Chapter 1

Introduction

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

As surely as computers will continue to run software, so will software continue to have bugs and mistakes. That's not pessimism; it's reality. After all we live in an imperfect world. No formal method, nor static checker, or machine learning model can ever eliminate all mistakes—especially, for software which interacts with this imperfect, and unpredictable world. This is where debuggers—inevitably—come in.

1.1 The nature of programming mistakes

Programming mistakes come in many forms, and have many causes (McCauley et al., 2008). Some are simple syntax errors or off-by-one mistakes that static or model checkers can easily catch (García-Ferreira et al., 2014). Others have deep, and complex causes such as timing issues in concurrent systems (Li et al., 2023, Lu et al., 2008), memory corruption from hardware quirks (Bojanova and Eduardo Galhardo, 2021, Dessouky et al., 2018, Schroeder et al., 2009), non-deterministic edge cases triggered only under rare input conditions (Weiss et al., 2021), and so forth (Catolino et al., 2019). Crucially, many real-world bugs are not predictable or even known in advance (Mogul, 2006, Ubayashi et al., 2019). We often do not know what class of bug we are hunting until we start looking.

This is why automated verification (D'Silva et al., 2008, Rodriguez et al., 2019), however powerful, has limits. Formal proofs and model checking work only when the system and its requirements can be fully specified and the relevant properties articulated. But most software systems today are too large, too dependent on external environments, or too hastily evolving for perfect formalization. Many mistakes arise precisely where models stop: at the interfaces, in the messy real-world details, or in emergent behaviors no one thought to check. In these cases, the only path forward is empirical investigation. This almost exclusively involves inspecting program execution with debuggers, where developers trace the steps that led to the failure.

1.2 The history of debugging

Debugging has been central to programming since the earliest days of computer science. The term itself is often traced to *Grace Hopper's famous 'bug' story* from 1947, when a literal moth was found shorting a relay in the Harvard Mark II computer (Cohen, 1994). The term *debugging* came into circulation shortly after, and continues to be an integral part of programming today.

Actually, the term *bug* was already used to refer to a fault in a machine in the late nineteenth century (Wills, 2022).

Debugging evolved alongside programming languages and systems, from the early days of assembly language and machine code to the high-level languages we use today.

The introduction of integrated development environments (IDEs) in later decades brought breakpoints, watchpoints, and step-through execution into the everyday workflow. More recently, advanced tools like time-travel debuggers, distributed tracing systems, and concurrency visualizers have emerged to address the complexity of modern, multi-threaded, and distributed systems. However, while many of these novel techniques are well understood in the research community, industry and popular IDEs often lag behind in their adoption.

Even around the 2010s, there was still no single definition of *a debugger*, and knowledge around their use was often informal and anecdotal (Layman et al., 2013). However, efforts continued on trying to arrive at a more systematic understanding of debugging, and its tools (Layman et al., 2013, McCauley et al., 2008).

But across all these stages, the core challenge remains unchanged: making the invisible state of a running program visible, so its failures can be understood and corrected.

1.3 What are debuggers?

As the history of debugging shows, the term debugger is fluid and can refer to a wide range of tools and techniques. Moreover, a wide variety of categories exists, often with overlapping features, and even different names for the same concepts.

The latest ISO vocabulary standard defines debugging as, *"to detect, locate, and correct faults in a computer program"* ("ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary," 2017). This is too broad to help us arrive at a precise definition of debuggers.

1.4 Why debuggers matter

Clearly not all errors, faults, or bugs can be found easily, let alone, automatically detected and fixed. Many bugs are unpredictable, non-deterministic, and only emerge under specific conditions. This is especially true for software running on embedded systems. Here, bugs can be caused not just by pure mistakes in the programming logic, but also by unexpected interactions with the hardware, specific timings, or unexpected behavior from the physical world. To track down the causes of such failures, we need direct access to the system's behavior—to stop execution, inspect memory, walk through the precise state transitions that led to failure.

This is what debuggers give us. They provide precise, and deterministic mechanisms for controlling and examining program execution, essential for diagnosing subtle bugs, concurrency issues, performance bottlenecks, and hardware-specific behavior.

While automatic tools such as static analyzers, model checkers, and type systems can catch many classes of errors, they are limited by what they are designed to check. They work when you know the kinds of mistakes you're guarding against. But when a system fails and you don't know why, and have no predefined property to verify, you need debuggers that let you observe the system directly.

Despite the rapid rise of large language models (LLMs) in software engineering, debuggers remain critical for program understanding, and finding mistakes. While LLMs can assist in bug detection or code generation, they operate as probabilistic tools without direct connection to runtime state, offering suggestions rather than guarantees.

Debuggers, by contrast, are deterministic and precise, providing direct access to program execution and memory state. Ongoing debugger research not only enhances these capabilities but also drives advances in program analysis, visualization, security, and education. In fact, since more and more code is generated probabilistically with LLMs, there is arguably an even greater need for deterministic and precise debugging tools to inspect the generated code.

1.5 The challenges of resource-constraints

1.6 The challenges of non-determinism

1.7 The promise of universal bytecode interpreters

Chapter 2

Foundations for Debugging Techniques

Beware of bugs in the above code;
I have only proved it correct, not tried it.
— Donald Knuth, *personal communication c. 1970*

A central concern of this dissertation is the design of debuggers, and what makes a good debugger. To understand and answer this question, there are currently few formal foundations to build upon. Any such foundation must answer the fundamental question of what constitutes correctness for debuggers. Over the course of writing this dissertation, several correctness criteria for debuggers emerged, the essence of which we distill in this chapter into a general definition of correctness for debuggers.

2.1 Semantics of debuggers

Before we can begin to reason about the correctness of debuggers, we need to establish their formal semantics. Unfortunately, defining the semantics of debuggers has always received less attention than formalizations for programming languages or compilers (da Silva, 1992). This lack of interest, has resulted in quite a sparse collection of existing semantics, which focus on very different aspects, and are defined in very different ways. To this day, there is no clear consensus on what constitutes correctness for debuggers, or even, which are the essential aspects for a tool to fall under the broad category of debuggers.

2.1.1 A brief history of formal debuggers

To our knowledge, the earliest attempt at formally defining a debugger-like system is by Bahlke and Snelting (1986). The paper presents the *Programming System Generator*, which is a programming tool that generates an interpreter from the denotational semantics of a programming language. It supports interactive evaluation with the ability to inspect or redefine code, which is somewhat debugger-like in spirit.

However, the earliest work we are aware of—that formally describes a tool we would today recognize as a debugger—is the PhD thesis by da Silva (1992). It defines debuggers as any tool that can dynamically give some information of the intermediate states of program evaluation, on the request of the user. A definition not unlike the one we use in this dissertation. The thesis presents a way of formalizing debuggers using structural operational semantics, but the formalism does not separate the language semantics from the debugger semantics.

In 1995, Bernstein and Stark (1995) improved on this approach by explicitly separating the language and debugger operations in their formalisation, allowing them to define the debugger semantics in terms of the language semantics. To our knowledge they are the first to use this approach.

Another early attempt used PowerEpsilon (Zhu and Wang, 1992, 1991) to describe the source mapping used in a debugger as a denotational semantics for a toy language that can compile to a toy instruction set (Zhu, 2001a). While an interesting formalization, it does not say anything about the debugging operations themselves or their correctness.

A more recent work by Li and Li (2012) focussed on automatic debuggers. Its formalization is based on a kernel of the C language, and defines operational semantics for tracing, and for backwards searching based on those traces. The work proofs that its trace and search operations terminate, but defines no general correctness criteria.

However, most works after 2000 have largely used the approach first presented by Bernstein and Stark (1995), such as a number of recent works (Ferrari and Tuosto, 2001, Holter et al., 2024, Lauwaerts et al., 2024, Torres Lopez et al., 2017) that inspired and informed this dissertation. While there are still large differences in the way debuggers are formalised in recent works, it is clear that defining their semantics in terms of the underlying language is now accepted as the canonical approach.

By defining the operational semantics of a debugger in terms of the underlying language, it becomes much easier to reason about the correctness of the debugger, since the correctness can be stated in terms of the underlying language. In hindsight, this may seem an obvious solution to the reader, but that speaks to the fact that this is by far the best and most intuitive approach to take. As we are highly interested in the possible correctness criteria of debuggers in this dissertation, we will use this approach throughout.

Simply using the underlying language in an operational semantics—as an approach—still leaves a lot of flexibility in how to define the semantics of the debugger. Therefore, we will present this thesis' approach in more detail in this chapter, as we simultaneously discuss our general correctness criteria.

2.1.2 Four debuggers, four semantics

Given the wide variety of debuggers, we will present our formal framework —for debugger semantics and their correctness—by discussing four different debuggers with each their own semantics. Importantly, the semantics follow the same general design—presenting the overall formal framework we use in this dissertation.

While heavy in formal aspects, this chapter serves as a—hopefully somewhat gentle—introduction into the formal foundations of this dissertation.

Looking slightly ahead, we will define the following four debuggers, which always build on top of the previous one:

Debugger	Description
$\lambda^{*}_{\mathbb{D}}$	A tiny remote debugger, presenting a smallest working example.
$\lambda^{\rightarrow}_{\mathbb{D}}$	A remote debugger with support for the most conventional debug operations.
$\lambda^{\leftarrow}_{\mathbb{D}}$	A reversible debugger, which can step backwards in time.
$\lambda^{↯}_{\mathbb{D}}$	An intercession debugger, which can change the program at runtime.

The four debuggers allow us to introduce different aspects of our formal framework step by step. The semantics in this chapter, are blueprints for the more complex semantics we discuss later in this dissertation. They also serve to illustrate the general correctness criteria we define for debuggers.

In order to present our formal framework, we need a simple yet illustrative language. Fortunately there is a straightforward choice, the *simply typed lambda calculus* ($\lambda^{\rightarrow}$), proposed by Church (1940). Most readers will be familiar with the simply typed lambda calculus, but for those who are not, we provide a brief introduction.

2.2 $\lambda^{\rightarrow}$ as the running example

The simply typed lambda calculus, is arguably the simplest, and most well-known formal system used to study computation and programming languages. For fullness, we provide the core rules for the simply typed lambda calculus without any base types in Figure 2-1. In the lambda calculus, functions are the central form of computation, and there are only two basic operations; function application, and function abstraction. Function application is used to apply a function to another, while abstraction binds free variables to the function. In the simply typed version, each expression

Syntax

t ::=		*(terms)*
	x	*variable*
	$\lambda x : T.t$	*abstraction*
	$t\ t$	*application*
v ::=		*(values)*
	$\lambda x : T.t$	*abstraction*
T ::=		*(types)*
	$T \to T$	*function type*
Γ ::=		*(contexts)*
	$\varnothing$	*empty context*
	$\Gamma, x : T$	*variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \textit{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \textit{(E-App2)}$$

$$(\lambda x : T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]\ t_{12} \quad \textit{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \textit{(T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \quad \textit{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \textit{(T-App)}$$

Figure 2-1. Pure simply typed lambda calculus $\lambda^{\to}$. The syntax, evaluation, and typing rules for the simply typed lambda calculus with no base types (Pierce, 2002).

The rules for $\lambda^{\to}$, in both Figure 2-1 and Chapter B, are taken from the definitive work, *Types and Programming Languages* from Benjamin C. Pierce.

is assigned a type, and functions are given types that describe the kinds of inputs they accept and outputs they produce.

2.3 $\lambda^{*}_{\mathbb{D}}$: A tiny remote debugger for $\lambda^{\to}$

We start by defining the syntax of a tiny remote debugger for $\lambda^{\to}$ with booleans and natural numbers, defined as peano numbers (Kennedy, 1974, Peano, 1891). The complete set of syntax, evaluation, and typing rules for booleans and natural numbers for $\lambda^{\to}$ can be found in Chapter B. Because the debuggers we discuss in this dissertation are each debuggers for distributed systems, and therefore remote debuggers of a kind, we start with a simple remote debugger. However, the easiest way to define such a debugger is to start from a local debugger, and simply add a messaging system on top of it.

The rules for our tiny remote debugger are shown in Figure 2-2—these rules define the operation of the debugger backend. Typically, a debugger will also have a frontend for users to interact with the debugger, but this is beyond the scope of the semantics. The rules therefore only model the interface between the backend and the frontend as a simple messaging system.

The evaluation rules in Figure 2-2 are split into two sets, the internal debugging steps ($\xrightarrow{c}$) with c the debugging operation, and the remote debugging steps ($\longrightarrow_{\mathbb{D}}$), which wraps the former steps. The rules specific to the remote

Syntax

d ::=	*(internal debugger)*
$t \mid [\![m]\!]$	
δ ::=	*(remote debugger)*
$[\![\mathrm{c}]\!] \mid d$	
m ::=	*(output)*
$\varnothing$	*nothing*
t	*term*
ack c	*acknowledgement*
c ::=	*(debug commands)*
$\varnothing$	*nothing*
step	*single step*
inspect	*inspection*

Evaluation $\boxed{\delta \longrightarrow_{\mathbb{D}} \delta'}$

$$\frac{t \longrightarrow t'}{t \mid [\![\varnothing]\!] \xrightarrow{\text{step}} t' \mid [\![\text{ack step}]\!]} \quad \textit{(E-Step)}$$

$$\frac{}{v \mid [\![\varnothing]\!] \xrightarrow{\text{step}} v \mid [\![\text{ack } \varnothing]\!]} \quad \textit{(E-Fallback)}$$

$$\frac{}{t \mid [\![\varnothing]\!] \xrightarrow{\text{inspect}} t \mid [\![t]\!]} \quad \textit{(E-Inspect)}$$

$$\frac{}{t \mid [\![m]\!] \xrightarrow{\varnothing} t \mid [\![\varnothing]\!]} \quad \textit{(E-Read)}$$

$$\frac{d \xrightarrow{\mathrm{c}} d'}{[\![\mathrm{c}]\!] \mid d \longrightarrow_{\mathbb{D}} [\![\varnothing]\!] \mid d'} \quad \textit{(E-remote)}$$

Figure 2-2. Remote debugger semantics $\lambda^{*}_{\mathbb{D}}$. The syntax and evaluation rules for a simple remote debugger ($\longrightarrow_{\mathbb{D}}$) for the simply typed lambda calculus $\lambda^{\rightarrow}$ with natural numbers and booleans, defined over the internal operations ($\xrightarrow{\mathrm{c}}$).

debugger are highlighted in the figure, without them, the remaining rules define a tiny local debugger.

2.3.1 The syntax rules of the $\lambda^{*}_{\mathbb{D}}$ debugger

The steps of the remote debugger $\longrightarrow_{\mathbb{D}}$ are defined over a configuration $[\![\mathrm{c}]\!] \mid t \mid [\![m]\!]$, where we have respectively, the state of the remote debugger, the current program state, and the state of the internal debugger.

Internal in this context refers to the place where the program is running.

The configuration of the internal debugger is split into two parts, (1) the current state of the program—in this case a $\lambda^{\rightarrow}$ term t—and (2) the output displayed by the debugger frontend, modeled as the message box $[\![m]\!]$. Messages boxes are our way of modeling both inter-process and intra-process communication. In the case of the internal debugger, the message box is used to model the inter-process communication between the debugger backend and the debugger frontend within the same debugger process. Therefore, we can think of $[\![m]\!]$ as a high-level abstraction of the debugger frontend.

The configuration of the remote debugger is similar, but the message box $[\![m]\!]$ now models the intra-process communication from backend to frontend, and a second message box $[\![\mathrm{c}]\!]$ models the intra-process communication from frontend to backend. This corresponds, respectively, to the output returned from the debugger, and the instructions send to it.

The debugger can return as output, either nothing, a term, or an acknowledgement of a debug command. The debug commands supported by the debugger are *step* and *inspect*—to take a simple step in the program, and to inspect the current state of the program. Sometimes we also need the internal steps ($\xrightarrow{\text{c}}$) to perform an internal step, which does not correspond to a debug command visible to the user. For such cases, we also provide a nothing command ($\varnothing$).

