Language-Agnostic Debugging for Microcontrollers

Carlos Rojas Castillo
Vrije Universiteit Brussel
Brussels, Belgium
carlos.javier.rojas.castillo@vub.be

Matteo Marra
Nokia Bell Labs
Antwerp, Belgium
matteo.marra@nokia-bell-labs.com

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Abstract

With the advent of WebAssembly (Wasm), programming microcontrollers (MCUs) has become possible by leveraging on a wide range of languages (e.g., Rust, AssemblyScript, C, C#, Go, C++) that compile to Wasm. However, current WebAssembly debugging support is still in early development and is designed for applications running on desktop machines, making it too resource-intensive for MCUs. While DWARF and OpenOCD have facilitated language-agnsotic debugging for languages like Rust, Go, and C, these solutions are limited to languages that compile to native machine code and fail to target IoT systems. Consequently, IoT systems often undergo only partial debugging, increasing the likelihood of severe and frequent concurrency and communication bugs.

In this position paper, we explore the challenges and issues associated with language-agnostic debugging. We identify several key requirements for effective language-agnostic debugging, such as the need for over-the-air debugging and the ability to perform distributed debugging operations. Additionally, we present an envisioned language-agnostic debugging approach based on WebAssembly, designed to support the debugging of large-scale distributed IoT systems.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Virtual machines*; • **Computer systems organization** → **Embedded software**.

Keywords

MCU, IoT, Distributed, VM, WebAssembly, Embedded Devices

ACM Reference Format:
Carlos Rojas Castillo, Matteo Marra, and Elisa Gonzalez Boix. 2024. Language-Agnostic Debugging for Microcontrollers. In *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques (DEBT '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3678720.3685317

1 Introduction

Microcontrollers (MCUs), such as the ESP32 [11], are hardware-constrained computers used to build *Internet of Things* (IoT) systems in domains such as smart cities, smart hospitals, and more. With the advancements in MCU hardware capabilities, modern MCUs are no longer limited to functioning solely as sensors. Instead, they have now the ability to perform *smart computation* such as analytics based on Machine Learning models [8].

While C and C++ have been the de facto standard for programming MCUs, over the last years, managed programming languages such as JavaScript [12, 28], Python [14], and Erlang [4], have gradually targeted MCUs through the use of lightweight Virtual Machines (VMs), or compilers that generate native code that runs on MCUs [32, 37]. As a result, modern IoT systems are complex distributed systems composed of numerous MCUs interacting with one another, through different communication channels [25] (e.g., Wi-Fi, Zigbee, LoRa) that perform smart computation.

As more and more programming languages (e.g., Rust, Go, Python) target MCUs, the need for suitable testing and debugging support for these languages becomes necessary. A large number of existing debugging support, however, is only available for C and C++ applications [7, 36, 40], whereas debugging support for other languages is scarce, if not missing. Several VMs targeting MCUs, such as Espruino [12] and WARDuino [17], offer debuggers with classical debugging operations, such as step into and step over. Numerous MCU VMs [4, 14, 28, 39], however, lack any debugging support. And as existing desktop debuggers cannot be used to debug MCUs due the hardware-constraints, debugging support has to be implemented over and over again for each new VM or language targeting MCUs.

To the best of our knowledge, OpenOCD [27] has emerged as the only language-agnostic debugging solution for MCUs, supporting several programming languages including Rust, Go, C, and C++. However, OpenOCD is specifically designed to debug applications that compile to native MCU machine code, making it unsuitable for debugging applications that compile to bytecode run on a MCU runtime [12, 14, 28, 39]. Additionally, OpenOCD was designed to target single MCUs, making it unfit for language-agnostic debugging of an entire IoT system. As a result, there is a lack of language-agnostic debugging support that can target bytecode-compiled languages and modern IoT systems.

In this paper, we explore the challenges of providing language-agnostic debugging able to target both individual MCUs, similar to OpenOCD, and modern IoT systems composed of numerous MCUs running different programming languages. We envision a solution that builds on top of *WebAssembly* (*Wasm*) [18], which is a binary instruction format that serves as a compilation target for numerous programming languages, including Go, C, C++, Rust, and AssemblyScript. By enabling language-agnostic debugging for WebAssembly, we believe that it is possible to target all the languages that compile to WebAssembly.

