Debugging beyond the screen, challenges of physical computing

Summary

Debugging physical computing spans three 'stuck points': hardware, software, and the connections between them. Naming these turns guesswork into structured inquiry.

Debugging is a distinct skill with invisible processes:

- Explore the problems and opportunities unique to physical computing
- Use analogy, metaphor, and roleplay to build mental models or notional machines

Design activities that require and promote troubleshooting:

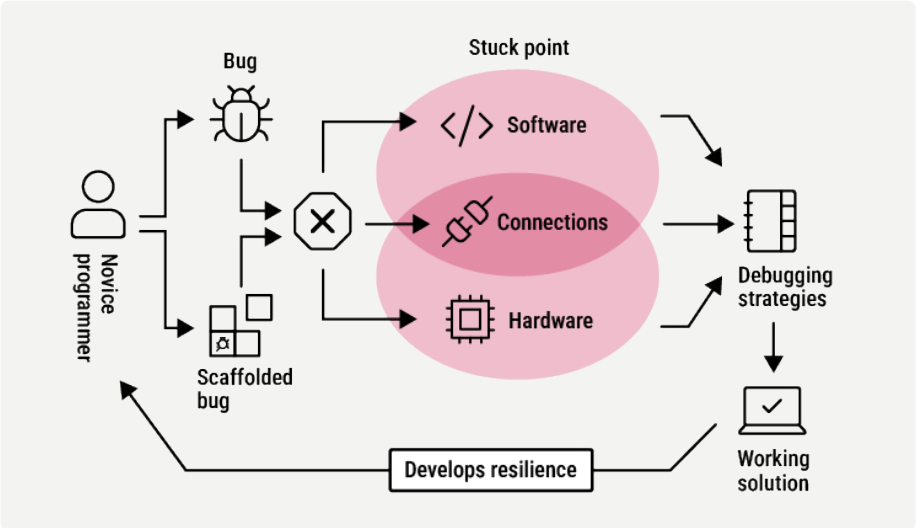
- Build failure artefacts: projects with deliberate bugs
- Let learners create bugs for peers

Failure is normal; frustration often precedes persistence:

- Choose challenges that lead to learning
- Persist after setbacks
- Praise effort and constant improvement
- Model resilience, methodical problem solving, and comfort with failure
- Celebrate errors as learning opportunities

Introduction

When we talk about debugging, we often only think about finding and fixing errors in code. However, when learners debug their physical computing projects, they face different challenges to fixing code on a screen. Errors in code may still occur, but there's also the potential for errors in circuits, in connections between hardware and software, or as a result of faulty mental models of programming. A loose wire and a typo can look identical from the outside and often the hardest part is working out which it is. Understanding these distinctive challenges is therefore an important step in delivering effective physical computing learning.



Additional physical computing challenges related to debugging

Debugging beyond the screen

Physical computing projects usually include some combination of programmable devices, circuits, sensors, and craft materials. As a result, learners encounter debugging challenges that do not arise in screen-only projects.

As Fields and colleagues observe, "extending debugging beyond the screen makes things more complicated for learners who must not only attend to a variety of code-based errors, but also physical errors such as incorrect placement of a sensor on a robot or incorrect circuitry in an e-textiles project."[1] Distinguishing between the types of error requires a different kind of reasoning than reading an error message.

However, this complexity is not a reason to avoid physical computing. Instead, it is a reason to teach debugging explicitly within that context. Many researchers have discussed how debugging is a distinct skill that needs deliberate instruction, one that includes systematic strategies, not just reactive problem solving.[2] Learners who develop these troubleshooting skills in physical computing contexts must reason simultaneously about code, hardware, and the interactions between them, building a systems-level understanding that screen-only debugging does not require.

Mapping the stuck points

Further research by Hennessy Elliott and colleagues identifies three categories of error:[3]

- Hardware errors - problems with physical components or their connections
- Software errors - bugs in code
- Hardware/software connection errors - mismatches in how programs and circuits interact

This last category is distinctive to physical computing and is particularly challenging, because the source of the problem spans two domains at once.

