WizzDev

How to Minimize Firmware Bugs?

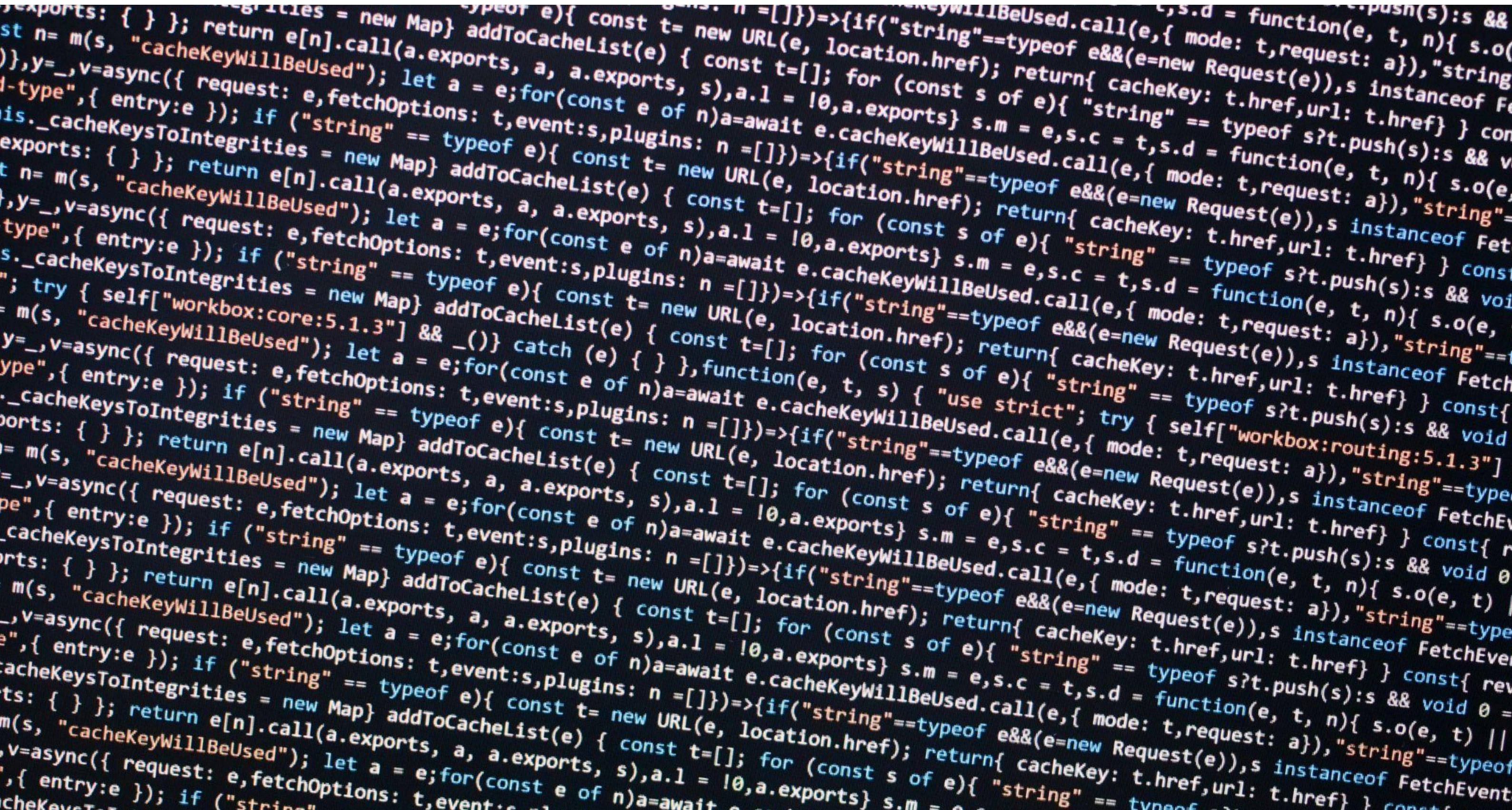

Table of content

Introduction

In today’s interconnected world, firmware is the invisible foundation behind smart devices in life sciences, smart homes, and urban infrastructure. Firmware bugs are not just inconvenient; they can stop device functionality, damage your brand, and delay time-to-market.