2.3.2 The evaluation rules of the $\lambda^*_{\mathbb{D}}$ debugger

The entire evaluation of the debugger ($\delta \longrightarrow_{\mathbb{D}} \delta'$) is captured by only four rules. The first three steps are internal steps, which describe the operation of the internal debugger.

E-Step When the current term t can reduce to t', than the debugger can take a step to t', and output an acknowledgement of the successful step.

E-Fallback This fallback rule allows the debugger to drop *step* messages in case there is no $t \longrightarrow t'$. For the $\lambda^{\rightarrow}$, this means that the term must be a value v. In this case, we output an acknowledgement of $\varnothing$, to indicate that the command was processed, but did not have any effects.

E-Inspect The inspect step outputs the current term t.

E-Read The previous two steps require the output to be empty, to clear the output we introduce the *E-Read* rule.

To lift these internal steps to describe the operation of a remote debugger, we only need to add one rule which takes the next debug command from the input message box, and takes the correct corresponding internal step.

E-Remote The remote debugger takes the next debug command c from the input message box, and performs the corresponding internal step ($\xrightarrow{\text{c}}$).

The evaluation of the remote debugger is informed by the commands that arrive in the input message box, a debug session can therefore be seen as a series of remote steps ($\delta \longrightarrow^*_{\mathbb{D}} \delta'$) that are the result of a sequence of debug commands, which we write as (c^*).

Now that we have the formal semantics for a remote debugger that can step through and inspect a $\lambda^{\rightarrow}$ program, we can define what correctness means for such a debugger.

2.3.3 Correctness criteria for the $\lambda^*_{\mathbb{D}}$ debugger

Since we define our debugger in terms of the underlying language, the most intuitive definition of correctness for a debugger is that it should not change the semantics of the program being debugged. An intuition shared by the earliest works on debugger correctness such as da Silva (1992). We develop the idea into two correctness criteria, *debugger soundness* and *debugger completeness*.

Debugger soundness demands that for any debug session that begins at the start of the program, there is a path in the underlying language semantics that leads to the same final program state. In the theorem, we use the shorthand notation t_δ to denote the current term of a debugging configuration δ.

Theorem 2-1. (Debugger soundness) Let δ_{start} be the initial configuration of the debugger for some program t. Then:

$$\forall\, \delta \,.\, \left(\delta_{\text{start}} \longrightarrow^*_{\mathbb{D}} \delta\right) \Longrightarrow \left(t \longrightarrow^* t_\delta\right)$$

Proof. The proof proceeds by induction on the number of steps taken in the debugger. Since *E-Step* is the only rule that changes the term t in the debugger configuration, and *E-Step* uses the internal step ($\longrightarrow$); there is necessarily a path $t \longrightarrow^* t_\delta$ in the underlying language semantics. □

Debugger completeness is the dual of soundness, but in the opposite direction. Completeness demands that any path in the underlying semantics can be observed in the debugger.

Theorem 2-2. (Debugger completeness) Let t be a $\lambda^{\rightarrow}$ program, and δ_{start} the start configuration of a debug session for this program. Then:

$$\forall\, t' \,.\, \left(t \longrightarrow^* t'\right) \Longrightarrow \exists\, \delta \,.\, (\delta = [\![\mathrm{c}]\!] \mid t' \mid [\![m]\!]) \wedge \left(\delta_{\text{start}} \longrightarrow^*_{\mathbb{D}} \delta\right)$$

Proof. Given any path $t \longrightarrow^* t'$ in $\lambda^{\rightarrow}$, we can construct a sequence of debug commands c^* to be the exact number of *step* commands corresponding to the path in $\lambda^{\rightarrow}$. Then the debug session starting in δ_{start} with the commands c^* will take the exact same path by construction (see rule *E-Remote* and *E-Step*), resulting in a configuration ($[\![\varnothing]\!] \mid t' \mid [\![\varnothing]\!]$). □

Debugger soundness and completeness together ensure that the debugger does not deviate from the semantics of the program being debugged, and that the debugger and the normal execution observe the same program behaviour. This is the most essential property for any type of debugger.

New syntactic forms

d ::=		*(internal debugger)*
	$t \mid n, e, b, [\![m]\!]$	
e ::=		*(execution state)*
	paused	*paused state*
	play	*unpaused state*
b ::=		*(breakpoints)*
	$\varnothing$	*empty*
	n, b	*list of numerics*

m ::=		*(output)*
	…	
	hit n	*breakpoint hit*
c ::=		*(debug commands)*
	…	
	play	*unpause*
	pause	*pause*
	$\text{bp}^{+}\ n$	*add breakpoint*
	$\text{bp}^{-}\ n$	*remove breakpoint*

Numericals from $\lambda^{\rightarrow}$

n ::=		*(numeric values)*
	0	*constant zero*
	succ n	*succ*

Figure 2-3. Syntax rules of the conventional live debugger $\lambda^{\rightarrow}_{\mathbb{D}}$. The syntax rules for *pause*, *play*, and *breakpoints* for the $\lambda^{*}_{\mathbb{D}}$ debugger semantics. Changes to existing rules are highlighted.

Both theorems are trivial to prove for our tiny remote debugger $\lambda^{*}_{\mathbb{D}}$, however, this by no means implies that they are trivial to prove for every debugger, or that they have no value. To illustrate the usefulness of the correctness criteria, we will discuss them for a few interesting debuggers built on our tiny remote semantic.

2.4 $\lambda^{\rightarrow}_{\mathbb{D}}$: A conventional debugger for $\lambda^{\rightarrow}$

The tiny remote debugger $\lambda^{*}_{\mathbb{D}}$ is perhaps to simple to be really considered —what we conventionally call—a live remote debugger. The most obvious missing pieces are *pause* and *play* commands, and support for *breakpoints*. The semantics so far consider the program to be paused at all times, and the debugger only moves forward when the user issues a *step* command.

To support the pausing of the program's evaluation, as well as breakpoints, we extend the syntax of the tiny remote debugger with the rules shown in Figure 2-3. The internal debugger configuration is extended with a *program counter*, a plain numerical value as defined by the syntax of $\lambda^{\rightarrow}$, an *execution state* that can either be *paused* or *play*, and a set of *breakpoints*.

Using these three new fields (n, e, b), we can define the new evaluation rules for the conventional debugger. Figure 2-4 shows the new evaluation rules

Internal Evaluation $\boxed{d \xrightarrow{\text{c}} d'}$

$$\frac{e = \text{paused} \qquad t \longrightarrow t'}{t \mid n, e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{step}} t' \mid \text{succ}\ n, e, b, \llbracket \text{ack step} \rrbracket} \quad \textit{(E-Step)}$$

$$\frac{e \neq \text{paused}}{t \mid n, e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{step}} t \mid n, e, b, \llbracket \text{ack } \varnothing \rrbracket} \quad \textit{(E-Fallback2)}$$

$$\frac{}{t \mid n, e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{pause}} t \mid n, \text{paused}, b, \llbracket \varnothing \rrbracket} \quad \textit{(E-Pause)}$$

$$\frac{}{t \mid n, e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{play}} t \mid n, \text{play}, b, \llbracket \varnothing \rrbracket} \quad \textit{(E-Play)}$$

$$\frac{b' = n, b}{t \mid n', e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{bp}^{+}\ n} t \mid n', e, b', \llbracket \varnothing \rrbracket} \quad \textit{(E-BreakpointAdd)}$$

$$\frac{b' = b \setminus n}{t \mid n', e, b, \llbracket \varnothing \rrbracket \xrightarrow{\text{bp}^{-}\ n} t \mid n', e, b', \llbracket \varnothing \rrbracket} \quad \textit{(E-BreakpointRemove)}$$

Global Evaluation $\boxed{\delta \longrightarrow_{\mathbb{D}} \delta'}$

$$\frac{e = \text{play} \qquad t \longrightarrow t' \qquad n \notin b}{\llbracket \varnothing \rrbracket \mid t \mid n, e, b, \llbracket \varnothing \rrbracket \longrightarrow_{\mathbb{D}} \llbracket \varnothing \rrbracket \mid t' \mid \text{succ}\ n, e, b, \llbracket \varnothing \rrbracket} \quad \textit{(E-Run)}$$

$$\frac{n \in b}{\llbracket c \rrbracket \mid t \mid n, \text{play}, b, \llbracket \varnothing \rrbracket \longrightarrow_{\mathbb{D}} \llbracket c \rrbracket \mid t \mid n, \text{paused}, b, \llbracket \text{hit } n \rrbracket} \quad \textit{(E-BreakpointHit)}$$

Figure 2-4. Evaluation of conventional live debugger operations for $\lambda^{\to}_{\mathbb{D}}$. The evaluation rules for *pause*, *play*, and *breakpoints* for the $\lambda^{*}_{\mathbb{D}}$ debugger semantics. Changes to existing rules are highlighted.

added to or replacing the existing rules. The full set of rules for the conventional debugger are shown in Chapter C.

Now, we can easily let the debugger stop at any point in the reduction of the $\lambda^{\to}$ program by adding a rule for normal unpaused execution (*E-Run*) and by adding two rules to change the execution state to either *paused* or *play*. For breakpoint support we need to keep track of a program counter, which for the $\lambda^{\to}$ can simply be a numerical value that counts the number of reductions. To increase the counter correctly, we only need to change the *E-Step* and *E-Run* rules to increment the counter by one for every reduction in the $\lambda^{\to}$. Lastly, we need to add two new rules to add and remove breakpoints from the set of breakpoints in the debugger configuration, and an extra fallback rule to handle the case where the debugger is not paused, but a step command is received.

All other rules from the tiny remote debugger remain unchanged, apart from the additional fields in the configuration. The exact values of these fields are immaterial for those remaining rules.

E-Step The *E-Step* rule now only applies when the debugger is in the paused state.

E-Fallback2 We add a second fallback rule, for when a step command is received, but the debugger is not paused. The execution state is irrelevant in the other fallback rule.

E-Pause The *E-Pause* rule changes the execution state to *paused*.

E-Play The *E-Play* rule changes the execution state to *play*.

E-BreakpointAdd The *E-BreakpointAdd* rule adds the breakpoint n from the (bp^{+} n) command to the set of breakpoints in the debugger configuration.

E-BreakpointRemove The *E-BreakpointRemove* rule removes the breakpoint n—specified by the (bp^{-} n) command—from the set of breakpoints in the debugger configuration.

E-Run When the execution state is *play*, and there are currently no commands in the message box of the remote debugger, nor is the current program counter n an element of the breakpoints set b, then the debugger will take a single step in the underlying language semantics $t \longrightarrow t'$. Through this rule the debugger will continue normal execution until it reaches a breakpoint, or the program is paused, or the program is cannot be reduced anymore.

E-BreakpointHit When the execution state is *play*, and the program counter n is part of the breakpoint set b, the debugger pauses the program by changing the execution state to *paused*. Finally, it outputs an alert of the breakpoint hit containing the current program counter.

2.4.1 Correctness criteria for the $\lambda_{\overrightarrow{\mathbb{D}}}$ debugger

We will apply the same *soundness* and *completeness* criteria to the conventional debugger as we did for the tiny remote debugger. We briefly sketch the proofs here, starting with soundness.

Proof. (Debugger soundness for the conventional debugger) The proof proceeds by induction on the number of steps taken in the debugger. The *E-Step* and *E-Run* rules are the only rule that changes the term t in the debugger configuration, and both use the internal step ($\longrightarrow$). This means that there is necessarily a path $t \longrightarrow^{*} t_{\delta}$. □

The proof for completeness is identical to the proof given for the tiny remote debugger, since we do not need to introduce any breakpoints in the debugging session. The rules *E-Remote* and *E-Step* can still faithfully re-execute the path $t \longrightarrow^{*} t'$. ,

2.5 $\lambda^{\leftrightarrows}_{\mathbb{D}}$: A reversible debugger for $\lambda^{\rightarrow}$

Another interesting extension to the tiny remote debugger, is to turn it into a reversible debugger (Engblom, 2012). We start from the conventional debugger semantics in Section 2.4, and add a *backwards step* command.

A common approach to implementing a reversible debugger is to periodically store snapshots of the program state, and reconstruct the execution from the last snapshot (Engblom, 2012, Klimushenkova and Dovgalyuk, 2017). Formalising this approach requires only few extensions to the semantics of the conventional debugger. We list the new syntax and evaluation rules for the reversible debugger in Figure 2-5.

We extend the syntax of the debugger with a list of snapshots, which are tuples of program counters and terms. The commands are extended with a *backwards step* command, which takes a single step backwards in the program. To handle this command we need five internal rules, specifically, the *E-BackwardStep0*, *E-BackwardStep2*, and *E-BackwardStep2* rules, along with two fallback rules.

E-BackwardStep0 The *E-BackwardStep0* rule applies when the program counter is not zero, but only the start snapshot is present in the snapshot list. In this case the program reduces n times starting from the initial configuration, to arrive exactly one reductions before the current term t.

E-BackwardStep1 The *E-BackwardStep1* rule applies reduces the program counter by one, and reduces the term t' from the last snapshot exactly $n - n'$ times, to the term t''.

E-BackwardStep2 The *E-BackwardStep2* rule applies when the program counter is exactly one higher than the program counter of the last snapshot. In this case, the debugger only restores the snapshot and removes it from the snapshot list.

New syntactic forms

d ::=		*(internal debugger)*
	$t \mid n, e, b, \boxed{s}, [\![m]\!]$	
c ::=		*(debug commands)*
	…	
	$\boxed{\text{step}^{\leftarrow}}$	$\boxed{\textit{backwards step}}$
s ::=		*(snapshots)*
	$(0, t)$	*start snapshot*
	$(n, t), s$	*list of snapshots*

Internal Evaluation $\boxed{d \xrightarrow{c} d'}$

$$\frac{s = (0, t') \qquad e = \text{paused} \qquad t' \longrightarrow^{n} t''}{t \mid \text{succ } n, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{step}^{\leftarrow}} t'' \mid n, e, b, s, [\![\text{ack step}^{\leftarrow}]\!]} \quad \textit{(E-BackwardStep0)}$$

$$\frac{n \neq n' \qquad s = ((n', t'), s') \qquad e = \text{paused} \qquad t' \longrightarrow^{n-n'} t''}{t \mid \text{succ } n, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{step}^{\leftarrow}} t'' \mid n, e, b, s, [\![\text{ack step}^{\leftarrow}]\!]} \quad \textit{(E-BackwardStep1)}$$

$$\frac{s = ((n, t'), s') \qquad e = \text{paused}}{t \mid \text{succ } n, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{step}^{\leftarrow}} t' \mid n, e, b, s', [\![\text{ack step}^{\leftarrow}]\!]} \quad \textit{(E-BackwardStep2)}$$

$$\frac{e \neq \text{paused}}{t \mid n, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{step}^{\leftarrow}} t \mid n, e, b, s, [\![\text{ack } \varnothing]\!]} \quad \textit{(E-BackwardFallback1)}$$

$$\frac{s = (0, t)}{t \mid 0, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{step}^{\leftarrow}} t \mid 0, e, b, s, [\![\text{ack } \varnothing]\!]} \quad \textit{(E-BackwardFallback2)}$$

Global Evaluation $\boxed{\delta \longrightarrow_{\mathbb{D}} \delta'}$

$$\frac{e = \text{play} \qquad t \longrightarrow t' \qquad s' = ((\text{succ } n, t'), s) \qquad n \notin b \qquad (\text{succ } n) \,\%\, \theta = 0}{[\![\varnothing]\!] \mid t \mid n, e, b, s, [\![\varnothing]\!] \longrightarrow_{\mathbb{D}} [\![\varnothing]\!] \mid t' \mid \text{succ } n, e, b, s', [\![\varnothing]\!]} \quad \boxed{\textit{(E-Run1)}}$$

$$\frac{e = \text{play} \qquad t \longrightarrow t' \qquad n \notin b \qquad (\text{succ } n) \,\%\, \theta \neq 0}{[\![\varnothing]\!] \mid t \mid n, e, b, s, [\![\varnothing]\!] \longrightarrow_{\mathbb{D}} [\![\varnothing]\!] \mid t' \mid \text{succ } n, e, b, s, [\![\varnothing]\!]} \quad \boxed{\textit{(E-Run2)}}$$

Figure 2-5. Syntax and evaluation rules of the reversible debugger $\lambda_{\mathbb{D}}^{\leftrightarrows}$. The semantics of $\lambda_{\mathbb{D}}^{\leftrightarrows}$ extend the conventional debugger semantics $\lambda_{\mathbb{D}}^{\rightarrow}$, shown in Figure 2-3 and Figure 2-4.