Designing a (WebAssembly) language-agnostic debugger for MCUs comes with several challenges:

DEBT '24, September 19, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1110-7/24/09
https://doi.org/10.1145/3678720.3685317

Hardware-Constraints Debuggers targeting programs running on MCUs need to account for the MCU hardware restrictions. These restrictions require software to be written in a way that minimizes memory consumption, computation power, and energy consumption. According to a 2021 study [24], 63% out of 193 surveyed IoT developers find this task very challenging.

Over-The-Air Debugging MCUs are not always physically accessible as they can be deployed in remote locations. This forces tooling support to be open for over-the-air debugging via communication channels such as Wi-Fi, BLE, and Zigbee. Additionally, debugging support should also be provided in a way that minimizes over-the-air communication as this tends to drain the battery life of MCUs [5].

Distributed and Scalable IoT systems are inherently distributed and can scale to thousands of MCUs that (partially) interact with each other to provide a service. Implementing a debugger for such systems requires debuggers that can deal with many MCUs and coordinate debugging operations across groups or all of them. Additionally, large-scale debugging operations must be carefully designed to minimise network overhead and system interruptions.

MCUs Run Different Languages Modern IoT systems may consist of MCUs running the same or different languages. Consequently, debuggers must ensure that debug operations remain applicable across various MCUs, regardless of the programming language used on each MCU. This is challenging because debugging operations are often tied to a programming language, and some operations are only relevant to some programming languages.

In the remainder of this paper, we will first elaborate on WebAssembly (section 2) and the opportunities and challenges that arise from using it as a language-agnostic foundation. Then we will overview the problem statement (section 3) and our envisioned objectives (section 4).

2 Debugging WebAssembly on MCUs

We discuss what makes WebAssembly interesting for targeting MCUs, the current approaches for debugging WebAssembly, their limitations, and the challenges linked to implementing debuggers for WebAssembly.

2.1 WebAssembly for MCUs

WebAssembly presents several strengths that make it a suitable candidate for language-agnostic debugging of MCUs.

First, numerous programming languages (e.g., Rust, C, C++, Go, AssemblyScript, F#), already compile to WebAssembly while many more are gradually targeting it [3]. The debugging support can potentially target all of these languages that do compile to WebAssembly. Second, WebAssembly's bytecode has been intentionally designed to be compact and small in size [18], making it suitable for hardware-restricted devices such as MCUs. Finally, WebAssembly is hardware-independent [18], meaning that as long as a WebAssembly runtime is available for a target hardware. WebAssembly applications can, as far as we know, with little to no compiler configuration, target the hardware via the runtime.

In the case of MCUs, several WebAssembly runtimes (e.g., WARDuino [17], Wasm3 [39], WAMR [2]) can already target MCUs. This enables many languages that compile to Wasm to also run on MCUs.

2.2 WebAssembly Debugging Formats

To debug WebAssembly applications, compilers typically generate debugging information according to two debugging formats:

Source Map Spec The *Source Map Spec* [33] was originally introduced for debugging TypeScript applications transpiled to JavaScript on the Web. WebAssembly language implementers have typically opted to generate this debugging information when targeting the Web because it is natively supported by all major browser vendors. WebAssembly-compatible languages that generate this format include AssemblyScript and Rust, where Rust relies on an external tool to produce it [26].

DWARF DWARF [9] was originally introduced for debugging ELF-binary files, a standard executable file format for the UNIX OS, and has since been extended to other OSs. It is typically the debugging format of choice when compiling applications to run on desktop machines. Currently, several debuggers rely on DWARF, such as GDB [15], LLDB [23], and OpenOCD [27] for MCUs. WebAssembly-compatible languages that generate this format include Rust, Go, and Zig.

We have identified a number of limitations that arise when using such formats for building debuggers.