At WizzDev, we specialize in supporting companies that design and manufacture their own embedded hardware. Our unique combination of firmware expertise, electronics design, and cloud integration allows us to guide clients especially startups and mid-sized innovators through the complexities of building reliable, production-ready devices.

This whitepaper presents a practical, experience-based strategy to minimize firmware bugs. It is designed specifically for CTOs, Engineering Managers, and Product Owners seeking to build smart, secure, and cloud-connected devices. Whether you're prototyping a wearable or scaling a certified IoT system, our framework helps you reduce rework, ship with confidence, and exceed user expectations.

Problem

Bugs in firmware are not just technical issues, they are strategic liabilities. These issues don’t just stay technical they often ripple out, affecting everything from engineering workflows to how customers see your product. When devices reset out of nowhere or sensors start acting up, customers notice. And they lose trust fast. Meanwhile, addressing these issues post-production can be four to ten times more expensive than fixing them during development, draining engineering capacity and increasing support overhead. That’s why it is important to catch them as fast as we can. It isn't just smart engineering - it’s just plain good business.

WizzDev's 5-Step Approach to Bug Minimization

1

Clear & Aligned Requirements

A successful firmware project starts with crystal-clear requirements. At WizzDev, we place heavy emphasis on requirement engineering because it's often the most overlooked cause of bugs.

We collaborate directly with hardware designers, CTOs, and product owners to align expectations and identify possible risks during code development. This clarity prevents scope creep, miscommunication, and costly backtracking later on.

Key practices include:

- Creating and validating user stories and technical acceptance criteria.
- Functionality to hardware characteristics (e.g., hardware constraints and product requirements).
- Simulating real-world use cases early to identify potential failure paths.

We saw a great example of this recently with a client in the Smart Buildings space on a new wireless sensor. The initial spec was pretty straightforward, but when we started to drill down, we realized there were a few edge cases they hadn't considered. For example, what happens if the sensor loses power mid-transmission? By collaborating closely with their team and simulating different scenarios early, we were able to build a more robust solution from the very beginning. It meant we caught what could have been a nasty bug in the field, saving them a ton of money and a major headache later on.

We believe this shared understanding across teams is the most critical bug-reduction strategy there is. It's not just about the code; it's about making sure everyone is on the same page from day one.

2

Code Reviews

Code reviews are not just a formality. They are an essential quality gate. At WizzDev, we've embedded code review culture into our agile sprints and DevOps processes.

Our engineers conduct structured pull requests to:

- Catch logical errors and design flaws.
- Suggest a more robust and safer implementation.
- Enforce consistent style and safety-critical coding standards.

Implemented changes are merged into the repository only after passing multiple integrity checks, including peer review and successful build.

Every review session also serves as a learning opportunity, especially when junior developers pair with senior engineers. This mentorship fosters continuous improvement while ensuring that all code meets a consistent, reviewable standard.

We used this approach on a recent medical device project for a client, where a single line of code was responsible for handling a critical sensor reading. During a peer review, one of our engineers flagged a potential edge case where a timing issue could cause a data misread under specific load conditions.

The issue wasn't a bug in the traditional sense, but it had the potential to compromise the device's accuracy in a way that wouldn't have been caught by standard testing. By catching this subtle issue through our structured review process, we avoided what could have been a serious post-deployment problem and helped the client maintain their regulatory compliance.

Ultimately, rigorous code review is a force multiplier for long-term stability. It’s a core part of how we de-risk a project.

3

Automated Testing

Automation cannot cover everything in firmware, but it can accelerate confidence and protect against regressions. At WizzDev, we prioritize automation in a pragmatic, resource-aware way.

The main reasons for conducting automated testing:

- Consistency - once written tests can be run multiple times with the same result.
- Early detection of errors - unit tests allow bugs to be found at the code development stage.
- Stability - simple checking whether the new changes have not broken the existing functionalities.
- Documentation - well-written tests are the foundation of how the system should work.

Every major project requires a solid website, because in today's digital age, your online presence is often the first impression you make. In our company, we use the Selenium framework and Playwright to ensure that our websites remain robust and stable. By implementing these frameworks, we can cut regression time by 90%, cover most of the edge cases and increase overall test coverage. By allowing parallel test execution, we make sure that it will work in any web browser.