E-BackwardFallback1 The *E-BackwardFallback1* rule applies when the execution state is not paused. Analogous to the forward step, the debugger will not step back if the program is not paused, and simply send an empty acknowledgement to indicate that nothing has changed.

E-BackwardFallback2 The *E-BackwardFallback2* rule applies when the program counter is zero, in this case, the only sensible option is to also return an empty acknowledgement, since the program cannot step back any further.

Given these internal evaluation rules, we only need to specify in the global evaluation rules how and when snapshots are created. Several strategies can be used to determine when to create new snapshots, for simplicity we will let the debugger create a snapshot every few steps by replacing the *E-Run* rule by the following two rules.

E-Run1 We change the *E-Run* rule to add a new snapshot to the list s whenever the program counter is a multiple of θ, which we consider a static configuration of the debugger.

The value of θ could be changed through some meta-rules for the debugger.

E-Run2 In case the program counter is not a multiple of θ, the *E-Run2* rule is the same as the original *E-Run* rule.

To summarize the reversible semantics, when the reversible debugger is at a term t with program counter succ n, then to step back once, it will restore the last snapshot and take exactly $n - n'$ steps where n' is the program counter of the snapshot.

2.5.1 Correctness criteria for the $\lambda^{\leftarrows}_{\mathbb{D}}$ debugger

Again, we apply the same soundness and completeness criteria to the reversible debugger as we did for the two previous debuggers. The proofs however, are slightly more involved, since we need to reason about the snapshots. To make this easier, we will first proof two lemma about the snapshots that are helpful in the proofs of soundness and completeness.

Lemma 2-1. (Snapshot preservation) The semantics of a reconstructing reversible debugger is said to be *snapshot preserving* if the following holds:

$$\begin{gathered}\forall \delta \,.\, \delta = [\![\mathrm{c}]\!] \mid t \mid n, e, b, s, [\![m]\!] \wedge \delta_{\text{start}} \longrightarrow^{*}_{\mathbb{D}} \delta \\ \Longrightarrow \\ \forall (n', t') \in s \,.\, n' \leqslant n \wedge t_{\text{start}} \longrightarrow^{*} t'\end{gathered}$$

Proof. (Snapshot preservation for $\lambda^{\leftarrows}_{\mathbb{D}}$) The proof is straightforward by induction on steps taken in the debug session ($\longrightarrow^{*}_{\mathbb{D}}$). In the base case, this is always trivial to prove by construction. In the inductive case, each case is straightforward to prove given the induction hypothesis. □

New syntactic forms		
c ::=		*(commands)*
	...	
	subst $t_1 t_2$	*substitute*

Internal Evaluation $\boxed{d \xrightarrow{\text{c}} d'}$

$$\frac{\Gamma \vdash t_2 : T' \qquad \Gamma, t_1 : T' \vdash t : T}{t \mid n, e, b, s, [\![\varnothing]\!] \xrightarrow{\text{subst } t_1 t_2} [t_1 \mapsto t_2]\, t \mid n, e, b, s, [\![\text{ack subst } t_1 t_2]\!]} \quad \textit{(E-Subst)}$$

Figure 2-6. Intercession debugger semantics extending $\lambda^{*}_{\mathbb{D}}$.

Given Lemma 2-1, we know that any snapshot list produced by the reversible debugger observes the program order, and that there is never a snapshot that lies in the *future* of the current program state.

Proof. (Debugger soundness for $\lambda^{\leftrightarrows}_{\mathbb{D}}$) The proof proceeds by induction over the steps taken in the debug session. Except for the new backward stepping rules, the cases proceed analogous to the proof for $\lambda^{\rightarrow}_{\mathbb{D}}$. Given the induction hypothesis and Lemma 2-1, the backward rules *E-BackwardStep0*, *E-BackwardStep1*, and *E-BackwardStep2* are straightforward to prove. □

2.6 $\lambda^{\lightning}_{\mathbb{D}}$: An intercession debugger for $\lambda^{\rightarrow}$

Our debuggers so far have only observed the execution of a program, without interceding in it. Even our reversible debugger, does not intercede in the control flow of the program, it only replays a previously observed execution. Yet, it is quite common for debuggers to support changing the value of variables (Stallman et al., 1988), or influence the control flow of the program ("Alter the Program's Execution Flow," 2024, Lauwaerts et al., 2022, Stallman et al., 1988).

Intercession debuggers are an interesting case to study in terms of our correctness criteria. Since, we expect the debugger to observe the same semantics as the program, we need to be careful when changing the program state. It is very easy when changing even just a simple variable to break debugger correctness. Luckily, we can illustrate this in the $\lambda^{\rightarrow}$ by allowing the debugger to substitute terms at runtime.

The substitution debug command is similar to substitutions through let bindings in $\lambda^{\rightarrow}$ (Pierce, 2002).

Figure 2-6 shows our intercession debugger semantics, as again an extension on the previous debugger semantics—shown in Figure 2-5. We add a new debug command subst $t_1 t_2$ to the debugger, which allows the user to substitute the current term t_1 with a new term t_2 of the same type.

2.6.1 Intercession breaks straightforward correctness

Unfortunately, the intercession debugger is not sound by the definition of the previous debuggers. The previous soundness criteria are defined in terms of the entire debugging sessions, starting from the beginning of the program. This criterion can never be satisfied for all debugging session of an intercession debugger that can arbitrarily update the program code. We can illustrate this by the following example (Example 2-1), where we use the *substitution* command to change the program at runtime.

Example 2-1. The following shows a sequence of steps in the intercession debugger. Intercession commands are shown in bold.

$$\begin{array}{rc}
 & (\lambda x : \text{Nat} \,.\, \text{isZero}\ x)\ (\lambda y : \text{Nat} \,.\, \text{succ}\ y)\ 0 : \text{Bool} \\
\text{E-AppAbs} & \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
 & (\lambda x : \text{Nat} \,.\, \text{isZero}\ x)\ ([y \mapsto 0]\ (\text{succ}\ y)) : \text{Bool} \\
\textbf{E-Subst} & \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
 & [\mathbf{succ\ 0} \mapsto \mathbf{0}]\ (\boldsymbol{\lambda x} : \mathbf{Nat} \,.\, \mathbf{isZero}\ \boldsymbol{x})\ \ \mathbf{succ\ 0} : \mathbf{Bool} \\
\text{E-AppAbs} & \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
 & [x \mapsto 0]\ \text{isZero}\ x : \text{Bool} \\
\text{E-isZero} & \overline{\qquad\qquad\qquad\qquad} \\
 & \text{true} : \text{Bool}
\end{array}$$

In Example 2-1, the debugger changes all occurrences of *succ 0* in the program to simply *0* in the middle of the debugging session. Through this intervention, the program results in true, while the original code can clearly only be false. To our correctness criteria, this means we designed an incorrect debugger. However, there are many reasons for designing a debugger that can update the program during a debugging session, allowing developers to patch code as they debug it. Moreover, there is nothing in the function of the debugger that would lead us to believe—on the face of it—that the debugger is incorrect. After all, the new program is still well typed, and the debugger observes the correct behaviour of the updated program. Therefore the problem is not that our debugger is incorrect, but that the correctness criteria are too strict.

The solution here is rather intuitive—we consider the point where a program is updated by the debugger, as the start of a new debugger session. We explore this idea in the following section, where we redefine debugger soundness and completeness for intercession debuggers.

2.6.2 Updating the correctness criteria for intercession debuggers

Informally, the correctness of debuggers depends on their faithful observation of a program's behaviour. Intercession debuggers are a common type of debugger that shows this criteria is far from trivial. There are two major ways in which debuggers typically intercede with a program's execution, and correspondingly, two general principles those intercessions must follow.

Firstly, intercession debuggers that change the behaviour of a program—often my changing control flow or throwing exceptions ("Alter the Program's Execution Flow," 2024)—in order to be correct, may only introduce behaviour that could be observed in the underlying semantics. Secondly, as a general rule for intercession debuggers that change the program itself, the debugger must faithfully observe the new program from the moment the code was updated, and the updated program must remain well-typed. Our previous correctness criteria already cover the former rule, but the criteria are too strict for the latter class of intercession debuggers. We will adapt the soundness and completeness theorems to fit our second principle for debuggers that can change a program.

Until now, the debugger soundness theorem mirrored the *progress* theorem for programming languages (Pierce, 2002), however, since we now introduce program updates, we also need to think about *preservation* (Pierce, 2002). For the debugger, this means that any changes to the program code must keep the program well-typed.

Lemma 2-2. (Debugger preservation) A debugger semantic is said to be *preserving* if the following holds:

$$\forall \delta, \delta'. \delta \longrightarrow_{\mathbb{D}} \delta' \wedge t \in \delta \text{ is well-typed} \implies t' \in \delta' \text{ is well-typed}$$

Proof. (Preservation for $\lambda^{\lightning}_{\mathbb{D}}$) The proof proceeds by case analysis on the rules of the debugger. Only the $\xrightarrow{\text{subst}\, t_1 t_2}$ rule is interesting, since it is the only rule that changes the program code. However, since the rule replaces t_1 in the well-typed program t with a term of the same type t_2, the case is also trivially true. □

As reader may have noticed, this lemma is very general and can be applied the previously defined debugger semantics as well—and probably most other semantics for that matter. However, since the previous semantics in this chapter do not change the program code, they are trivially preserving. That said, for other intercession debuggers the preservation property is a crucial criterion for correctness.

Now we define debugger soundness, as the preservation of the program's well-typedness, and the existence of a path in the underlying language semantics starting from an arbitrary configuration δ—rather than the starting configuration.

Theorem 2-3. (Debugger soundness) A debugger semantic is said to be *sound* if it is *preserving* and the following holds:

$$\forall \delta, \delta'. \delta \longrightarrow^{*}_{\mathbb{D}} \delta' \wedge \text{subst } t_1 t_2 \notin (\longrightarrow^{*}_{\mathbb{D}}) \wedge t \text{ is well-typed} \implies t \longrightarrow^{*} t',$$

where $t \in \delta$ and $t' \in \delta'$ and $\delta_{\text{start}} \longrightarrow^{*}_{\mathbb{D}} \delta'$.

Proof. By Lemma 2-2, we know that $\lambda^{\lightning}_{\mathbb{D}}$ is *preserving*, so we only need to prove the second part of the theorem. The proof proceeds by induction on the steps in the debug session ($\delta \longrightarrow^{*}_{\mathbb{D}} \delta'$). Since we know that there are no substitution commands in the debug session, we can ignore the *E-Subst* rule, and the proof is analogous to the previous proofs. Only in the base case, do we not have δ_{start} but an arbitrary configuration δ. Since it is reachable from the start configuration and it's term is well typed, this makes little difference to the proof. □

Unlike soundness, debugger completeness is not broken because of intercession. After all, there is no reason that any intercession commands should take place during the debugging session we construct in the proof. Therefore—analogous to the previous extensions to the semantics—the addition of the *E-Subst* rule makes no difference, and the same proof for completeness holds.

2.7 Discussion: general debugger correctness

Given the wide variety of debuggers and the vagueness around what constitutes as a debugger, it is not possible to formally define a general correctness criterion that is the same for all types of debuggers. Therefore, it should not surprise anyone that the correctness criteria presented in this chapter depend on, and are different for, each of the debugger semantics. Especially, the criteria for the intercession debugger depend in a crucial way on the type of intercession the debugger supports.

However, the *soundness* and *completeness* criteria presented in this chapter do present the same general principle, which is that the debugger should observe the same semantics as the program being debugged. In the case of the intercession debuggers these criteria need to be adapted on a case by case basis, depending on the type of intercessions supported, but their general principles still hold.

The extensive discussion of the different debugger semantics for the $\lambda^{\rightarrow}$ in this chapter serve to show the general applicability of debugger soundness and completeness, and support our claim that these are the most essential correctness properties for any type of debugger. The same criteria will be used throughout this dissertation, as we explore how to develop sound out-of-place and multiverse debugging techniques for constrained environments. These debuggers bridge a wide spectrum of debugger types, and intercede in the program's execution in intricate ways. They present semantics that are much more complex than the simple semantics we presented in this chapter. This will illustrate further that the correctness criteria presented in this chapter are indeed useful properties for any type of debugger.

Furthermore, the spirit of the debugger semantics in this chapter closely aligns to the design of the debuggers we will present. For instance, our multiverse debugger presented in Chapter 5 contains similar semantics to our reversible debugger for exploring the possible execution paths of non-deterministic programs. The rest of this dissertation will also mirror the structure of this chapter, by first presenting a remote debugger for WebAssembly on microcontrollers, and then extending it with more advanced features in the following chapters.

Chapter 3

A Remote Debugger for WebAssembly

Those who abjure debugging can only do so by others debugging on their behalf.
— *adapted from* George Orwell

Developing and investigating novel debugging techniques for microcontrollers within our new formal framework, requires an easy way to instrument the program execution, and ideally prototype new debuggers quickly. The best way to achieve this is unarguably, to use a virtual machine that can run on the microcontrollers. Luckily, earlier work at Ghent University, developed just such a virtual machine, called WARDuino—which was the first-ever WebAssembly virtual machine for microcontrollers (Gurdeep Singh and Scholliers, 2019).

However, the original work was limited to a proof of concept, and many of the promises of the new WebAssembly-based approach to programming microcontrollers were not fully realised—such as, programming in high-level languages, highly portable code, the ability to easily handle asynchronous events, and by extension support for asynchronous I/O actions. In this chapter, we present a more complete version of WARDuino, developed as part of this dissertation. We will discuss the full range of features and benefits of the new approach to programming microcontrollers proposed by WARDuino.

Three features of WARDuino, and in particular their formalisation, will be crucial for developing our advanced out-of-place and multiverse debugging techniques, in the later chapters of this dissertation. These are the (1) the *remote debugger*, (2) atomic *actions* for I/O operations, and the (3) *asynchronous event-driven callback system* in the virtual machine.

However, we will start at the beginning. Why is debugging seen as such a frustrating task in the embedded world? How can virtual machines help with this? And what are the broader challenges in embedded development that a WebAssembly-based virtual machine for microcontrollers can help overcome?

3.1 Challenges of Programming Microcontrollers

Recent advances in microcontroller technology have enabled everyday objects (things) to be connected through the internet. Smart lamps, smart scales, smart ovens and refrigerators have all become commodity devices which are connected through the internet, making up the Internet of Things (IoT). This is largely thanks to microcontrollers—small and energy efficient computers—becoming very cheap. However, the drawbacks of microcontrollers are their limited processing power and memory size. Furthermore, microcontrollers typically do not run a full-fledged operating system (VanSickle, 2001), but instead run statically compiled firmware, or a tiny real-time operating system (RTOS) (De Sio et al., 2023, Hambarde et al., 2014, Tan and Anh, 2009) specialized for microcontrollers. Due to these differences and the resource constraints of the underlying hardware, developing software for microcontrollers differs significantly from conventional computer programming, where these severe constraints do not exist to the same degree. The WARDuino virtual machine seeks to close this gap, and focus on the following six major challenges unique to IoT development.

Low-level coding. First, embedded software are usually written in low-level programming languages, such as C (Aspencore, 2023, Kernighan and Ritchie, 1989). Although C is very efficient, developing programs in C is error-prone and time-intensive. Crucially, C requires developers to manually manage memory allocations which has been shown to be notoriously difficult for complex programs (English et al., 2019, van der Veen et al., 2012).