Source Map Spec Lacks Debugging Information. The Source Map Spec does not provide sufficient debugging information for source-level debugging. In particular, it cannot map WebAssembly bytecode level state to source-level state. Consequently, Source Map Spec debuggers, such as those in browsers, are forced to provide debugging support at the level of the WebAssembly bytecode. For instance, if the application developer pauses the debugger at a specific source-location, the debugger cannot show the content of variables in-scope (e.g., object fields, string). Instead, it only shows bytecode-level content (e.g., linear memory, stack). To view source-level variables, tool users must mentally reconstruct their state from the bytecode-level content.

DWARF Requires Machine to Source-Level Adaptors. Unlike the Source Map Spec, DWARF provides debugging information that enables source-level debugging. Debuggers that rely on DWARF (e.g., OpenOCD, GDB, LLDB), however, have been primarily designed for applications that compile to native machine code. Using these debuggers for bytecode applications, such as WebAssembly, requires the runtime implementers to create mappers that translate machine-level operations to source-level debug operations. For instance, a *step into* operation, which advances computation to the next source-location, requires among others runtime implementers to ensure that the PC pointing to the machine code of the runtime stops when the *instruction PC* (i.e., the PC used internally by the runtime to point to a bytecode instruction) reaches the instruction corresponding to the next source-location. Two mappings are required: one from machine code to bytecode-level and one from bytecode-level to source-code level. This effort must be repeated for each debug operation and WebAssembly runtime that wishes to integrate a DWARF-based debugger.

Debugging Formats Are Resource-Intensive. Regardless of the generated debugging format, the obtained debugging information is too large for storing on MCUs. To showcase this issue, consider the small *Blink Led* example written in Rust depicted in Figure 1. The application repetitively turns a led on and off after a constant period of time. When compiling the example to WebAssembly with the debugging flags enabled[1], the Rust compiler generates a *Wasm module* (i.e., the WebAssembly bytecode that can be given to a WebAssembly runtime) extended with DWARF-based debugging information. The obtained module totals a size of 4.5 KB where approximately *84%* of the size is due to the debugging information. As is, this module cannot be deployed on a MCU due to its limited storage capabilities (e.g., the M5StickC [34] has 4 MB of Flash Memory and 520 KB of SRAM). Additionally, the need to deploy along the module extra software, such as the WebAssembly runtime and OS, further reduces the available storage space. Similarly, the Source Map Spec debugging format is also heavily weighted.

While tools are available to trim the debugging information from the Wasm module [1], the large size of the debugging information has implications for the design of MCU debugging support. Since MCUs cannot store debugging information, WebAssembly debuggers must (1) operate from an external machine where the debugging information is held and (2) map debugging operations from source-level to bytecode-level, as MCU WebAssembly runtimes lack direct access to source-level debugging information.

```
1   // ...
2
3   pub fn delay(ms: u32) {
4       unsafe{
5           // extern env function
6           chip_delay(ms);
7       }
8   }
9
10  #[no_mangle]
11  pub fn main(){
12      const LED: u32 = 10;
13      const SLEEP: u32 = 1000;
14      const OUTPUT: u8 = 2;
15      const ON: u8 = 1;
16      const OFF: u8 = 0;
17
18      pin_mode(LED, OUTPUT);
19
20      loop {
21          digital_write(LED, ON);
22          delay(SLEEP);
23          digital_write(LED, OFF);
24          delay(SLEEP);
25      }
26  }
```

Figure 1: Rust Blink Led Example that at each iteration turns *ON* and *OFF* a *LED* after *SLEEP* ms. The functions called in the *main* are implemented in terms of function calls imported from the WARDuino [17] runtime.

2.3 Challenges for Debugging WebAssembly

With the aforementioned limitations, we can observe several challenges.

Variability in Debugging Information. Overall, the number of debugging formats, the differences in debugging information provided by the formats, as-well-as the potential inconsistencies within a debugging format (e.g., DWARF content may vary depending on the compiler [9, 22]) pose significant challenges for the implementation of language-agnostic debuggers for WebAssembly. On one hand, debuggers must account for different debugging formats to ensure compatibility with any language. On the other hand, debuggers need to offer consistent debugging support despite the different depths and inconsistencies of the debugging information provided by a debugging format.