In practice, naming these categories for learners, and explicitly teaching strategies for each, helps move troubleshooting from an anxious and unfocused activity towards structured inquiry.[2] Even a simple prompt such as "is this a hardware problem, a software problem, or something in between?" gives learners a productive starting point.

Teaching the invisible machine

A major source of difficulty in physical computing is that the processes connecting programs to outcomes cannot be seen. Variable values, execution flow, pin states: all are invisible at runtime. Without a working mental model of these processes, learners have no clear basis for systematic debugging and may only try things at random.

Notional machines are simplified, pedagogically useful representations of program dynamics that can help learners understand these invisible processes. Notional machines are not fixed, but can evolve during learning. Concrete, real-world analogies can become more computing-specific representations as understanding develops.

Jayathirtha[4] documents how one experienced teacher used various notional machines in practice, including:

- Verbal analogies, such as the program as a play, with variables as characters and the microcontroller as the director
- Participatory roleplays in which students physically took on the roles of program elements and acted out its execution step by step

Crucially, the roleplays made learner thinking visible — the teacher could observe misconceptions in action and address them in the moment.

The emotional landscape of debugging

Debugging is not only a cognitive activity, it also depends on affective factors such as resilience, patience, and mindset. Morales-Navarro and colleagues[5] observed five growth mindset practices in learners during physical computing debugging:

- Choosing challenges that lead to more learning
- Persisting after setbacks
- Giving and valuing praise for effort
- Approaching learning as constant improvement
- Developing comfort with failure

Strikingly, frustration was frequently observed to precede persistence rather than block it. What may look like a learner disengaging is often the moment before they try again.

To support learners effectively, teachers need to consider both emotional and practical aspects. Rich and Strickland[2] argue for fostering a productive debugging culture in which errors are framed as an expected and generative part of learning rather than evidence of failure. In practice, this can look like focusing on the troubleshooting steps required rather than simply reaching a solution quickly and letting learners lead the problem-solving process.

Designing for debugging

Learners will usually encounter debugging at some point in a physical computing project, but you can also deliberately design projects that encourage your learners to focus on this process. Fields and colleagues[1] describe 'failure artefacts', projects intentionally designed with bugs in them, which learners work to identify and fix. This approach lets you control the type and location of errors, and to scaffold the experience in ways that accidental faults do not permit.

Building on this, Morales-Navarro and colleagues[5] discuss 'Debugging by design', a student-led activity in which learners create their own bug-containing projects for peers to solve. They observed growth mindset practices more often in peer-to-peer interactions, suggesting that the social dimension of debugging is as important as the technical one. When learners must articulate and construct a fault for someone else to find, they deepen their own understanding of how systems fail, and how to reason about fixing them.

References

[1] Fields, D. A. et al. (2016). Deconstruction kits for learning: Students' collaborative debugging of electronic textile designs. the-cc.io/qr34_1

[2] Rich, K. M. and Strickland, C. (2020). Testing and debugging. the-cc.io/qr34_2

[3] Hennessy Elliott, C. et al. (2023). Toward a debugging pedagogy: Helping students learn to get unstuck with physical computing systems. the-cc.io/qr34_3

[4] Jayathirtha, G. (2022). Video analysis of a teacher's use of notional machines in an introductory high school electronic textile unit. the-cc.io/qr34_4

[5] Morales-Navarro, L. et al. (2025). Understanding growth mindset practices in an introductory physical computing classroom: High school students' engagement with debugging by design activities. the-cc.io/qr34_5

This resource by the Raspberry Pi Foundation is licensed under CC BY-NC-SA 4.0. To view a copy of this licence, visit https://creativecommons.org/licenses/by-nc-sa/4.0/.

Download this resource at the-cc.io/qr34
Last updated September 2026
Got feedback? the-cc.io/qrfeedback