Portability. Second, many of the functionalities of a microcontroller are memory mapped, these mappings are highly specific for each microcontroller and can differ even between devices of the same microcontroller family. Completely different microcontrollers vary even more in the way they initialize and control peripherals. Therefore, porting programs from one platform to another can be difficult and time-consuming.

Slow development cycle. Third, uploading programs to a microcontroller is a slow and tedious process. For every change in the program, however small, the entire program must be recompiled and flashed (uploaded) to the device. This slows down the development cycle as developers need to wait for this process to finish before they can test their programs.

Debuggability. Fourth, debugging facilities are often not available for microcontrollers without (expensive) hardware debuggers. Even then, the debuggers usually only work for the C language. The lack of debugging facilities makes that developers cannot easily inspect the internal state of a microcontroller. They can only observe its external behavior, for example, that an LED that should be blinking, instead remains off. When the device is not behaving as expected, it is difficult to find the root cause of the problem. Many developers resort to printing values to the serial bus to figure out what the device is doing when something goes wrong (Makhshari and Mesbah, 2021), but this is very slow and inconvenient.

Hardware limitations. Fifth, the embedded devices powering IoT applications have severe hardware limitations compared to conventional computer systems. The most important and universal constraints for these devices, are their limited processing power and memory size.

Bare-metal execution environments. Finally, due to their heavy constraints, microcontrollers rarely run full-fledged operating systems. Instead, software is run on top of a tiny RTOS, or simply as statically compiled firmware.

3.2 Programming Microcontrollers with High-level Languages

Many of the difficulties in programming microcontrollers disappear when using higher-level languages. The abstractions in these languages can prevent whole classes of bugs and can ease development. Specifically, high-level languages relieve the programmer from manual memory management, and can provide stronger type systems to further avoid mistakes. They also make it easier to support advanced features such as over-the-air updates—where software is updated without flashing via a physical connection—and remote debugging, where a device is debugged through instructions sent from another remote device.

Some high-level languages have been ported to embedded devices using small custom virtual machines (nanoFramework Contributors, 2021, Williams, 2014, Zerynth s.r.l., 2021). Unfortunately, these virtual machines only support a subset of the language's features and only work on a specific range of hardware platforms. A popular example is MicroPython (George, 2021), which is a subset of Python for microcontrollers. For performance reasons access to the peripherals is often baked into high-level programming language. In MicroPython support for displays and sensors is baked into the language and implemented directly in C. As such, if a specific peripheral device is not supported, the language is of limited use. Another issue, is the lack of debuggers for these languages. This is also the case for MicroPython, which has no official debugger, and third party alternatives are very limited. The Mu debugger for example, supports only classic breakpoints and step instructions, and works exclusively on Raspberry Pi devices, which are much more powerful than embedded devices under consideration in this chapter. Finally, many high-level languages do not directly support embedded devices at all, as we will discuss further in Section 3.11.

3.3 WARDuino: WebAssembly for Microcontrollers

In this work we take a different approach aimed at enabling multiple high-level languages on microcontrollers while mitigating their downsides. To accomplish this goal, we created WARDuino (Gurdeep Singh and Scholliers, 2019), a virtual machine (VM) designed to run WebAssembly (Haas et al., 2017) on microcontrollers. Since WebAssembly is a universal compile target, it can enable programs written in a wide variety of languages to run on low-end embedded devices. This is an important design choice to improve the *portability* of our solution. The design of WebAssembly further focuses on a compact representation, since the byte code is intended to be streamable and efficient (Haas et al., 2017). This compactness is especially important when executing programs within the *hardware limitations* of the embedded devices. Additionally, WebAssembly can achieve performance speeds close to native code (Haas et al., 2017, Jangda et al., 2019), potentially outperforming other interpreters for high-level languages on microcontrollers.

WARDuino supports the first release of WebAssembly (MVP).

The WARDuino virtual machine was first presented in 2019 by Gurdeep Singh and Scholliers (2019), and addressed *the slow development cycle* and the challenging *debuggability* through the initial implementation of a remote debugger with over-the-air reprogramming capabilities. The paper presented these two features as extensions to the operational semantics of WebAssembly, in order to show their interaction and compatibility with the WebAssembly standard, and to ease re-implementation in other virtual machines. Additionally, WARDuino provided support for a limited set of hardware features through WebAssembly functions embedded in the virtual machine. It is necessary to embed this support in the virtual machine, since the *bare-metal execution environments* of embedded devices provide no conventional interfaces to the hardware. Simultaneously, the primitives should be exposed at the level of WebAssembly in order to achieve the highest *portability* possible. However, the paper left some important problems as future work.

First, WebAssembly does not natively support asynchronous code, but many standard M2M protocols for IoT applications such as MQTT (Banks and Gupta, 2014) rely on asynchronous events. Similarly, IoT applications often rely on asynchronous handling of hardware interrupts. Due to this limitation in WebAssembly, WARDuino lacked any support for either hardware primitives that rely on asynchronicity, or M2M protocols. In this chapter, we extend the WebAssembly operational semantics with support for event-driven callback handling. Through this system, callback functions can subscribe to asynchronous events at the WebAssembly level. The functions will be executed whenever such an event occurs. While other proposals for asynchronous code in WebAssembly are being developed (WebAssembly Community Group, 2022), these proposals are still in early stages, and often focus heavily on browser applications, making them unsuitable for resource-constrained microcontrollers.

Second, since hardware support is exposed at the WebAssembly level,pa-perthere is a language barrier that has to be bridged for every higher-level language. The original version of WARDuino did not address this issue, and in practice lacked any real support for high-level languages. In this chapter, we show that WARDuino can practically solve the *low-level coding* challenge as promised in the original paper (Gurdeep Singh et al., 2019). We illustrate how WARDuino can support high-level languages through language symbiosis, by showing different levels of language integration for AssemblyScript, a TypeScript-like language. These examples serve as a general recipe for implementing other language libraries.

Third, the formal rules for the debugger semantics were not used to prove any interesting properties or guarantees for the debugger. In this chapter, we improve the semantics for the debugging and over-the-air updates, and provide a proof for observational equivalence between the debugger semantics, and the underlying WebAssembly semantics. This equivalence means that the executions observed by the debugger semantics are precisely the same as those observed by the underlying language semantics.

Fourth, the virtual machine was never used to implement a real-world IoT application, and its evaluation was limited to a comparison of execution speed with only one alternative approach. In this chapter, the evaluation of WARDuino has been expanded to include a comparison to another WebAssembly runtime that can run on microcontrollers We also present real-world IoT application written in AssemblyScript and developed using WARDuino.

To further illustrate how WARDuino can provide an improved development experience, closer to conventional programming, we present how WebAssembly enables fast prototyping of emulators, and the improved tool support with the visual debugger plugin for WARDuino in the VS Code IDE. This plugin is an important contribution towards the increased *debuggability* of IoT software in WARDuino. The chapter also includes additional code examples that explain how hardware peripherals can be accessed from WebAssembly, as well as a notably improved and expanded presentation of the WARDuino virtual machine architecture.

In summary, our novel contributions compared to the initial paper (Gurdeep Singh et al., 2019) are:

The latest version of the VM is freely available under the Mozilla Public License 2.0

- A detailed and expanded presentation of the *improved WARDuino VM*: A WebAssembly virtual machine for embedded devices. (Section 3.5)
- *Support for IoT primitives* (asynchronous hardware peripherals and common M2M protocols) at the WebAssembly level. (Section 3.7 and Chapter F)
- A general recipe for supporting WebAssembly-level primitives in high-level languages in the form of *language symbiosis* implemented for the AssemblyScript language, presented through multiple code examples. (Section 3.6)
- The first formally described *event-driven callback system* and implementation for handling asynchronous code in WebAssembly. (Section 3.8)
- The first proof of *observational equivalence* between a debugger semantics and the WebAssembly operational semantics. (Section 3.8.3)
- An improved development experience thanks to better tool support through a *visual debugging environment* in VS Code, which currently supports debugging of WebAssembly and AssemblyScript code—and the possibility for fast prototyping of emulators thanks to WebAssembly. (Section 3.9)
- A smart light application written in AssemblyScript showcasing the new IoT primitives in WARDuino, and demonstrating that the callback system can handle both interrupts from the embedded device itself and from the network via asynchronous communication protocols such as MQTT. (Section 3.10.1)
- An expanded comparison of the execution speed of the virtual machine with another WebAssembly runtime that can run on low-end embedded devices. (Section 3.10.2)

The rest of the chapter is organized as follows. First, we show an example program and illustrate how WebAssembly code can access hardware peripherals in Section 3.4. In Section 3.5 we discuss the overall design of WARDuino. The section goes into further detail on the execution of WebAssembly programs, and the handling of interrupts within the virtual machine. Then we show how we bridge the language barrier with WebAssembly in Section 3.6. Section 3.7 briefly discusses how developers can extend the WARDuino machine themselves to support new or custom hardware peripherals. A formal description of our extensions is given in Section 3.8. In Section 3.9 we give a detailed overview of the available tools for debugging WARDuino applications. We follow this discussion with the evaluation of our implementation in Section 3.10. Finally, we present related works in Section 3.11 and conclude in Section 3.12.

3.4 WARDuino: WebAssembly Programming in Practice

WARDuino is a virtual machine for the 2019 core WebAssembly standard (Rossberg, 2019). The standard does not provide instructions to interact with the environment, for example controlling the pins of a microcontroller. Neither does the *bare-metal execution environment* of embedded devices provide useful abstractions and interfaces to interact with the hardware, in the way a full-fledged operating system might. To address this shortcoming, WARDuino provides a set of primitives to interact with the environment as importable WebAssembly functions.

However, the end goal of WARDuino is not to develop IoT applications in WebAssembly, instead it is meant to enable IoT developers to use high-level programming languages. Many high-level languages can already be compiled to WebAssembly, and we can lift the WARDuino primitives to those programming languages. This means that developers can use the WARDuino primitives as normal functions in their high-level language of choice. We start with a small example to make this general idea more concrete.

3.4.1 Developing IoT programs

Programs in over 40 languages can be compiled to WebAssembly bytecode, and many popular languages provide additional support such as interacting with WebAssembly directly. Developers can use these languages to write programs for WARDuino. We will use the AssemblyScript (The AssemblyScript Project, 2023) programming language as an example in this section. AssemblyScript is a language specifically designed for WebAssembly. The main purpose of AssemblyScript is to allow web developers to use WebAssembly without needing to learn a new language. This is why the language is based on TypeScript, and many TypeScript programs are indeed valid AssemblyScript programs. AssemblyScript's main purpose strongly aligns with our goal of letting developers program embedded systems in the languages they already know and prefer. Furthermore, by being a standalone language AssemblyScript can better prioritize small code size and fast code execution, which are both very important for embedded software.

There is no official list of languages that compile to WebAssembly, but the community maintains a nearly complete list.

Listing 3-1 contains a minimal example of an MQTT program written in AssemblyScript. MQTT (Banks and Gupta, 2014) is one of the most used M2M protocols (Mishra and Kertesz, 2020) for communication in IoT applications.

```
 1 import {delay, MQTT, print, WiFi} from "as-warduino";
 2
 3 function until(attempt: () => void, done: () => boolean): void {
 4     while (!done()) {
 5         delay(1000);
 6         attempt();
 7     }
 8 }
 9
10 export function main(): void {
11     until(() => { WiFi.connect("ssid", "password"); }, WiFi.connected );
12
13     let message = "Connected to wifi network with ip: ";
14     print(message.concat(WiFi.localip()));
15     MQTT.init("broker.example.com", 1883);
16     MQTT.subscribe("helloworld", (topic, payload) => { print(payload); });
17
18     while (true) {
19         until(() => { MQTT.connect("clientid"); }, MQTT.connected);
20         MQTT.poll();
21         delay(1000);
22     }
23 }
```

Listing 3-1. MQTT AssemblyScript program for WARDuino.

It allows devices to communicate with a large network of other devices via a server, designated as the MQTT broker. The broker accepts topic-based messages from clients and passes these messages on to all clients that have subscribed to these topics. WARDuino provides an MQTT module with all the necessary primitives for the microcontroller to function as an MQTT client.

The code in Listing 3-1 starts by importing all necessary WARDuino primitives from the *as-warduino* package for AssemblyScript on Line 1. The `print` function is WARDuino's primitive for printing to the serial bus. The `MQTT` and `WiFi` namespaces expose the functions for communicating via the MQTT protocol and connecting to Wi-Fi networks. On Line 3, we define a helper function `until()` that takes two functions as arguments: `connect` and `connected`. When `until` is called, the `connect` function is executed every second until the `connected` function returns true. We use this function in our program to establish a connection to the Wi-Fi network and the MQTT broker.

The entry point to our program is the `main` function (Line 10). It starts by connecting to the local Wi-Fi network with the help of `until()` and two WARDuino primitives: the `WiFi.connect` function that initiates a connection to a network, and the `WiFi.connected` that returns whether the microcontroller is connected to a Wi-Fi network. Once connected, Line 14 prints the microcontrollers IP address to the serial port. Again, two WARDuino primitives are used: `WiFi.localip` and `print`. The code then configures the URL and port of the MQTT broker (Line 15), and subsequently subscribes to the *helloworld* topic (Line 16). Alongside a topic string, the `MQTT.subscribe` primitive requires a callback function as second argument. WARDuino invokes this callback function for every incoming MQTT message with the set topic.

Now the microcontroller is ready to receive messages from the MQTT broker. A while loop (Line 18) checks if there are messages. To ensure the client remains connected to the server, Line 19 periodically checks if it is still connected, and otherwise attempts to reconnect. After verifying the connection, we call the `MQTT.poll` function to signal WARDuino to check for new messages.

Our example highlights the goal of the WARDuino project: programming microcontrollers from many high-level language. The code illustrates how developers can use all the features of their high-level language, even those WebAssembly itself does not fully support, such as strings and anonymous functions. Using the WARDuino primitives in high-level languages, does require some glue code behind the scenes. The exact details are discussed in Section 3.6.

Underneath the high-level language library, the primitives are implemented in the WARDuino virtual machine as WebAssembly modules. The WebAssembly standard includes the use of custom WebAssembly modules, which can be used to expose functions designed to interact with the environment. In web browsers such custom modules provide interoperability with JavaScript. WARDuino uses the same mechanism to provide access to its primitives: the hardware functionalities of the microcontroller.

The implementation of the primitives is backed by Arduino libraries. Arduino (Banzi, 2008) is an open-source electronics platform that supports a wide range of microcontrollers. By providing a thin layer on top of C++, Arduino increases the portability of programs on microcontrollers. The Arduino platform does an excellent job of defining uniform libraries. For example, the code implementing the iconic blinking LED program is identical for all supported devices. This is made possible by the fact that these microcontroller boards implement a core set of libraries to access and address the input-output pins. The constant `LED_PIN` is one of those provided addresses, it holds the pin number of an LED on the board. By building on top of the Arduino libraries, we can bring the same kind of interoperability to programs compiled to WebAssembly.

Our "built-in" modules provide the most important Arduino features for controlling peripheral devices. These include `GPIO`, `SPI`, `USART` and `PWM` as well as more advanced networking modules. Specifically, we have modules with primitives to connect to Wi-Fi networks and to use the HTTP and MQTT protocols. Currently, all our modules are exposed in one single custom WebAssembly module named "`env`". This is in line with the WebAssembly System Interface (WASI) specification (Hickey et al., 2020). An overview of these primitives and their WebAssembly interface can be found in Chapter F.

3.4.2 Conclusion

In this section we have shown a basic IoT program written in AssemblyScript running on top of our WARDuino VM. Our goal is to enable IoT developers to program microcontrollers in high-level languages. To facilitate this WARDuino allows developers to run WebAssembly on microcontrollers. The idea is that programs in high-level languages are compiled to WebAssembly, and executed by WARDuino. Unfortunately WebAssembly did not yet support interacting with the hardware of the microcontroller. To resolve this we provide `GPIO`, `SPI`, `USART` and `PWM` modules. Network related features are another lacuna of WebAssembly we filled with the `WiFi`, `HTTP`, and `MQTT` modules. Thanks to our modules, WebAssembly becomes a viable platform to program microcontrollers with WARDuino. In the next section we will take a closer look at the architecture of WARDuino.