Map Source-Level to Bytecode-Level Operations. The large size of the debugging information forces WebAssembly debuggers to be implemented as components that primarily drive the debugging behaviour externally from the MCU. This new design poses challenges for the implementation of debuggers. In particular, the implementation of some debug operations (e.g., *break on line 18*, *step into call*) will have to primarily occur on an external machine. Where the external machine will have to find a way how to map source-level operations to bytecode-level operations. This is not trivial as several debug operations are language-dependent and the connection between the bytecode and language syntax or semantics is not necessarily clear.

High Debugging Runtime Overhead. The large size of the debugging formats has also implications on the runtime performance of the debugging support. In particular, as each debug operation applied on the runtime happens at the bytecode-level, source-level debugging can become highly inefficient. For instance, a source-level *step* operation that leads to the next line of source code may require 10 consecutive *step* operations at the level of WebAssembly. This can result in drastic overhead, particularly, when implementing more advanced debug operations and debugging over-the-air.

3 Problem Statement

In what follows, we identify several problems that hinder the debugging of a single MCU or an entire IoT system.

Redundant Modus Operandi for Debugger Implementation. Existing desktop debugging tools are not tailored to the unique characteristics of MCUs or IoT systems, particularly, the limited hardware capabilities of MCUs [24]. As a result, application developers cannot rely on desktop debuggers to target MCUs. Instead, tool implementers have to continuously invest effort in implementing new debuggers that target MCUs. With the rise in lightweight VMs for MCUs [4, 12, 14, 28, 39], this effort must be repeated over and over again for each new programming language that targets MCUs. While implementing a new debugger per language can result in a debugger fully tailored to that target language, particularly when

[1] rustc -C link-self-contained=no -C link-args=--no-entry -C link-args=-zstack-size=32768 --target wasm32-unknown-unknown -g blink.rs

the tool implementers control the VM implementation. This approach, however, does not scale to IoT systems running different languages, as it would require the language-dependent debugger to accommodate all possible MCU VMs and languages.

Unexisting Debugging Support Leaves Systems Partially Untested. Modern IoT systems can consist of MCUs running different languages. When debugging such systems, application developers are potentially forced to alternate between the available debuggers to target their desired languages.

However, debugging support is not always available for every language [4, 14, 28, 39], which forces application developers to rely on *log-based debugging* [24], or on tools that are not tailored for source-level debugging of MCU bytecode applications such as OpenOCD [27]. Therefore, as long as debugging support is nonexistent for some languages, it is very hard for application developers to debug all parts of an IoT system.

(Advanced) Debug Operations Are Language-Dependent. Debugging support for MCUs has already been in place for a couple of years [12, 20, 27, 31, 40]. However, not every debugger provides the same level of debug operations. Although many do provide debuggers with classical debug operations such as *step into*, *step over*, and more, only a few [20, 40] provide advanced debugging operations highly beneficial for MCUs. A 2012 study by Britton et al. [6] has shown that having access to advanced debug tool operations, such as back-in-time debugging, is crucial to significantly reduce debugging time and development costs. In the literature, we can find some examples: Clairvoyant [40] provides the ability to trigger interrupt handlers on demand which is highly beneficial as MCUs applications are *interrupt-driven*. Moreover, recent work [20, 31] has made it possible to reduce debugging overhead on MCUs and enabled *step-back debugging* on MCUs.

However, by design, these (advanced) debug operations are only available to the languages for which the debugger was designed. Other languages cannot benefit from them. Instead, now application developers are potentially forced to decide on the programming language based on the available debugging support.

Incompatible Bytecode-Level Debugging Support. Currently, most of the available debugging support has been designed to target applications that compile to MCU machine code. For instance, openOCD [27], which is the dominant approach for debugging MCUs, allows us to debug applications written in languages such as C, C++, Go [37], and Rust [32] since compilers exist that generate native machine code for MCUs such as the ESP32 [10].

However, debugging bytecode applications run on top of a VM is not possible at the source-level. With OpenOCD application developers will only be able to debug the VM and not their application. More specifically, any debug operation applied will be applied to the VM's execution and not at the source-level of the bytecode. As a result, tool users need to manually perform mental adaptions between machine-level to source-level code when using OpenOCD.

