Requirements versus Architecture – and identification – which drives Debugging for pursuit of correct behavior *specified by Requirements.*

(c) Test Descriptions specify needed procedures to adjudicate correct behavior subject to Requirements.[9]

If even an abbreviated form of this Process is explained and used by the teams under review it's still worthwhile to demonstrate the awareness to these principles.

The modern software/firmware development activities may be enjoying a rush of Agile/Scrum development practices, yet that doesn't bypass the simple truths:

- *Know what you're going to do before you do it.*
- *How do you measure success?*
- *How do you know you're done?*

10. **Q:** *Last question. Anything else I haven't covered above that you would like to add? Also, would you be willing to continue to advise me as I develop the course. I would be sending you lecture material and lab experiments to review and critique.*

This kind of class opens a can of worms.

In some ways the class is a case-study in and of itself about all kinds of aspects of design and architecture which lead to debugging issues.

But if the aim of the class is to focus solely on Debugging, perhaps given the span of ten-week course there may not be time to cover the preliminaries described by this paper.

Perhaps the ideal course is simply this:

- Get oriented: JTAG, SWD, debugger software. Get feet wet with simply using gdb on a native Linux application (breakpoints, conditions, symbols, map files, single-stepping, etc…)
- Every week, a new embedded software project. Examples from 37+ years ago in a course aptly named `Hardware Systems` are good to consider. Or, actually, some of your past classes had some interesting problems to revive for this class (LCD, Debouncing, etc…). I'd add some ANSI terminal driver for simple games. If you remove the visual IDE from the mix then you get down to the magic-sand debugging faster.
- Every other exercise give them a drop of a software package that is intentionally buggy.
- On each team a role is filled by development and another role filled by debugging and they exchange between these roles.

[9] For all of the dimensions of Test – Nominal (happy-path), Fault, Safety-Critical, Hazard-Critical, and so on. The very same requirement can exist in a Nominal Test Case as a Fault Test Case. Example: `R-02: UPON button push the X shall Y.` In the nominal case Y occurs when the button is pushed. The Fault case is the same except the prerequisites of the test perturb inputs to guarantee failure of Y occurring. In the latter, the test may appear to pass (Y failed to occur), but there was no requirement to detect those "perturbed inputs". This back and forth between Nominal, Fault, and other dimensions of test cases proliferates strong binding to the Requirements or it leaks new capabilities into the Software/Firmware un-tethered by requirements. The prime reason why *Design for Test* is bested by *Design for Flight*

- At the end of the course they have debugged with shell tools (and IDE if you insist), are aware of the back-end GDB server topologies (openocd, JLinkGDBServer …, they have written their Swiss Army knife tool which can source and sink Command and Control (C2) messages for debugging purposes into their benchmark embedded software application.

That keeps things squarely on the Debugging goal.
Otherwise the course can braid into all kinds of areas that may not appropriate for a ten-week undergraduate stand-alone course.

The Last Question you asked and answered cuts to the core of what I'd recommend. Amended:

If this a class that involves SW? If so, the last answer stands.

SW and HW, hand in hand, trying to bring up the board with flight (i.e., on target) "software" (firmware perhaps). More so when the target software (embedded) is multi-tasking, and complex and multi-staged. The danger is when HW teams develop their own day-0 software builds for the hardware fabs, assume things are fine. But the OS team or more dedicated application teams finds that the HW isn't right (devices cannot be driven, etc..). Hence the marriage between HW and SW effort.

Else:

If this class does not involve SW, then debugging is shifted towards a "DVE" Role (DVE is what Microsoft would label a Design Verification Engineer – a role where Requirements of the system are ingested, domain knowledge of the subsystem used, and literal go/no-go decisions made about the impact of the latest fab changes on the design).

If your class is the latter, then this is a different kind of Debugging role which uses software, but doesn't necessarily write target software.

Example: National Instruments scripts and programs to drive automation of test equipment. A DVE bread-and-butter job. But pure HW DVE role responsibilities is not my swim lane. I observe them, assist them, but not having the EE background to be a papered DVE, I cannot say for sure what the class would mean.

The concept of this class is exciting and encouraging. I hope that a version of this class (however you adjust it) is accepted and sections are filled to wait-list capacity.

Ever ready to help, send your ideas, of course. *–jdw*

Questions? `portside.shell190@passmail.net`

Acronyms

IDE	integrated development environment. 5
MCU	micro-controller unit. 2
RTOS	real time operating system. 2

Glossary

.elf Executable and Linkable Format; a common binary format for embedded executables and debug information. 9

.ld A common file extension for linker scripts used to control memory layout and section placement. 9

bit-twiddling Low-level manipulation of individual bits (masking, shifting, toggling) to control hardware or encode data. 4

Blinky A project/example name as the prototypical application that makes the LED blink – Usually a sample that is combined with the SDK from a vendor of an evaluation board. A representative embedded bring-up exercise.. 9

boot The process and early execution phase that begins after reset or power is applied and leads into normal operation. 9

breakpoint A debugger mechanism that halts execution when a specified location or condition is reached. 17

Compile-time The phase when source code is compiled; contrasted with run-time behavior and failures. 8

compiler A tool that translates source code into object code or executable code for a target platform. 5

const A C/C++ qualifier indicating an object should not be modified through that identifier. 4

Cortex-M3 An ARM Cortex-M family micro-controller-class CPU core commonly used in embedded systems. 11

CTx A class of requirement shorthand. "Critical To X" for some X that could be Customer, Function, Safety, etc.... Often classified as requirements that are marked (with attributes) to note they exist as line of last defense against defects not permitted to be part of the release software. Some bugs may be ignored (low priority, low reproducibility) but if a requirement (CTC) is mandated as a show stopper, it will supersede.. 7