3.5 WARDuino: Virtual Machine Architecture

The WARDuino virtual machine is at heart a byte-code interpreter for WebAssembly, around which several components have been built to improve the development cycle of IoT software. WebAssembly is the compile target because it tackles both the challenges of *portability* and *low-level coding*. However, for WebAssembly to be a viable platform, we need a way to interact with the environment, i.e. control hardware peripherals, communicate over the internet, and so on. To address this WARDuino includes a set of primitives. Responsive IoT applications require these primitives to handle asynchronous events, standard WebAssembly does not support this. WARDuino is therefore extended with a callback handler to process asynchronous events, and pass them to user-defined callback functions. To tackle the challenge of *debuggability*, WARDuino includes a remote debugger with support for over-the-air updates. Combined with standard debugging operations, the over-the-air updates allow developers to iterate quickly and test fixes while debugging. This way, WARDuino aims to significantly improve on the *slow development cycle* of embedded software. In this section, we give an overview

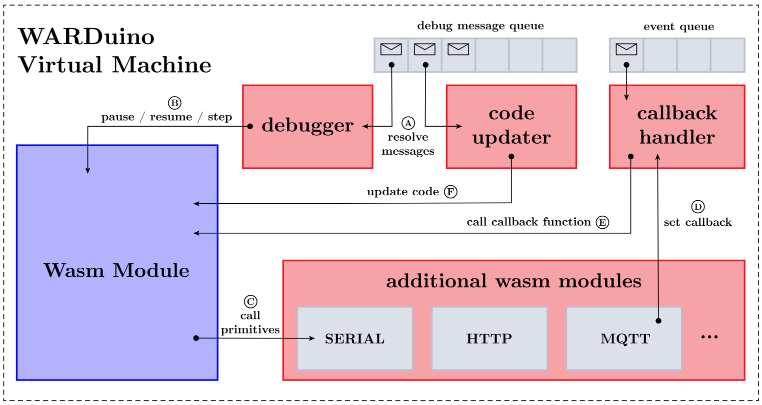


Figure 3-2. This diagram shows the architecture of the WARDuino virtual machine. The different components of the virtual machine are shown in red; the debugger, live code updater, callback handler, and additional WebAssembly modules. External devices can send debug messages to the virtual machine, for both the debugger and live code updater. They are parsed concurrently to the interpretation loop, and placed into the message queue. Asynchronous events for the callback handler are processed analogously, and placed into the event queue. Both queues are shown in gray to indicate that they are populated by platform specific code, outside the interpretation loop. The program (WebAssembly module) executed by the virtual machine is shown in blue. The arrows indicate the interactions between components, which are executed in the interpretation loop.

of the virtual machine architecture, and show how the components interact with the interpretation of the loaded program, as well as each other.

3.5.1 WARDuino Components

Figure 3-2 gives a high-level overview of the virtual machine's architecture, highlighting how the novel components (shown in red) relate to each other and interact with the running program (shown in blue). This program is a WebAssembly module, which will usually be compiled from a high-level language.

WARDuino allows developers to debug and update the running program over the air. The virtual machine can receive update and debug messages over different channels, such as Wi-Fi or the serial port. Whenever a message arrives, it is put in the debug message queue, where it is visible to the main interpretation loop. During interpretation, the virtual machine will periodically check the queue for new messages and resolve them one at a time as shown in Figure 3-2 Ⓐ. Debug messages can instruct the VM to pause, step or resume execution Ⓑ. Update messages can replace the entire WebAssembly module, single functions, or even single variable values Ⓕ.

Importantly, WARDuino contains a number of WebAssembly modules, implementing common libraries and functionality for microcontrollers, such as the `Serial` module or the `HTTP` module Ⓒ. WebAssembly programs can use the primitives from these modules in order to access the hardware of the microcontrollers. Due to the primitives being defined at the level of WebAssembly, it is not always straightforward for developers to use the primitives in their high-level source language. Language specific libraries can enhance the interoperability between a source language and the low-level interfaces of the WebAssembly primitives. Section 3.6 goes into further detail on supporting high-level languages through language specific libraries.

Microcontrollers often receive signals from peripherals through hardware interrupts. These are then typically processed by asynchronous interrupt handlers. This way, the embedded device does not have to block execution by actively waiting for input. Unfortunately, calling a function asynchronously is not supported by standard WebAssembly. In other words, a standard WebAssembly virtual machine cannot call a function to handle asynchronous events such as MQTT messages, hardware interrupts, and so on. To address this shortcoming, we have implemented a novel callback handling system for executing WebAssembly functions when certain events happen. WARDuino programs can use this callback handling system to handle asynchronous events. The MQTT module, for instance, can be used to register a WebAssembly function from the loaded program as a callback for specific events Ⓓ. Whenever these events occur, our callback handler will invoke the registered function and pass all relevant information to it Ⓔ. In the rest of our chapter, we will refer to these functions as callback functions or simply callbacks.

In Appendix F.7 the MQTT *subscribe* primitive illustates how primitives can receive the table index of callback functions as arguments.

3.5.2 WARDuino Interpretation

WebAssembly is a stack based language, defined over an implicit operand stack. This means WebAssembly runtimes do not have to explicitly use this stack. In WARDuino, however, we implement the VM as a stack based virtual machine based on the open source *wac* C-project by Joel Martin[1]. Our WebAssembly operand stack is implemented as two separate stacks: the main operand stack, and a call stack. The call stack keeps track of the active functions and blocks of the program and where the execution should continue once they complete. When initializing the module, we seed the call stack with a call to the main entry point of the program. The main operand stack holds a list of numeric values from which WebAssembly operations pop their arguments, and to which they push their results. This stack starts out empty.

In WebAssembly `loop` and `if` instructions are also placed on the call stack as so-called "blocks". These blocks are needed to ensure that branch (`br`) instructions can only jump to safe locations.

[1]https://github.com/kanaka/wac

Algorithm 3-1 shows the main interpretation loop of WARDuino as pseudocode. WARDuino executes a WebAssembly module *m*, instruction by instruction, in a single loop (Line 3). Before any instruction is interpreted, the *resolveDebugMessage* function checks the debug message queue for new incoming messages (Line 4), resolves the oldest one, and possibly pauses the runtime. If the runtime is not paused, the virtual machine checks the event queue for new asynchronous events, and possibly resolves at most one before starting the actual interpretation. If the runtime is paused however, the virtual machine will go to sleep until a new message arrives in the queue (Line 6). The *awaitDebugMessage* function does not resolve debug messages, instead the code jumps back to the start of the loop and the debug message is resolved by the *resolveDebugMessage* function. We discuss the debug message and event resolution in more detail in, respectively Section 3.5.3 and Section 3.5.5.

```
1  Require module $m$ and running state $s$
2  done ← false, success ← true
3  while done and success do
4      resolveDebugMessage($s$) ▶ Can update the running state $s$
5      if $s$ = paused then
6          awaitDebugMessage() ▶ Wait until debug queue is not empty
7          continue ▶ Go back to the start of the loop
8      resolveEvent() ▶ Run callback for event, if any in the queue
9      opcode ← getOpcode($m$)
10     switch opcode do
11         ...
12         case 0x7c ... 0x8a
13             success ← interpretBinaryi64($m$, opcode) ▶ Perform i64 binary operation
14         ...
15     m.pc ← m.pc +1 ▶ Increment program counter
16 if not success then
17     throw trap ▶ Check if any operation threw a trap
```

Algorithm 3-1. Main loop for interpretation in the WARDuino virtual machine.

For interpreting instructions, the virtual machine keeps track of its own program pointer *m.pc*, which points to the next instruction to be executed in the program buffer. We may dereference this pointer to get the next opcode to execute (Line 9), for example `0x7f`. A switch statement then matches the current opcode (Line 10). For our example, the switch determines our opcode to be a binary operator for 64-bit integers that will be handled by the *interpretBinaryi64* function. This function resolves the instruction further and returns whether it succeeded. If so, the while-loop continues, and the next opcode is processed. Otherwise, success will become false, and the while loop will stop interpretation.

We refer interested readers to the paper by Haas et al. (2017) for more information on traps in WebAssembly.

When the interpretation loop stops due to a failure, the virtual machine will throw the underlying trap (Line 17). Alternatively, interpretation halts whenever the end instruction (`0x0b`) of the main entry point is reached. In this case, the *done* variable will be set to `true`, and the main interpretation loop will stop successfully without throwing a trap.

In Algorithm 3-2 we show the most relevant parts of *interpretBinaryi64*. The function is used to interpret all binary operators defined by WebAssembly on 64-bit integers. First, it gets the two arguments for the binary operation from the operand stack (Line 2). Next, the function matches the opcode with a specific operation, in our case the `i64.div_s` operation. If the arguments are valid for the operation, the division is executed (Line 10), the result is placed on the top of the stack (Line 12), and the function returns `true` indicating success (Line 13). When the function encounters an illegal operation such as a division by zero, it returns `false` instead (Line 8). In that case, the main loop of the virtual machine will stop interpretation of the program and throw an exception, as shown on Line 17 in Algorithm 3-1. Most of the code for interpreting the WebAssembly operations is structured analogously to the function highlighted in Algorithm 3-2.

3.5.3 Resolving Debug Messages

Debug messages for WARDuino are received and parsed concurrently from the main interpretation loop, by device and communication channel specific code. Messages are placed on a FIFO queue. The interpretation loop will check this queue at the start of each iteration (Algorithm 3-1, Line 4). When the queue is not empty, exactly one message is processed. This means the running state of the virtual machine can change, for instance from running to paused. If the program should be paused, the virtual machine will wait until the debug messages queue is not empty (Algorithm 3-1, Line 6), before continuing at the start of the interpretation loop.

```
function interpretBinaryi64(m, opcode)
    $d, e \leftarrow$ popStack($m$, 2)
    $f \leftarrow 0$
    switch opcode do
        ...
        case 0x7f
            if e = 0 then
                throw "division by zero"
                return false
            $f \leftarrow d \div e$
        ...
    pushToStack($m$, $f$)
    return true
```

Algorithm 3-2. Function to interpret operators for 64-bit integers.

3.5.4 Calling primitives

WARDuino primitives are exposed to the running program as imported WebAssembly functions. In fact, all imported functions must be primitives defined in the virtual machine, since the current version allows only one user-defined WebAssembly module to be loaded at a time. The WARDuino primitives are exposed through one single "`env`" module. This is in line with the WebAssembly System Interface (WASI) specification (Hickey et al., 2020). As stated before, the `env` module is implemented in the VM.

When an opcode specifies that a WebAssembly function must be called, some extra work is needed. First, the current instruction and stack pointers are stored in a frame and pushed to the call stack. The program pointer *m.pc* is replaced by the first instruction of the function to be called. In the next interation of the while loop (Algorithm 3-1, Line 3), the virtual machine will execute the function's instructions. When the end instruction (`0x0b`) is encountered, the function has finished. The execution must now continue from the point where the call was originally made. To jump back to this place we pop the last frame from the call stack. This frame contains the program counter where we ought to continue execution. We reset our program and stack pointer to the appropriate values and continue executing. If the function returned a value, it will reside on the top of the main operand stack.

WARDuino programs can be developed in multiple high-level languages thanks to WebAssembly. Currently, the WARDuino project includes example programs written in Rust, AssemblyScript, and C. We go into further detail on the support for high-level languages in Section 3.6.

Rust and AssemblyScript programs can be found under the tutorials folder in the GitHub repository, and the benchmarks folder includes C programs.

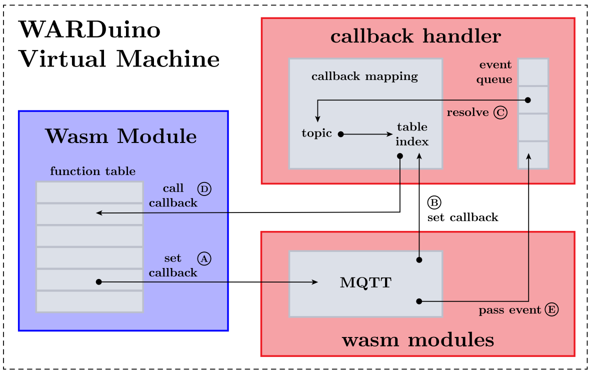


Figure 3-3. This diagram shows how callbacks are resolved in the virtual machine. The event queue is populated with events concurrently to the interpretation loop, any time the microcontroller receives a hardware interrupt.

The primitive operations (i.e. `digital_write`) are implemented in the WARDuino virtual machine in native C code. This implementation depends on the target microcontroller platform. Currently, WARDuino focuses on the Arduino platform which can be used with many families of embedded devices, such as ESP32's, Arduino boards, and some Raspberry Pi devices. To illustrate the portability of WARDuino, the virtual machine includes partial support for ESP IDF. The WebAssembly interface of the primitives is the same for both implementations, to provide the best portability for WARDuino programs. To make these interfaces compatible with the VM the primitives conform to the standard WebAssembly calling conventions, i.e. they read their arguments from the stack and place their return value on the stack.

The current status of supported platforms can be found on the documentation website.

3.5.5 Callback Handling

The WARDuino callback handling system is used to call WebAssembly functions when specific real-world events occur. These events can range from interrupts caused by a local button press to MQTT messages arriving over Wi-Fi. Similar to debug messages, asynchronous events are received and placed into a queue concurrently to the main interpretation loop. As shown in Algorithm 3-1, the main interpretation loop will resolve a single event—if any is present—immediately after checking for incoming debug messages.

Before events can be resolved, callback functions must be registered for the topics of the events the program expects to receive. Figure 3-3 gives a schematic overview of the callback handling system in the WARDuino virtual machine. When developing our callback handling system, the two most important concerns are: (a) to keep the system lightweight, and (b) to offer the flexibility required to support the wide range of asynchronous protocols and libraries that already exist for microcontrollers. Precisely for these concerns, we developed a reactive event-driven system.

Callback handling in WARDuino works as follows. Within WebAssembly, functions can be stored into a table, enabling them to be referenced by their table indices. WARDuino uses this same mechanism for the callback functions. For instance, consider an MQTT `subscribe` primitive that subscribes to an MQTT topic with a given callback function (more information can be found in Appendix F.7). In the WebAssembly program we pass the table index of the callback function to the `subscribe` primitive, as shown in Figure 3-3 . The WARDuino MQTT library then uses this index to register a callback with the global callback handler in the WARDuino virtual machine. This handler holds a mapping of topic strings to table indices. Each topic string can be mapped to at most one callback.

We do not allow multiple callbacks for a single topic string at the level of WebAssembly instructions, but the WebAssembly primitives we built on top of this system do in fact support registering multiple callbacks. This gives the same result for developers writing programs in a high-level language in WARDuino. However, it does make a significant difference for the WebAssembly specification, as we will explain in Section 3.8.5.

Whenever the virtual machine wants to resolve an event (Algorithm 3-1, Line 8), the callback handler takes the oldest event from the queue and looks up its topic in the callback mapping . The mapping returns the table index of the registered callback function. Through this index the callback handler can set up the call for the correct WebAssembly function on the call stack, and add the topic and payload of the event as arguments to the operand stack . In other words, the callback handler does not execute the callback functions itself, it merely sets up the appropriate calls on the stacks. When the interpretation loop resumes it will automatically execute the callback function. Executing callbacks is therefore completely transparent to the virtual machine, since it is just another function call. Furthermore, the virtual machine does not need to know whether an event was actually processed by the callback handler. This does force callback functions to never return a value. However, this is a reasonable requirement that many other microcontroller platforms also impose on their interrupt callbacks (Banzi, 2008, Espressif Systems, 2023a). After all, since the callbacks are executed concurrently to interpretation and in complete isolation, there is no way of using the return value anyway. Therefore, after the callback function is resolved, the interpretation of the program continues as if no additional function was called.

The precise signature is shown as part of the operational semantics in Section 3.8.5.