Lack of Distributed and Scalable Debugging Support. Many of the debuggers that target IoT systems such as Sympathy [30], MontiThings [19], and IoTReplay [13], are designed for *Wireless Sensor Networks* (WSN) [35] where MCUs are sensors that periodically send telemetry data to central servers. Troubleshooting WSNs is primarily achieved by (passively or actively) observing network activity. However, modern IoT systems are composed of MCUs that perform smart computation and may require complex interactions to enable a distributed service. Thus introducing the need for debuggers able to debug individual MCUs or subsets of a System.

To the best of our knowledge, Clairvoyant [40] is one of the few debuggers capable of targeting individual MCUs or subsets of a system as it provides debug operations that apply to both individual MCUs (e.g., *break*, *step*) or the whole system (e.g., *global break*, *global continue*). However, the system-wide debug operation of Clairvoyant are 5 operations that apply to all MCUs of a system. There is no support for distributed debug operations that can for instance be applied to a selection of MCUs (e.g., *break MCU x and y on message received from MCU z*). These kinds of distributed debug operations are crucial to help detect concurrency bugs which according to several studies [21, 24, 38] are considered to be frequent and severe in IoT systems.

Debugging Formats Require Additional Hardware and Adaptors. Debugging formats, such as DWARF [9] and the Source Map Spec [33], are memory-intensive and therefore cannot be stored on MCUs. As a result, MCU debuggers using such debugging formats are forced to operate from an external machine. For instance, in the case of OpenOCD, the debugging software is deployed on a separate machine and it connects physically to a MCU to enable debugging via the JTAG [16] interface usually physically present on a MCU.

However, this setup becomes rapidly costly and unpractical when transitioning to over-the-air debugging. This is because these debuggers need to maintain a physical connection to the MCU for debugging. For instance, in the case of OpenOCD, when transitioning to over-the-air debugging, the physical connection to the MCU needs to be maintained between OpenOCD and the MCU. This implies for instance that for a desktop machine to be able to debug a MCU over Wi-Fi, additional hardware (e.g., Raspberry Pi [29]) with Wi-Fi capabilities needs to be deployed per MCU. The hardware has then the responsibility to accept client-side desktop connections and run the OpenOCD software that applies debug operations to the MCU.

Additionally, as discussed in section 2, the use of DWARF requires the implementation of adaptors that translate between machine-level debug operations to source-level operations to make existing debugging support target bytecode applications.

4 Envisioned Language-Agnostic Debugging

Based on the aforementioned problems (section 3) and WebAssembly debugging challenges (section 2), we formulate requirements that a debugger should exhibit to enable WebAssembly language-agnostic debugging tailored to MCUs and IoT systems:

Truly Language-Agnostic Debugger. A truly language-agnostic debugger is a debugger that can target any language that compiles to WebAssembly. To achieve this objective, the envisioned techniques should not make any assumption on the generated debugging format as this can vary per language and compiler.

Additionally, a truly language-agnostic debugger ensures that debug operations (e.g., *step into*, *step over*) are applicable across all languages that compile to WebAssembly. The operations should not be lost when targeting a different language. However, some debug operations are language-dependent and only applicable to specific sets of languages. For instance, a debug operation that *breaks on instance creation*, is only applicable to object-oriented languages. Therefore, the debugger should provide a way to (1) distinguish between language dependent and independent debug operations, (2) apply language-dependent operations in a language-agnostic manner, and (3) implement a mechanism to selectively enable or disable debug operations based on the target language.

Portability of Debugging Support Across WebAssembly Runtimes. The investigated language-agnostic debugging techniques should be portable to other MCU WebAssembly runtimes. This is particularly important as it eases the integration process of debugging support into a WebAssembly runtime, thus reducing the need for runtime implementers to reimplement existing debugging support over and over again.

Over-the-air Debugging. The debugger should be designed in a way to enable over-the-air debugging. However, this capability introduces additional latency, which must be carefully accounted for, especially when debugging IoT systems. For instance, reducing the need for communication with the WebAssembly runtime can help minimise the debugging latency.