DAL A classification used in safety-critical systems to indicate the required rigor of development, verification, and validation activities based on the severity of potential failure conditions. Higher DALs correspond to more severe consequences of failure and therefore mandate increasingly stringent processes, evidence, and independence in design and verification. . 8

De-bouncing Filtering or conditioning input signals (e.g., buttons) to remove transient oscillations and false transitions. 17

EV0 A label for an early engineering validation build used for initial testing and characterization. 4

Fab-A A label for an early fabrication build or board spin of hardware used for bring-up and integration. 4

fault An error condition detected by hardware or software that indicates an exceptional or illegal operation. 6, 9

firmware Low-level software closely coupled to hardware, commonly responsible for control and initialization of devices. 2

function pointers Pointers that reference functions, enabling indirect calls and callback-style interfaces. 4

gdb GNU Debugger; a command-line debugger used for inspecting and controlling program execution. 5, 9, 17

GPU/CPU A combined reference to graphics processing and central processing subsystems, often used when discussing packages or architectures. 6

Hardware-In-The-Loop A test approach where real hardware is integrated with simulated components to validate behavior under controlled conditions. 14

Hazard-Critical A descriptor indicating that failure can create hazards, driving stronger verification and assurance practices. 17

heap A memory region used for dynamic allocation during program execution. 4

Hello-world A minimal program or test used to validate basic toolchain, runtime, or I/O function. 14

instrumentation Adding measurement hooks (code or hardware) to observe execution, timing, and state during runtime. 6, 10

ISR's Interrupt Service Routines; code invoked asynchronously in response to interrupts. 4

J-Link A common family of hardware debug probes used for flashing and debugging via JTAG/SWD. 5

JLinkGDBServer A SEGGER server that connects GDB to a target via a J-Link debug probe. 18

JTAG A standardized hardware debug/test interface commonly used for boundary scan and on-chip debugging. Often used in lieu of the standard IEEE 1149.1 – IEEE Standard Test Access Port and Boundary-Scan Architecture.. 5

linker A tool that combines object files and libraries into an executable image and resolves symbol addresses. 9

MAC/PHY Media Access Control and Physical Layer; Ethernet interface components often discussed together in embedded networking. 11

map files Linker-generated files that summarize symbol placement and memory layout in the produced binary. 17

openocd Open On-Chip Debugger; a tool that bridges debug probes to targets via JTAG/SWD for flashing and debugging. 5, 9, 18

pin-muxed A configuration where physical pins are multiplexed among multiple peripheral functions and must be selected via configuration. 3

POR Power-On Reset; the reset condition and sequence that occurs when power is first applied. 9

single-stepping A debug mode that executes one instruction or source line at a time to observe control flow. 17

SRS Software Requirements Specification; Primarily defined in IEEE standard ISO/IEC/IEEE 29148 (replaced IEEE 830) for artifacts containing software requirements. . 4

SSS System of Systems Specification. A higher level requirement above SRS level. Examples: *The Coffee Maker shall brew coffee upon power button push.* Very high level, a lot of sub systems involved. Compared to SRS: *Upon detection of Button push, within 100ms, the Coffee Maker Heater Subsystem shall energize the Element subject to the ICD for heater element control.* A bit more specific, perhaps not as specific as we would like but, you get the idea.. 4

stack A memory region used to store call frames, return addresses, and local variables during execution. 4

startup-code Early initialization code (often assembly) that runs before the main application and sets up CPU/memory state. 11

static A C/C++ storage/class specifier affecting lifetime and linkage, often used for internal linkage or persistent storage. 4

SWD Serial Wire Debug; a two-wire debug interface commonly used on ARM-based micro-controllers as an alternative to JTAG. 5

symbol-table Metadata associating program symbols (functions/variables) with addresses, used for linking and debugging. 9

telemetry Runtime state and measurement data emitted by a system for observation, logging, or analysis. 4, 10, 14, 15

toolchain The set of build tools used together (compiler, assembler, linker, etc.) to produce an executable. 9

trade A research performed to compare and contrast multiple design solutions presented by the architecture. A Trade is a process where leading solutions considered are examined by rigor and studied for their cost, timeliness, and correct resolution to the architectural problem forced by the driving requirements of the system. A Trade can take several weeks to perform and involve many stakeholders within the organization.. 4

undefined-behavior Program behavior not defined by the language standard; outcomes may vary by compiler, optimization, or runtime conditions. 6

vector tables Tables of addresses used by a CPU to dispatch exceptions and interrupts to their handlers. 4

volatile A C/C++ qualifier indicating a value may change unexpectedly outside program control, preventing certain compiler optimizations. 4

Bibliography

M Ben-Ari. *Principles Concurrent Programming.* Prentice Hall, Philadelphia, PA, August 1982.

Dan Saks. What `const` really means. *Embedded Systems Programming*, 11(8):??–??, August 1998. URL `https://dansaks.com/articles/1998-08WhatconstReallyMeans.pdf`. 3 pp.

Sanjeev Shankar. Best practices for debugging embedded software. *European Journal of Computer Science and Information Technology*, 13(42):133–146, 2025. ISSN 2054-0957. DOI: 10.37745/ejcsit.2013/vol13n42133146. URL `https://doi.org/10.37745/ejcsit.2013/vol13n42133146`.

Richard H Thayer and Merlin Dorfman, editors. *Software Requirements Engineering.* Practitioners. IEEE Computer Society Press, Los Alamitos, CA, 2 edition, February 1997.