There is a possible pitfall with adding callbacks to the call stack at any point during execution. In light of the microcontroller's limited memory, it is easy for the call stack to grow too rapidly. Therefore, we prohibit that callbacks

Note that the blocked callbacks do not get lost, they are processed after the current callback completes.

interrupt other callbacks. In practice, the virtual machine keeps track of whether a callback is being executed by adding a marker on the call stack just before the callback. When the virtual machine encounters this marker again, it knows that the callback completed. So when *resolveEvent* is called in Algorithm 3-1, and the last callback has not yet completed, the callback handler will never resolve an event.

3.5.6 Summary

The WARDuino virtual machine has all the ingredients to develop IoT applications for microcontrollers in high-level languages. We discuss the practicalities of using high-level languages in Section 3.6, The virtual machine includes primitives that give the WebAssembly programs access to the hardware peripherals and other IoT capabilities of the microcontrollers. The architecture of our virtual machine is extensible in many ways, as we discuss in Section 3.7. Through the framework discussed in Section 3.7 many Arduino libraries can be implemented in WARDuino.

A current list of already implemented libraries can be found in the official documentation of WARDuino.

Our novel callback handling system provides an event-based callback system where WebAssembly functions can be assigned to topics. Events are handled concurrently to interpretation of the WebAssembly program, and callback functions subscribed to the corresponding topic are called at well-defined points in the program execution. This way, these functions can react to hardware interrupts or other asynchronous events.

Because microcontrollers often do not have a keyboard or screen, WARDuino provides remote debugging support. This enables developers to set breakpoints and pause or step the execution remotely. Even more, WARDuino allows over-the-air updates of variables and functions as a whole. Section 3.9 contains more details on the tools available for debugging with WARDuino. In the next section we will look further into the features of the WARDuino project that aim to overcome this language barrier.

3.6 Support for High-level Languages

In WARDuino hardware functionality is exposed through WebAssembly primitives. These low-level building blocks allow developers to build Internet of Things applications for microcontrollers. While our primitives are valuable in their own right, their interfaces are low-level compared to the high-level languages we want to use them from. Unfortunately, there is no generic way to create a high-level interface that is fit for every language. Languages differ in their design philosophy or may even use a different programming paradigm altogether. Furthermore, when creating interfaces, an implementer

```
export function fib(n: i32): i32 {
    let a = 0, b = 1;
    if (n > 0) {
        while (--n) {
            let t = a + b;
            a = b;
            b = t;
        }
        return b;
    }
    return a;
}
```

Listing 3-2. AssemblyScript function that calculates the n-th Fibonacci number.

can choose to what degree they wish to offer language interoperability. In this section, we discuss how to implement high-level interfaces for WARDuino primitives, by showing different levels of interoperability for the AssemblyScript programming language. This implementation strategy is similar for all languages compiling to WebAssembly.

3.6.1 AssemblyScript as Source Languages

Let us first consider AssemblyScript code that does not use WARDuino features. Listing 3-2 shows a function that calculates the n-th Fibonacci number. When we compile a basic program like this to WebAssembly, any runtime fully implementing the official WebAssembly specification should be able to run it. However, this is not the kind of program we typically want to run on microcontrollers with WARDuino. The computed Fibonacci only lives inside the microcontroller and is not visible to the outside world. To make it visible, the program needs to affect the pins of the microcontroller in some way.

3.6.2 Importing External WebAssembly Functions

Our next example (Listing 3-3) is a small program that uses WARDuino primitives in AssemblyScript. On the left side we show the code for the traditional blinking LED program. The right side contains the minimal glue code required to make the program work. The glue code imports our WARDuino primitives and defines some useful constant values.

Entities from external WebAssembly modules can be imported in AssemblyScript with an `@external` annotation. This annotation specifies both the module and primitive name to be imported. The function declaration below the annotation mirrors the imported primitives interface. On Lines 7 and 8, for example, WARDuino's `chip_delay` primitive is imported and declared

```
import * from "as-warduino";

export function main(): void {
  let led = 16;
  pinMode(led, OUTPUT);

  let pause = 1000;
  while (true) {
    digitalWrite(led, HIGH);
    delay(pause);
    digitalWrite(led, LOW);
    delay(pause);
  }
}
```

```
export const LOW: u32 = 0;
export const HIGH: u32 = 1;
export const OUTPUT: u32 = 0x2;

@external("env", "chip_delay")
export declare function delay(ms: u32): void;

@external("env", "chip_pin_mode")
export declare function pinMode(pin: u32,
                                mode: u32): void;

@external("env", "chip_digital_write")
export declare function digitalWrite(pin: u32,
                                     value: u32): void;
```

Listing 3-3. Blinking LED example in AssemblyScript with the necessary and minimal glue code.

to AssemblyScript as the function `delay` which has one `u32` argument and returns `void`.

Our glue code will be the same for all AssemblyScript programs. As such, we can implement it as an AssemblyScript library. By using the `export` keyword we export our declarations. This approach abstracts away the underlying WARDuino interfaces. Developers can now simply import the `as-warduino` library as shown on the first line of the blinking LED program. When they do this, they can use the WARDuino primitives as if they were normal TypeScript functions.

3.6.3 Using Interfaces with Strings

For the blinking LED example, we only used primitives with simple numeric parameters and return values. Primitives with string arguments and return values are less straightforward to port. As detailed in Appendix F.4, we represent strings as two integers: a start index in the memory and the length of the string.

Unlike WebAssembly, AssemblyScript contains types for representing and manipulating strings directly. As such it is unnatural for developers to pass strings as numeric values to functions in AssemblyScript. When AssemblyScript code is compiled to WebAssembly, the compiler translates strings into a new representation using only basic numeric types. We created a similar translation from strings to numeric types when implementing primitives with strings in our VM. Unfortunately, we have no guarantee that these two translations are the same. AssemblyScript encodes strings with UTF-16 by

```
import * from "as-warduino";
export function main(): void {
  // ... connect to Wi-Fi ...
  // Send HTTP request
  let url = "https://example.com/post";
  let body = "Bridge the Language Gap";
  let content_type = "text/plain";
  let response = new ArrayBuffer(100);
  httpPOST(String.UTF8.encode(url, true),
    String.UTF8.byteLength(url, true),
    String.UTF8.encode(body, true),
    String.UTF8.byteLength(body, true),
    String.UTF8.encode(content_type, true),
    String.UTF8.byteLength(content_type, true),
    response, response.byteLength);
}
```

```
export const WL_CONNECTED: u32 = 3;

@external("env", "http_get")
export declare function httpGET(
  url: ArrayBuffer, url_len: u32,
  buffer: ArrayBuffer,
  buffer_size: u32): i32;

@external("env", "http_post")
export declare function httpPOST(
  url: ArrayBuffer,
  url_len: u32,
  body: ArrayBuffer,
  body_len: u32,
  content_type: ArrayBuffer,
  content_type_len: u32,
  buffer: ArrayBuffer,
  buffer_size: u32): i32;
```

Listing 3-4. Example AssemblyScript program with HTTP GET without strings.

default, which uses two bytes to encode most characters. But as we expect to use mostly ASCII characters, and we want to keep code sizes small for microcontrollers, we prefer to use UTF-8 encoding instead, where characters are represented primarily with only one byte.

If we use the same approach as we did for the LED example we arrive at the code in Listing 3-4. It shows a simple AssemblyScript program that uses WARDuino's HTTP POST primitive. On the right side of the figure, we give the minimal glue code that only imports the WARDuino primitive with their exact interfaces. This means that each string argument must be translated to two integers by the developer. This is not the only hurdle they must overcome. Due to the encoding inconsistencies between AssemblyScript and WARDuino, strings must be manually transformed to UTF-8. Additionally, to receive a response, WARDuino expects a memory slice as last argument where the response is stored to. The developer must allocate an `ArrayBuffer` for this and pass this as the last two arguments of the call.

Working with this minimal glue code requires very specific knowledge about the inner workings of WARDuino. This is not desirable. The minimal glue code does not effectively bridge the differences in abstraction levels between AssemblyScript and our WARDuino primitives. We can improve on the glue code by extending it with functions that actually use strings instead of numeric values.

```
import {HTTP} from "as-warduino";

export function main(): void {
  // ... connect to Wi-Fi ...
  // Send HTTP request
  let response = HTTP.post(
    "https://example.com/post",
    "Bridge the Language Gap",
    "text/plain");
}
```

```
@external("env", "http_get")
declare function _http_get(...): i32;

@external("env", "http_post")
declare function _http_post(...): i32;

export namespace HTTP {
  function get(url: string,
               buffer: ArrayBuffer): i32 {
    return get(String.UTF8.encode(url, true),
               String.UTF8.byteLength(url, true),
               buffer, buffer.byteLength);}

  function post(url: string, body: string,
                content_type: string): string {
    let response = new ArrayBuffer(100);
    _http_post(String.UTF8.encode(url, true),
      String.UTF8.byteLength(url, true),
      String.UTF8.encode(body, true),
      String.UTF8.byteLength(body, true),
      String.UTF8.encode(content_type, true),
      String.UTF8.byteLength(content_type, true),
      response, response.byteLength);
    return String.UTF8.decode(response, true);}
}
```

Listing 3-5. Example AssemblyScript program with HTTP GET with glue code for strings.

The improved glue code for the HTTP primitives unburdens the developer from managing text encoding, by handling it in the AssemblyScript library. Listing 3-5 shows the new library code on the right. The code imports the WARDuino HTTP primitives under the names `_http_post` and `_http_get`. Instead of exporting these functions directly, the glue code wraps them in another function. These wrappers take care of the necessary conversions and have a more natural external interface: they use AssemblyScript strings as argument and return type. The library now exports these more natural wrappers instead of the "raw" WARDuino primitives. We can even use AssemblyScript namespaces to group the HTTP functions together, to avoid name collisions and form a logical interface. Developers can now write the much more naturally feeling code on the left of Listing 3-5, where the post function accepts strings and returns a string. The type annotations on our wrapper function provide an extra benefit: they allow the AssemblyScript type checker to validate whether the function is indeed called with strings.

```
import {HTTP} from "as-warduino";

export function main(): void {
  // ... connect to Wi-Fi ...

  // Send HTTP request
  let options: HTTP.Options = {
    url: "https://example.com/post",
    body: "Bridge the Language Gap",
    content_type: "text/plain"
  };
  let response = HTTP.post(options);
}
```

```
export namespace HTTP {
class Options { url: string; body: string;
                content_type: string; }

function post(options: Options): string {
  let response = new ArrayBuffer(100);
  _http_post(String.UTF8.encode(options.url, true),
    String.UTF8.byteLength(options.url, true),
    String.UTF8.encode(options.body, true),
    String.UTF8.byteLength(options.body, true),
    String.UTF8.encode(options.content_type, true),

    String.UTF8.byteLength(options.content_typ
    true),
    response, response.byteLength);
  return String.UTF8.decode(response, true);
}}
```

Listing 3-6. Example AssemblyScript program with HTTP GET with glue code for objects.

3.6.4 Higher Levels of Language Interoperability

While the string version of the HTTP POST primitive is already a huge improvement over the numeric version, it still requires three string arguments. Conventionally, TypeScript-like languages use objects to send complex arguments to functions. In Listing 3-6, we show a program, and the associated glue code where the exported `post` function accepts an object of class `Options` rather than three strings. The class declaration on the first line of the right listing in Listing 3-6 defines that a value of type `Options` must have the keys `url`, `body` and `content_type` which all should be assigned to a string. Thanks to this definition, AssemblyScript's type system enforces that all required keys are present in the arguments to `post`.

3.6.5 Other Modules and Languages

Language interoperability is needed to facilitate access to WARDuino's primitives. We implement it as a library that can be easily imported by developers. Although we only highlighted the HTTP module in this section, our AssemblyScript library contains glue code for all the WARDuino primitives discussed in section

Because different programming languages follow different conventions or even different programming paradigms all together, language interoperability must be dealt with separately for each language. Luckily we can follow the same approach as we did for AssemblyScript to create interoperability libraries for other languages. As an example, we also created a WARDuino library for Rust, another popular programming language with WebAssembly support. Rust encodes strings as UTF-8 by default, so our library for this language does not need to change the string encoding.

Interoperability can be provided at different levels. We have seen three implementations of AssemblyScript glue code for WARDuino's HTTP module. The first version simply exported the raw" WARDuino primitives. This meant that the developer needed to know the inner workings of WARDuino to use these functions. They needed to know how strings were represented, for example. Our second version abstracted the interface, and allows developers to use it without having to worry about WARDuino internals. By abstracting the interface we also allowed AssemblyScript to validate the types of arguments to our primitives. Finally, in a third version we adapted the glue code to adhere more closely to the informal conventions of the language. By doing so, WARDuino has similar function signatures to other libraries in the language.

3.6.6 Summary

To use high-level languages with WARDuino in practice, the primitives need to be lifted from their WebAssembly interface to the host language. In this section we showed how this interoperability can be implemented to various degrees, ranging from using the low-level WebAssembly interface directly, to a high-level interface that integrates completely with the paradigms of the higher-level language. The examples listed here can be used as a general recipe for implementing language integration libraries in other languages that compile to WebAssembly. For instance, programs written in C, Rust, and AssemblyScript have been used with WARDuino using the implementation strategies outlined here.

Example programs can be found on the documentation website and in the GitHub repository

3.7 Extending the Virtual Machine

In the previous sections we explained that WARDuino has native support for the most significant features of the microcontroller, such as monot("GPIO->SPI"), `PWM`, `SPI`, as well as communication protocols, such as `HTTP` and `MQTT`. However, we need to keep the memory constraints of the microcontrollers in mind. Given the *hardware limitations* of embedded devices, it is important to keep the WARDuino virtual machine as small as possible. We therefore restricted the supported libraries to the most essential ones for embedded applications. Furthermore, when compiling the virtual machine, developers can disable select primitives to reduce the size of WARDuino

```
typedef struct Type {
  uint8_t form;
  uint32_t
  param_count;
  uint32_t *params;
  uint32_t
  result_count;
  uint32_t *results;
  uint64_t mask;
} Type;
```

```
uint32_t
param_U32_arr_len2[2]
  = {U32, U32};

Type twoToNoneU32 = {
  .form =  FUNC,
  .param_count =  2,
  .params =
  param_U32_arr_len2,
  .result_count =  0,
  .results =  nullptr,
  .mask =  0x80011
};
```

```
def_prim(digital_write,
         twoToNoneU32)
         {
  auto pin =
  arg1.uint32;
  auto val =
  arg0.uint32;
  digitalWrite(pin,
  val);
  pop_args(2);
  return true;
}
```

Listing 3-7. *Left*: The Type struct. *Middle*: A Type specifier for a primitive that takes two 32-bit unsigned integer (u32) and returns nothing. *Right*: The implementation of the `digital_write` primitive.

further. On the other hand, developers can add new primitives to the WARDuino VM for specific functionality or hardware they require for their projects. In this section we give an overview of how to add new primitives to the WARDuino VM.

For readers familiar with OS architectures, this is somewhat familiar to the unikernel approach.

3.7.1 Creating User-Defined Primitives

In this section, we show how we implemented the `digital_write` primitive in the WARDuino virtual machine. Developers that need a library that is not supported by our VM can use a similar approach to add it to WARDuino.

Our virtual machine keeps a table of all primitive functions. Each entry contains a name, a type specifier and an implementation. Programmers can extend this table by providing these details. The type specifier is used by the VM to validate if the primitive is called with the right arguments. If an inconsistency is detected at runtime, WARDuino throws an error. Note that this will not happen if the programmer has type-checked their code. Almost all compilers that produce WebAssembly will produce type-checked code. Our runtime checks are useful during the development of the primitives themselves.

The process for adding new primitives consists of the four steps we describe below.

First, the programmer needs to indicate that the number of primitives has changed by increasing the `NUM_PRIMITIVES` constant, this variable is used to allocate the primitives table.