Source-Level to Bytecode-Level Mappings. As the debugging formats are too memory-intensive to store on MCUs. The implemented debugger should therefore primarily run on a desktop machine and function under the assumption that MCU WebAssembly runtimes only operate at the bytecode-level, not at the source-level. As a consequence, when performing debug operations, the debugger will have to compute the behaviour of most of the debug operations on the desktop machine and apply them on the MCU runtime when needed. Part of this calculation involves mapping debug operations between source-level code to the WebAssembly bytecode-level.

Distributed and Scalable Debug Operations. Due to the scalability and distribution nature of IoT systems, debug operations should easily target all or some MCUs part of a system. The debug operations should also account for the possibility that they may be applied jointly upon several MCUs. For this, we anticipate the need for coordination mechanisms between those MCUs. In doing this, we will be able to introduce distributed debug operations that can help with the debugging of concurrency bugs that are omnipresent in modern IoT systems.

System-wide Language-Agnostic Debugging. The debugger should be able to dynamically target different programming languages and smoothly transition between them. This is because when debugging an IoT system where MCUs may run different languages, debug operations need to remain language-agnostic. In particular, we anticipate two kinds of debug operations for which language-independence needs to be preserved: (1) debug operations that target individual MCUs part of a system, and (2) distributed debug operations that involve several MCUs part of a system.

5 Conclusion

In this paper, we have highlighted several challenges and problems that complicate the implementation of language-agnostic debuggers for modern IoT systems. One major challenge is the need for over-the-air debugging, which, combined with the hardware constraints of MCUs, poses significant challenges for tooling support. Moreover, the need for additional hardware and adaptors has raised more problems that complicate the implementation of scalable over-the-air debugging support for large IoT systems. We have also outlined key requirements for any language-agnostic debugging solution. Specifically, the ability to identify both language-dependent and independent debugging operations is crucial for ensuring that debugging operations are portable across applications written in different languages. Lastly, we have discussed and motivated an envisioned solution for language-agnostic debugging based on WebAssembly.

By building on previous work [20, 31], we are taking steps towards implementing our envisioned WebAssembly language-agnostic debugger. These early efforts made debugging over-the-air possible but are only applicable to applications written in Textual WebAssembly (i.e., a human-readable, bytecode-level programming language syntactically close to WebAssembly). More recently, we have extended these debuggers to support the debugging of Rust and AssemblyScript applications. We are also exploring an approach that could generalise the debugger to any language that compiles to WebAssembly, though we currently lack sufficient evaluation to confirm this claim. Additionally, we are designing a debugging API that aims to enable distributed debugging operations, crucial for modern IoT systems.

Acknowledgments

We are very grateful for the financial support provided by the Fonds Wetenschappelijk Onderzoek - Vlaanderen - that helped finance PhD-SB fellow Carlos Rojas Castillo Project number: 1SHEU24N.

References

[1] Bytecode Alliance. 2024. CLI and Rust libraries for low-level manipulation of WebAssembly modules. https://github.com/bytecodealliance/wasm-tools. Accessed: June 04, 2024.
[2] Bytecode Alliance. 2024. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. Accessed: June 04, 2024.
[3] appcypher. 2024. Awesome WebAssembly Languages. https://github.com/appcypher/awesome-wasm-langs. Accessed: June 04, 2024.
[4] AtomVM. 2024. AtomVM. https://www.atomvm.net/. Accessed: June 04, 2024.
[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* 54, 15 (2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010
[6] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2012. Quantify the time and cost saved using reversible debuggers. *Cambridge Judge Business School, Tech. Rep* (2012).
[7] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. 2008. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (Raleigh, NC, USA) *(SenSys '08)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/1460412.1460422
[8] Sauptik Dhar, Junyao Guo, Jiayi (Jason) Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. 2021. A Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective. *ACM Trans. Internet Things* 2, 3, Article 15 (jul 2021), 49 pages. https://doi.org/10.1145/3450494
[9] DWARF. 2024. DWARF Debugging Format. https://dwarfstd.org/. Accessed: June 04, 2024.
[10] Espessif. 2024. Tips and Quirks. https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-guides/jtag-debugging/tips-and-quirks.html. Accessed: June

04, 2024.

[11] ESPRESSIF. 2024. ESPRESSIF. https://www.espressif.com/. Accessed: June 04, 2024.

[12] Espruino. 2024. Espruino. https://www.espruino.com/. Accessed: June 04, 2024.