Since most modern IoT uses mobile apps for either control or monitoring, having a reliable and intuitive app interface is essential. For mobile app automation, we use the Appium framework, which allows us to interact with the app just like real users would. It supports both Android and iOS platforms, simulating actions such as taps, swipes, and text input to test the user interface and important functionalities. Beyond accuracy and consistency, automation with Appium drastically reduces regression testing time, enabling faster release cycles. It also allows us to execute complex test scenarios - like multi-device interactions, background app behavior, and performance under load that would be time-consuming or nearly impossible to perform manually.

While manual testing remains important, automation ensures that every build remains predictable and testable.

4

Continuous Integration

Our CI philosophy is simple: no code gets merged without passing automated gates. It's our primary safety net, the first line of defence against bugs and integration conflicts that can derail a project.

Pros of Continuous Integration:

- Early and continuous bug detection - potential bugs and errors are noticed immediately after pushing the new code, rather than just before release.
- Better code quality - automated testing and static code analysis help maintain standards.
- Faster releases - changes can be implemented more quickly because of automated testing and building.
- Higher productivity for developers - automation of manual tasks allows programmers to focus on writing new code and solving problems instead of repeating routine activities.

It's about more than just running tests. Our CI pipelines are designed to automatically build the firmware, run a full test suite, and even perform static code analysis. We use Jenkins and GitHub actions, which allow us to automate these checks every time a developer pushes a new commit to a feature branch. It's a bit like having a silent, tireless co-worker who instantly validates everything we do. This immediate feedback loop helps us catch issues within minutes, not days or weeks.

For example, recently an engineer implemented a feature, which resulted in a memory leak. Our CI pipeline flagged the issue automatically during the static analysis check, preventing the code from ever making it to the main branch. This resulted in saving the development team from a hard-to-debug problem. Moreover, it was a valuable lesson for our engineers to help them understand the impact of their code quality in a live system.

This isn't just about catching errors. It's about making sure our team's work is always in sync and that our codebase is always in a deployable state. It's how we ensure we're ready for new features and updates, without the chaos.

5

User Acceptance Testing

Nothing replaces testing with real users on real hardware in real conditions.

User Acceptance Testing (UAT) validates whether firmware meets the expectations of the field beyond specs and test cases. It captures nuances like:

- Device functionality and user experience with a user, not a developer.
- Device behaviour under environmental conditions (e.g., heat, vibration, signal loss).
- Usability friction points (e.g., button timing, LED feedback).
- Domain-specific edge cases not easily reproduced in the lab.

Just recently, we worked on a project for a client in the Medical Devices space, developing firmware for a portable diagnostic device. Everything worked perfectly in our lab, the sensor data was accurate, the communication was stable, and the battery life was great on paper. But we knew that wasn't enough. We established a UAT phase, during which a small batch of devices was provided to a clinical team for a few weeks to use in a real-world setting.

What we found was fascinating. One of the primary issues was with the button press. The UAT showed that nurses, wearing gloves and moving quickly between patients, needed a more tactile and definite haptic feedback to confirm the device had registered the input. This wasn't something we had a spec for, and it wasn't a bug in the traditional sense, but it was a crucial usability issue that could impact workflow. Thanks to the UAT, we were able to pinpoint the problem and adjust the firmware's haptic feedback to make it more reliable for clinical staff.

This kind of real-world feedback is required to ensure that the device doesn’t just work, but is useful for the customer and improves their life by providing a better and robust experience.

Lessons Learned Across Projects

At WizzDev, our firmware methodology isn't theoretical; it's battle-tested across smart home platforms, health monitoring devices, and city-scale IoT systems. As we help product teams move from prototype to production, a few recurring lessons always stand out:

#1

No amount of testing can compensate for vague requirments.

Bugs often stem from misaligned assumptions, not faulty code. Achieving crystal-clear requirements upfront is paramount to preventing costly rework and ensuring project success.

#2

If it's not tested on real hardware, it's not tested.

Hardware edge cases like power brownouts, thermal drift, and sensor latency are nearly impossible to simulate fully. Real-world testing on physical devices is non-negotiable for robust firmware.

#3

Continuous Integration is the best long-term investment most startups ignore.

Skipping continuous integration early on creates significant bottlenecks and fosters "tribal knowledge" that becomes incredibly difficult to scale as your team and codebase grow.

#4

Quality improves when firmware, hardware, and cloud teams collaborate.