Second, the implementer defines the type of their custom primitive. In WARDuino the type of a function is represented by the struct shown on the left side of Listing 3-7. The form field indicates the form of the type, in the virtual machine, which is one of: function type, table type, memory type and global. For primitives this field will always be a “function type” i.e. `FUNC`. The following fields indicate how many arguments (`param_count`) and how many return values (`result_count`) the type has. Both counts are followed by a pointer to an array containing the specific types of the arguments/return values. Finally, each type has a `mask` that allows for quick comparison of types in the VM. The `get_type_mask` function can derive the appropriate mask for a type struct. We have predefined the most common types, these are available when defining new types. One of these predefined types is the type for a primitive taking two 32-bit integer as an argument and returning nothing, its definition is shown in the middle of Listing 3-7.

Third, after the programmer has defined the type specifier, the primitive itself can be implemented. On the right side of Listing 3-7, we show the implementation of our `digital_write` primitive. Primitives are defined using our `def_prim` macro. This macro expects two arguments, and a function body. The arguments are the name and type specifier of the primitive. The function body implements the primitive. Developers can use the macros `arg0` to `arg9` to access the first 10 values on the stack. The `arg0` macro returns the argument that was pushed most recently onto the stack. A `pop_args()` macro allows popping values from the stack. The implementation may use any library that is available at compilation time. Additionally, it may use the callback system, an example of which will be discussed in Section 3.7.2. Every primitive must return a boolean value. This value is used to indicate whether the function succeeded. If `false` is returned, the primitive has failed, and the virtual machine will throw a trap.

Finally, the implementer makes the custom function available to the rest of the virtual machine and the WebAssembly modules it executes. This only involves adding the primitive into the `primitives` table with the `install_primitive` macro. Once this is done, the primitive is ready to be used in WebAssembly programs for WARDuino.

3.7.2 Using Callbacks with Primitives

Our callback system enables developers to implement asynchronous libraries as modules for WARDuino. In this section we will illustrate how our callback system can be used to define asynchronous primitives. To do this, we look at the implementation of two MQTT primitives: `mqtt_init` and `mqtt_subscribe`.

```
 1 def_prim(mqtt_init, threeToNoneU32) {  // Initialize the Arduino MQTT Client
 2   uint32_t server_param = arg2.uint32; uint32_t length = arg1.uint32;
 3   uint32_t port = arg0.uint32;
 4   const char *server = parse_utf8_string(m->memory.bytes, length,
     server_param).c_str();
 5   mqttClient.setServer(server, port);
 6
 7   // Add MQTT messages as events to callback handling system
 8   mqttClient.setCallback([](const char *topic, const unsigned char *payload,
 9                             unsigned int length) {
10     CallbackHandler::push_event(topic, payload, length);
11   });
12   pop_args(3);
13   return true;
14 }
15
16 def_prim(mqtt_subscribe, threeToOneU32) {  // Subscribe to a MQTT topic
17   uint32_t topic_param = arg2.uint32; uint32_t topic_length = arg1.uint32;
18     uint32_t fidx = arg0.uint32;
19   const char *topic = parse_utf8_string(m->memory.bytes, topic_length,
     topic_param).c_str();
20
21   Callback c = Callback(m, topic, fidx);
22   CallbackHandler::add_callback(c);  // Register callback function with WARDuino
23
24   bool ret = mqttClient.subscribe(topic);
25   pop_args(2);
26   pushInt32((int)ret);
27   return true;
28 }
```

Listing 3-8. Implementation of the `mqtt_init` (top) and `mqtt_subscribe` (bottom) WARDuino MQTT primitives using the event-based callback handling system.

The heavy lifting of our MQTT module is carried out by the PubSubClient Arduino library. Our primitives act as a wrapper around this library.

Listing 3-8 shows the implementation of the `mqtt_init` primitive on Lines 1 to 13. This primitive initializes the underlying PubSubClient library, and sets the URL and port of the MQTT broker to connect to (Line 4). The PubSubClient library only supports assigning one callback that will receive all the events for all subscribed topics. WARDuino's callback handling system is more flexible and allows developers to assign different callback function for each topic. During initialization of the MQTT module we use a lambda expression to set the callback of the PubSubClient library on Line 7. This function forwards each incoming MQTT event to WARDuino's `CallbackHandler`. The `CallbackHandler` will then in turn invoke the right WebAssembly callbacks when messages arrive.

Lines 15-27 of Listing 3-8 implement the MQTT subscribe primitive. It allows developers to register a WebAssembly function as a handler for a specific MQTT topic. After retrieving and parsing the arguments to the primitive (Lines 16-17), the function does three things. First, it creates a `Callback` object that holds a reference to the WebAssembly module, the topic, and the index of the WebAssembly callback function (Line 19). Second, this `Callback` object is added to the `CallbackHandler`. Third, in order for the subscribed messages to be passed to the `Callbackhandler` on Line 8, the function needs to tell the underlying PubSubClient library to subscribe to the given topic (Line 22).

Our new callback handling system allows WARDuino to define asynchronous primitives. These primitives can handle asynchronous foreign events such as hardware interrupts. To work with asynchronous primitives, developers simply use them in their programs to add callback functions. WARDuino will then transparently execute them in response to incoming events.

3.7.3 Summary

In this section, we showed how users can define new primitives and add them to the VM in a four-step process. Using this system users can add support for new sensors and actuators to WARDuino. When implementing primitives, developers can use the internal callback handling system of the WARDuino virtual machine, to create asynchronous primitives.

The callback handling system is a key aspect of the virtual machine that extends standard WebAssembly. Another such aspect is the remote debugger. We discuss both components as formal extension to the WebAssembly specification in the next section.

3.8 Formal Specification of WARDuino

WebAssembly is strongly typed, and both its type system and execution are precisely defined by a small step semantic. Such a precise definition gives the WebAssembly community a universal way to propose changes and extensions to the standard.

In this section we formalize WARDuino's architecture, by presenting it as three extensions to the WebAssembly specification. We start with a very brief summary of WebAssembly's formal description in Section 3.8.1. This overview is followed by the small step semantics for our remote debugging (Section 3.8.2), over-the-air updates (Section 3.8.4), and callback handling features (Section 3.8.5). Each of these extensions can be defined entirely independently of the others, but here we present the over-the-air updates as an extension of the debugger semantics to highlight their compatibility. Primitives are part of the custom modules, and are therefore out of scope for the specification and will not be formalized here.

3.8.1 WebAssembly

WebAssembly is a memory-safe, compact and fast bytecode format designed to serve as a universal compilation target. The bytecode is defined as a stack-based virtual instruction set architecture, which is strictly typed to allow for fast static validation. However, its design features some major departures from other instruction sets, and resembles much more the structure of programming languages than other bytecode formats. Importantly, it features memory sandboxing and well-defined interfacing through modules, as well as structured control flow to prevent control flow hijacking. The original use-case of WebAssembly was to bring the high-performance of low-level languages such as C and Rust to the web.

The execution of a WebAssembly program is described by the small step reduction relation $\hookrightarrow_i$ over a configuration triple representing the state of the VM, where i indicates the index of the current executing module. The index i is necessary since WebAssembly can load multiple modules at a time. A configuration contains one global store s, the local values v^* and the active instruction sequence e^* being executed. The rules are of the form $s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*$. A more detailed overview of the WebAssembly specification can be found in Chapter E.

3.8.2 Remote Debugging Extensions

To formalize our debugging system, we extend the operational semantics of WebAssembly with the necessary remote debugging constructs. The goal of these extensions, is to provide constructs that are as lightweight as possible while still being powerful enough to provide the most common remote debugging facilities. We follow the recipe for defining a debugger semantics as outlined by Torres Lopez et al. (2019), where the semantics of the debugger are defined in terms of the underlying language's semantics: in this case the WebAssembly specifications. One advantage of this approach, is that it leads to a very concise description of the debugger semantics. More importantly, with this recipe you get a debugger whose semantics are observationally equivalent to those of the underlying language's semantics. This means that the debugger does not interfere with the underlying semantics, and therefore, only observes real executions. Or more precisely, any execution in the WARDuino debugger corresponds to an execution of a WebAssembly program,

$$\begin{array}{lll}
\text{(Debugger State)} & \mathrm{dbg} ::= \{\mathrm{rs}, \mathrm{msg}_i, \mathrm{msg}_o, \mathrm{s}, \mathrm{bp}\} \\
\text{(Running State)} & \mathrm{rs} ::= \mathrm{play} \mid \mathrm{pause} \\
\text{(Messages)} & \mathrm{msg} ::= \varnothing \mid \mathrm{play} \mid \mathrm{pause} \mid \mathrm{step} \mid \mathrm{dump} \mid \mathrm{break}^{+}\ \mathrm{id} \mid \mathrm{break}^{-}\ \mathrm{id}
\end{array}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^* \qquad \mathrm{id}(e^*) \notin \mathrm{bp}}{\{\mathrm{play}, \varnothing, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{play}, \varnothing, \varnothing, \mathrm{s}, \mathrm{bp}\}; v'^*; e'^*}\ \text{vm-run}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{\{\mathrm{pause}, \mathrm{step}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{pause}, \varnothing, \varnothing, \mathrm{s'}, \mathrm{bp}\}; v'^*; e'^*}\ \text{db-step}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{\{\mathrm{pause}, \mathrm{dump}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{pause}, \varnothing, \mathrm{msg}, \mathrm{s'}, \mathrm{bp}\}; v'^*; e'^*}\ \text{db-dump}$$

$$\frac{}{\{\mathrm{rs}, \mathrm{pause}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{pause}, \varnothing, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^*}\ \text{db-pause}$$

$$\frac{}{\{\mathrm{pause}, \mathrm{play}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{play}, \varnothing, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^*}\ \text{db-pause}$$

$$\frac{}{\{\mathrm{rs}, \mathrm{break}^{+}\ \mathrm{id}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{rs}, \varnothing, \varnothing, \mathrm{s}, (\mathrm{bp} \cup \{\mathrm{id}\})\}; v^*; e^*}\ \text{db-bp-add}$$

$$\frac{}{\{\mathrm{rs}, \mathrm{break}^{-}\ \mathrm{id}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{rs}, \varnothing, \varnothing, \mathrm{s}, (\mathrm{bp} \setminus \{\mathrm{id}\})\}; v^*; e^*}\ \text{db-bp-rem}$$

$$\frac{\mathrm{rs} \neq \mathrm{pause} \qquad \mathrm{id}(e^*) \in \mathrm{bp}}{\{\mathrm{rs}, \mathrm{break}^{-}\ \mathrm{id}, \varnothing, \mathrm{s}, \mathrm{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\mathrm{rs}, \varnothing, \varnothing, \mathrm{s}, (\mathrm{bp} \setminus \{\mathrm{id}\})\}; v^*; e^*}\ \text{db-bp-rem}$$

Figure 3-4. Core debugger semantics. Small step reduction rules ($\hookrightarrow_{d,i}$) for the WARDuino remote debugger, as extensions to the WebAssembly semantics.

and conversely that any execution of a program is observed by the debugger. The recipe also makes it straightforward to proof this non-interference of the debugger, as we will show in Section 3.8.3.

At the top of Figure 3-4 we give an overview of our syntactic extensions to the operational semantics of WebAssembly that provide remote debugging support. In the semantics we abstract away the underlying communication primitives, we assume that there is a system in place that reads messages from a stream and places them in the inbox. A concrete implementation may allow communication over the serial port, an HTTP connection or the SPI bus. For ease of exposition all these possible communication methods are modeled through messages *msg*.

To differentiate the debugger semantics from the underlying language, we write the reduction relation as ($\hookrightarrow_{d,i}$), where d indicates the debugging semantics and i is still the index for the currently executing module. But thanks to how we define the debugger semantics, the operation of a program during debugging is described by the combined reduction rules from the WebAssembly semantics and our debugger semantics.

The semantics of the debugger consists of a state transitioning system where each state consists of a debugger state *dbg*, zero or more local values v^* and a focused operation e^*. The main state of the debugger *dbg* is represented as a 5-tuple that holds the running state *rs*, the last incoming message msg_i the last outgoing message msg_o, the WebAssembly store s and, a set of breakpoints bp. The running state indicates whether the virtual machine is paused (PAUSE) or running (PLAY). Rules for setting msg_i when messages are received, and for clearing msg_o when delivering outbound messages are omitted from our semantics as these are dependent on the communication method. The reduction rules for remote debugging are shown in the lower part of Figure 3-4, we describe them below.

vm-run When in the PLAY state with no incoming or outgoing messages and no applicable breakpoints, the debugger takes one small step of the small step operational semantics $\hookrightarrow_i$. That is, a regular WebAssembly step is taken.

db-pause When the debugger receives a *pause* message, the debugger transitions to the PAUSE state. Note that it is allowed to transition from any previous state to the paused state. After transitioning to the paused state, the rule VM-RUN is no longer applicable.

db-dump In the paused state the debugger can request a dump of the virtual machine's state. This dump is communicated to the debugging host by an outgoing message, which contains the full WebAssembly state and the breakpoints of the debugger.

db-run When the debugger is in the PAUSE state, the programmer can restart execution by sending a *run* message.

db-step When the debugger receives the step message in the PAUSE state, it takes one step ($\hookrightarrow_i$). The debugger remains in the PAUSE state.

db-bp-add Breakpoints can be added in any run state.

db-bp-rem Breakpoints can be removed in any run state.

db-break When the debugger is not in the PAUSE state, and the *id* of the currently executing expression is in the list of breakpoints the debugger transitions to the PAUSE state.

It is important to note that the DB-DUMP adds a message to the outgoing messages, but the other rules expect the outgoing messages to be empty. Since the communication is abstracted away, we assume that incoming and outgoing messages are added and removed by an external system, and the debugging semantics cannot get stuck. In other words, the other rules in the semantics only handle incoming messages after all the outgoing messages are removed by the external communication system.

Below, we show three derived commands for stepping through the WebAssembly code after a breakpoint is hit. These are not written as rules in our formal semantics as they are simply a combination of the rules we already introduced.

step-into This stepping command is offered only for function calls. In order for the debugger client to verify whether this command should be active it can request a dump of the current execution and enable the step-into command in the GUI. Execution of the STEP-INTO command is the same as DB-STEP.

step-out When the programmer is debugging inside a function, they might want to step out of the function call. Because the end of a function is an actual instruction in WebAssembly the debugger can inspect the body of the function and add breakpoints for all the exit points of the function. Important here is that the debugger needs to take note of the call stack at the moment a STEP-OUT is requested. To handle recursive calls correctly, the program should only be paused if one of the breakpoints is hit while the call stack has the same height. If the breakpoint is hit on a larger call stack, the program should be resumed (by sending *play*).

step-over Like step-into, step-over only activates for the next call instructions. Instead of following the call the step-over stepping command stops the debugger when the call is finished. The instruction sequence to express step-over with our basic debugging constructs are: take one step to go into the function (DB-STEP), execute the STEP-OUT stepping command.

The semantics allow for more elaborate debugging operations to be build on top of those presented here. However, the previous three operations represent the most widely used debug operations, and should therefore accommodate most developers debugging needs.

3.8.3 Proof of Observational Equivalence

In order to proof the observational equivalence between the debugger semantics and the base language semantics, we use the same proof method as Torres Lopez et al. (2019), which proves observational equivalence by a weak bisimulation argument. With this proof, we show that if an arbitrary WebAssembly program P can take a step to a program P', the debugging semantics allows the debugger to reach the program P' from the program P by one or more debugging steps. The other way around, if the debugger allows a program P to transition to a program P', the normal WebAssembly evaluation will also allow the program P transition to the program P'.

In the semantics we leave out the specifics of the communication, and assume the incoming messages are added to the debugging state in the correct order. For the proof, we will reason over a stream of messages instead of a single one. Thanks to the recipe we follow for the debugger semantics, the proof follows almost directly by construction.