[13] Kaiming Fang and Guanhua Yan. 2020. IoTReplay: Troubleshooting COTS IoT Devices with Record and Replay. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 193–205. https://doi.org/10.1109/SEC50012.2020.00033

[14] Damien George. 2024. MicroPython. https://micropython.org/. Accessed: June 04, 2024.

[15] GNU. 2024. The GNU Project Debugger. https://www.gnu.org/software/gdb/. Accessed: June 04, 2024.

[16] IEEE 1149.1 Working Group. 2024. Official IEEE Std. 1149.1 Standard Working Group. https://grouper.ieee.org/groups/1149/1/. Accessed: June 04, 2024.

[17] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes - MPLR 2019*. ACM Press, 27–36. https://doi.org/10.1145/3357390.3361029

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[19] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. 2021. Understanding and improving model-driven IoT systems through accompanying digital twins. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Chicago, IL, USA) *(GPCE 2021)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3486609.3487210

[20] Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. 2022. Event-Based Out-of-Place Debugging. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/3546918.3546920

[21] Chao Li, Rui Chen, Boxiang Wang, Zhixuan Wang, Tingting Yu, Yunsong Jiang, Bin Gu, and Mengfei Yang. 2023. An Empirical Study on Concurrency Bugs in Interrupt-Driven Embedded Software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1345–1356. https://doi.org/10.1145/3597926.3598140

[22] LKML. 2024. Linux Kernel Mailing List Archive. https://lkml.org/lkml/2012/2/10/356. Accessed: June 04, 2024.

[23] LLDB. 2024. The LLDB Debugger. https://lldb.llvm.org/. Accessed: June 04, 2024.

[24] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

[25] Kais Mekki, Eddy Bajic, Frederic Chaxel, and Fernand Meyer. 2019. A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT Express* 5, 1 (2019), 1–7. https://doi.org/10.1016/j.icte.2017.12.005

[26] mtolmacs. 2024. Cargo WASM Sourcemap Utility. https://github.com/mtolmacs/wasm2map. Accessed: June 04, 2024.

[27] OpenOCD. 2024. Open On-Chip Debugger. https://openocd.org/. Accessed: June 04, 2024.

[28] Duktape organization. 2024. Duktape. https://duktape.org/. Accessed: June 04, 2024.

[29] Raspberry Pi. 2024. Raspberry Pi Foundation. https://www.raspberrypi.org/. Accessed: June 04, 2024.

[30] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. 2005. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems* (San Diego, California, USA) *(SenSys '05)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/1098918.1098946

[31] Carlos Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2023. Out-of-things debugging: A live debugging approach for Internet of Things. *The Art, Science, and Engineering of Programming* 7, 2, Article 5 (oct 2023), 33 pages. https://doi.org/10.22152/programming-journal.org/2023/7/5

[32] Rust. 2024. The Rust on ESP Book. https://docs.esp-rs.org/book/. Accessed: June 04, 2024.

[33] SourceMap. 2024. SourceMap. https://tc39.es/source-map/. Accessed: June 04, 2024.

[34] M5 Stack. 2024. M5 Stack. https://m5stack.com/. Accessed: June 04, 2024.

[35] Ryo Sugihara and Rajesh K. Gupta. 2008. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.* 4, 2, Article 8 (apr 2008), 29 pages. https://doi.org/10.1145/1340771.1340774

[36] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks* (Seattle, Washington) *(IPSN '15)*. Association for Computing Machinery, New York, NY, USA, 286–297. https://doi.org/10.1145/2737095.2737096

[37] TinyGo. 2024. A Go Compiler For Small Places. https://tinygo.org/. Accessed: June 04, 2024.

[38] Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. 2022. Understanding device integration bugs in smart home system. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 429–441. https://doi.org/10.1145/3533767.3534365

[39] wasm3. 2024. wasm3. https://github.com/wasm3/wasm3. Accessed: June 04, 2024.

[40] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems* (Sydney, Australia) *(SenSys '07)*. Association for Computing Machinery, New York, NY, USA, 189–203. https://doi.org/10.1145/1322263.1322282

Received 2024-06-21; accepted 2024-07-22