We've seen major defects resolved simply by syncing a hardware register map with a backend expectation. Breaking down silos and fostering cross-functional communication is key to success.

These lessons reinforce that process alone isn't enough - what matters is how it's applied.

What Clients Ask Us Most

After implementing our 5-step approach across dozens of projects in sectors like smart home, smart city, life sciences, and medtech, one pattern is clear: even the best strategy needs to be adaptable.

In conversations with CTOs, product leads, and firmware developers, we consistently hear the same kinds of questions:

- How much time should be spent on testing vs. development?
- Can we produce high-quality firmware without automated testing?
- Is it possible to reduce bugs if we don’t have a dedicated QA team?

The next section distills the answers we give our clients and the thinking that underpins them. Whether you’re scaling your engineering team or building your first connected product, these insights will help you apply the 5-step framework in real-world conditions.

Frequently Asked Questions

Q1: How much time should we spend on testing versus development?

In firmware projects, especially those involving custom hardware, we recommend allocating 40–50% of total development time to testing. This includes both manual and automated tests.

- 50–60%: Feature development and refactoring
- 20–30%: Automated testing (unit, integration, regression)
- 20–30%: Manual testing (exploratory, edge-case, hardware-specific)

The exact ratio depends on your project's complexity, hardware maturity, and certification requirements. Underinvesting in testing almost always leads to delays and instability.

Q2: Can we ship stable firmware without automated testing?

It's possible, but not advisable. Without automation:

- Manual regression testing becomes unscalable
- Bug detection depends heavily on human discipline
- Confidence in releases is reduced

In resource-constrained projects, we help clients prioritize the most critical automated tests first e.g., boot logic, sensor communication, and OTA flows while supporting manual testing for UX and edge scenarios.

Q3: What if we don't have a dedicated QA team?

No problem. WizzDev often acts as an extension of your team, providing:

- QA support for test case design
- Tools for flashing, logging, and capturing field data
- Exploratory testing in late-stage builds

In smaller teams, developers can conduct structured manual tests using documented checklists and logs supported by WizzDev's infrastructure.

Q4: Is 100% test coverage realistic?

In embedded systems, 100% is rare and often unnecessary. The goal is risk-based coverage:

- High for safety-critical or business-critical module
- Moderate for peripheral or less volatile functions
- Manual testing as a supplement for hard-to-automate flows - for example, all flows regarding hardware interactions are very hard to test

We aim for 60–80% automated coverage where feasible, and supplement with targeted manual test passes.

Q5: Can WizzDev support test automation setup?

Absolutely. We help clients:

- Set up CI/CD pipelines that include unit and integration tests
- Design HIL (hardware-in-the-loop) test setups Integrate tools like Unity, Ceedling, or custom Python frameworks

Additionally, we can improve your development setup by integrating tests with your git repository, so you can be sure that only working features are merged.

Q6: How do we validate firmware on real hardware?

We build repeatable test environments using:

- Test jigs for GPIO, UART, I2C validation
- Power supply triggers and signal generators for stress testing
- Cloud-integrated logging for field testing
- Extensive experience with hardware prototyping and hardware debugging

WizzDev also provides test tooling recommendations and validation checklists for engineering teams.

About WizzDev

WizzDev is a team of passionate technologists dedicated to building advanced software and embedded systems for innovative IoT solutions. As proud AWS partners, we transform creative ideas into market-ready products tailored for a fast-evolving world.

We focus on delivering software that's not only functional but also future-proof. Our experienced team excels at optimizing complex hardware and aligning it with the latest tech trends and upcoming challenges.

From electronics design and firmware development to PCB prototyping, our versatility is our strength. With an Agile mindset, we adapt quickly to changing requirements—crucial for companies aiming to scale fast.

Our collaboration with AWS gives our clients access to cutting-edge tools and resources, enabling us to build reliable, cloud-integrated IoT solutions. We've contributed to the growth of sectors like smart home, medtech, bioscience, and automotive.

At WizzDev, we believe innovation drives success. That's why we embrace every challenge with enthusiasm, delivering solutions that often exceed expectations. Let's build the future together.

Contact Us

E-mail: info@wizzdev.pl
Phone: +48 789 395 645

Address:
WizzDev P.S.A.
Reymonta 5/3
60-791 Poznań
Poland