Theorem 3-1. (Observational equivalence) Let S be the WebAssembly configuration $\{s; v^*; e^*\}$, for which there exists a transition ($\hookrightarrow_i$) to another configuration S' with $\{s'; v'^*; e'^*\}$. Let the debugging configuration $(\{rs, msg_i, msg_o, s, bp\}; v^*; e^*)$ with running state rs, incoming messages msg_i, outgoing messages msg_o, and set of breakpoints bp; be such that processing the stream of incoming message M^* takes exactly one externally visible step (VM-RUN or DB-STEP) in the debugger semantic ($\hookrightarrow_e$), then:

$$(\{s; v^*; e^*\} \hookrightarrow_i \{s'; v'^*; e'^*\})$$
$$\Leftrightarrow$$
$$(\{rs, msg_i, msg_o, s, bp\}; v^*; e^* \hookrightarrow_e \{rs, msg_i, msg_o, s', bp\}; v'^*; e'^*)$$

The left-hand side of the double implication presents a single step in the normal evaluation ($\hookrightarrow_i$) of a WebAssembly program, while the right-hand side presents one or more steps in the debugging semantics ($\hookrightarrow^*_{d,i}$) where only a single step is externally visible ($\hookrightarrow_e$). We will start by sketching the proof for the first implication, that is, an evaluation step in the WebAssembly semantics implies an equivalent series of debugging steps.

Proof Sketch. In case the debugger is in the PLAY state, two cases need to be considered. First, if there is no applicable breakpoint, the only applicable rule that is externally visible is the VM-RUN rule. Applying this rule, will transition the state S to S' by construction. Second, a number of internal rules of the debugger can transition the system into a PAUSE state (e.g., DB-PAUSE, DB-BREAK). By assumption, processing the stream of messages M^*

leads to exactly one externally visible step. None of the internally visible rules (e.g., DB-PAUSE, DB-BP-ADD) change the underlying state S of the program. This means, that whenever the externally visible step is taken, it will do so with the same underlying state S as at the start of the debugging steps. The only externally visible steps, are DB-STEP and VM-RUN, which take exactly the same transition as the underlying WebAssembly semantics. In case the debugger starts in the PAUSE state, a similar argument holds. □

Now we will provide the proof sketch for the second implication, that is a series of evaluation steps in the debugger semantics implies an equivalent evaluation step in the WebAssembly semantics.

Proof Sketch. Only the VM-RUN and the DB-STEP rules change the WebAssembly configuration S in the debugging configuration D. By construction, both rules rely directly on the underlying WebAssembly semantics for transitioning S to S'. □

3.8.4 Safe Over-the-air Code Updates

Our over-the-air update system allows programmers to upload new programs and to update functions and local variables. Here, we present the system as an extension of the debugger semantics, but the over-the-air updates can also be defined on their own without the debugger as we show in Chapter G. Note that the observational equivalence of the debugger semantics will no longer hold with the addition of over-the-air updates, since they allow for arbitrary code changes.

As with the debug semantics, rules for setting msg_i are omitted.

Figure 3-5 gives an overview of the additional reduction rules to dynamically update a WebAssembly program. In these rules the debug messages are extended with three update messages. In order to improve the usability of the semantics, the over-the-air updates can only be executed in the paused state. Additionally, the program will remain in the paused state to allow setting new breakpoints.

upload-m An *upload* message instructs WebAssembly to restart execution with a new set of modules m^*. We require all these modules to be well typed, $(\vdash m)^*$. The meta-function bootstrap represents WebAssembly's initialization procedure, described in the original WebAssembly chapter (Haas et al., 2017). Note that this procedure replaces the entire configuration, including the WebAssembly state, locals and stack. Furthermore, upon receiving the *upload* message the debugger state is reset and all breakpoints removed.

$$\begin{array}{lll} \text{(Messages)} & \text{msg} ::= \ldots \mid \text{upload } m^* \mid \text{update}_f \ \text{id}_i \ \text{id}_f \text{code}_f \mid \text{update}_l j v \\ \text{(Closure)} & \text{cl} ::= \{\text{inst } i, \text{idx } j, \text{code } f\} \end{array}$$

$$\frac{(\vdash m)^* \qquad \{s', v'^*, e'^*\} = \text{bootstrap}(m^*)}{\{\text{pause}, \text{upload } m^*, \varnothing, s, \text{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\text{pause}, \varnothing, \varnothing, s', \varnothing\}; v'^*; e'^*} \ \textsc{upload-m}$$

$$\frac{s' = \text{update}_f(s, \text{id}_i, \text{id}_f, \text{code}_f)}{\{\text{pause}, \text{update}_f \ \text{id}_i \ \text{id}_f \ \text{code}_f, \varnothing, s, \text{bp}\}; v^*; e^* \hookrightarrow_{d,i} \{\text{pause}, \varnothing, \varnothing, s', \text{bp}\}; v'^*; e^*} \ \textsc{update-f}$$

$$\frac{\vdash v : \epsilon \to t \qquad \vdash v^* : \epsilon \to t}{\{\text{pause}, \text{update}_l j v', \varnothing, s, \text{bp}\}; v_1^j v v_2^k; e^* \hookrightarrow_{d,i} \{\text{pause}, \varnothing, \varnothing, s, \text{bp}\}; v_1^j v' v_2^k; e^*} \ \textsc{update-local}$$

Figure 3-5. Extension of the debugging rules (Figure 3-4) with safe over-the-air updates.

update-f The message to update a function specifies the function to update and its new code (code_f). To identify a function we must supply the ID of the instance id_i it lives in and the index it exists at id_f in that instance. The meta-function update_f replaces the function in the state s and validates that its type remains the same.

WebAssembly's formalization transforms every function in a closure that holds its code f and the module instance it was originally defined in. When a function is imported into another module or placed in a table, its closure is copied to the other module instance. Because the closure holds the original instance, it can be executed in the right context. When it calls other functions, for example, these must be the functions from the original module rather than from the calling module. We extended closures with an extra identifier idx, which holds the index of the function in its defining module. Thanks to this, the update_f can replace all closures in s where the inst is id_i and the idx is id_f.

update-local Updating a local is done with an update_l message. This message holds the index of the local to be updated and its new value. We validate that the type of the new value is the same constant type $\epsilon \to t$ as the original value at the chosen index.

Note that we only allow updates if the underlying types remain the same. While this provides safety, it can still have undesirable effects. For example when updating, in the middle of a recursive function the new base conditions might have already been exceeded. The WARDuino VM does not tackle these kinds of problems. In future work we hope to improve on this by incorporating techniques from work on dynamic software updates (Tesone et al., 2018).

$$\begin{array}{lrl}
\text{(Store)} & \mathrm{s} ::= & \{\ldots, \mathrm{status\ rs}, \mathrm{evt\ evt}^*, \mathrm{cbs\ cbs}\} \\
\text{(Running state)} & \mathrm{rs} ::= & \mathrm{play} \mid \mathrm{pause} \mid \mathrm{callback} \\
\text{(Event)} & \mathrm{evt} ::= & \{\mathrm{topic\ memslice}, \mathrm{payload\ memslice}\} \\
\text{(Memory slice)} & \mathrm{memslice} ::= & \{\mathrm{start\ i32}, \mathrm{length\ i32}\} \\
\text{(Callback map)} & \mathrm{cbs}[x \to f] ::= & \lambda x.\ \mathrm{if}\ y = x\ \mathrm{then}\ f\ \mathrm{else\ cbs}\ y \\
\text{(Instructions)} & \mathrm{e} ::= & \ldots \mid \mathrm{event.push} \mid \mathrm{callback}\ \{e^*\}\ e^*\ \mathrm{end} \\
 & & \mid \mathrm{callback.set\ memslice} \mid \mathrm{callback.get\ memslice} \\
 & & \mid \mathrm{callback.drop\ memslice}
\end{array}$$

$$\frac{C \vdash e^* : \epsilon \to \epsilon \qquad C \vdash e_0^* : \mathrm{tf}}{C \vdash \mathrm{callback}\ \{e_0^*\} e^*\ \mathrm{end} : \mathrm{tf}} \qquad \frac{}{C \vdash \mathrm{callback.set\ memslice} : \mathrm{i32} \to \epsilon}$$

$$\frac{}{C \vdash \mathrm{callback.get\ memslice} : \epsilon \to \mathrm{i32}} \qquad \frac{}{C \vdash \mathrm{callback.drop\ memslice} : \epsilon \to \epsilon}$$

$$\frac{}{C \vdash \mathrm{callback.drop\ memslice} : \mathrm{i32} \times \mathrm{i32} \times \mathrm{i32} \times \mathrm{i32} \to \epsilon}$$

$$\text{(Contexts)}\ C ::= \left\{\mathrm{func\ tf}^*, \mathrm{global\ tg}^*, \mathrm{table}\ n^?, \mathrm{memory}\ n^?, \mathrm{local}\ t^*, \mathrm{label}\ (t^*)^*, \mathrm{return}\ (t^*)^?\right\}$$

Figure 3-6. The extended WebAssembly abstract syntax (top), and the typing rules (bottom) for the WARDuino callback handling system.

3.8.5 Callback Handling

In Section 3.5.5 we discussed the architecture of our callback handling system. The system follows an event-driven approach, where ordinary WebAssembly functions are registered as callbacks for a specific event. Before we can formalize how callbacks are executed by the WebAssembly runtime, we must extend the abstract syntax with the necessary concepts: events, callbacks, memory slices, and callback mappings. The top part of Figure 3-6 shows how we extend the syntax, starting from the WebAssembly abstract syntax with the additional syntax for remote debugging.

First, we add the CALLBACK state to the running state *rs* defined for the remote debugging extension. This state indicates that the virtual machine is executing a callback function. This state only changes the behavior of the callback handlers, which will not resolve any new events until the state changes back to PLAY. In other words, the CALLBACK and PLAY states are completely interchangeable in the context of the remote debugging extension.

Second, we add a list of events to the global store. This list represents the event queue of the callback handler. Events must contain one topic and one payload, which are both memory slices (*memslices*). A *memslice* refers to an area in WebAssembly linear memory. This buffer of bytes is defined in the syntax as a tuple of numeric values, the start index and the length. So while the buffers will most likely be strings in practice, the formalization intentionally refrains from specifying anything about the memory content.

This way we steer clear of trying to add strings to WebAssembly, which is not our goal. Events can be added to the event queue with the instruction. As the definition of an event in Figure 3-6 shows, an event contains two *memslices*: a topic and a payload. The instruction expects four numeric values on the stack, reflected in its type shown in the lower part of the same figure.

Third, the callback mapping is added to the global store. Adding, removing and retrieving functions from the callback mapping can be done from WebAssembly with the new instructions, , and respectively. Unlike WebAssembly instructions such as we cannot use an index space to refer to callback functions, because callbacks are stored in a mapping from strings to table indices. For this reason, the instructions for adding, removing and retrieving callbacks, take a memory *memslice* containing the topic string. Note that the map returns at most one function index for each topic string. We choose to limit the amount of callbacks per topic in this way, because the mapping would otherwise become too complicated for a simple low-level instruction set such as WebAssembly. However, we can achieve the same result for end-users by supporting multiple callbacks at the level of WebAssembly primitives instead.

We only formalize one callback for one topic, as multiple callbacks, we would need to introduce a new list type.

Finally, we extend the WebAssembly instructions with a new instruction. This construct is similar to the administrative instructions from the WebAssembly standard, used to simplify reasoning over control flow (Haas et al., 2017). Figure 3-6 shows the specific syntax and typing rules for this new construct. It holds two lists of instructions. The first sequence e_0^*, between curly braces, is the continuation of the callback. These are the instructions that will be executed once the callback has been completely resolved. The second list of instructions e^* is the body of the callback, which will be evaluated first. Because the callback can be called at any time, its body must leave the stack unchanged after its reduction, so execution can continue as it would have without the callback. Furthermore, because the stack can have any possible state when the callback is created, the body of the callback cannot expect any arguments from the stack. In other words, the body of the callback takes zero arguments and returns nothing (type $\epsilon \rightarrow \epsilon$).

With these syntactic extensions to WebAssembly, we are now able to formalize how events are processed, and callbacks executed. We list the additional small step reduction rules in Figure 3-7. To keep the rules readable, we will shorten *memslices* by simply writing *topic* or *payload* instead of every numeric value. For instance, in the first rule, $s_{\text{evt}}(0)_{\text{topic}}$ is a shorter form for $\left(\text{i32.const}, s_{\text{evt}}(0)_{\text{topic.start}}\right) \left(\text{i32.const}, s_{\text{evt}}(0)_{\text{topic.length}}\right)$. Similarly, we write the lookup for the table index of a callback function in the short form: $\left(s_{\text{cbs}}\left(s_{\text{evt}}(0)_{\text{topic}}\right)\right)$. This expression corresponds with exactly one (i32.const index) instruction. We describe each of the rules below.

$$\frac{s_{\text{cbs}}\Big(s_{\text{evt}}(0)_{\text{topic}}\Big) \neq \text{nil} \qquad s_{\text{status}} = \text{play} \qquad s'_{\text{status}} = \text{callback} \qquad s'_{\text{evt}} = \text{pop}(s_{\text{evt}}) \qquad \text{tf} = \text{i32} \times \text{i32} \times \text{i32} \times \text{i32} \rightarrow \epsilon}{s; v^*; e^* \hookrightarrow_i s'; v^*; \text{callback}\ \{e^*\}\Big(s_{\text{evt}}(0)_{\text{topic}}\Big)\Big(s_{\text{evt}}(0)_{\text{payload}}\Big)\Big(s_{\text{cbs}}\Big(s_{\text{evt}}(0)_{\text{topic}}\Big)\Big)(\text{call\_indirect tf})\ \text{end}}\ \textsc{Callback}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v*; e'*}{s; v^*; \text{callback}\ \{e_0^*\}\ e^*\ \text{end} \hookrightarrow_i s'; v^*; \text{callback}\ \{e_0^*\}\ e'^*\ \text{end}}\ \textsc{Step-Callback}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v*; e'*}{s; v^*; \text{callback}\ \{e_0^*\}\ \epsilon\ \text{end} \hookrightarrow_i s'; v^*; e_0^*}\ \textsc{Resume}$$

$$\frac{s; v^*; e^* \hookrightarrow_i s'; v*; e'*}{s; v^*; \text{callback}\ \{e_0^*\}\ \epsilon\ \text{end} \hookrightarrow_i s'; v^*; e_0^*}\ \textsc{Skip-Message}$$

$$\frac{}{s; v^*; \text{callback}\ \{e^*\}\ \text{trap end} \hookrightarrow_i s'; v^*; \text{trap}}\ \textsc{Callback-Trap}$$

$$\frac{s'_{\text{cbs}} = s_{\text{cbs}}[\text{topic} \rightarrow j]}{s; v^*; (\text{i32.const}\ j)(\text{callback.set topic}) \hookrightarrow_i s'; v^*; \text{trap}}\ \textsc{Register}$$

$$\frac{s'_{\text{cbs}}[\text{topic} \rightarrow \text{nil}]}{s; v^*; (\text{callback.drop topic}) \hookrightarrow_i s'; v^*; \epsilon}\ \textsc{Deregister}$$

$$\frac{s_{\text{tab}}(i, j)_{\text{code}} \neq (\text{func i32 i32} \rightarrow \epsilon\ \text{local}\ t^* e^*)}{s; v^*; (\text{i32.const}\ j)(\text{callback.set topic}) \hookrightarrow_i s; v^*; \text{trap}}\ \textsc{Register-Trap}$$

$$\frac{}{s; v^*; (\text{callback.get topic}) \hookrightarrow_i s'; v^*; s_{\text{cbs}}(\text{topic})}\ \textsc{Get-Callback}$$

$$\frac{s'_{\text{evt}} = \text{push}(s_{\text{evt}}, \{\text{topic}, \text{payload}\})}{s; v^*; (\text{topic})(\text{payload})(\text{event.push}) \hookrightarrow_i s'; v^*; \epsilon}\ \textsc{Push-Event}$$

Figure 3-7. Small step reduction rules for the WARDuino calback handling system.

callback The CALLBACK reduction rule shows how the WebAssembly interpreter can replace the instruction sequence e^* with a callback construct, whenever there are unprocessed events and no other callback is being processed. We do not allow nested callback constructs. To enforce this, we change the running state to the CALLBACK value in the CALLBACK rule and change it back to PLAY in the RESUME rule. The CALLBACK construct replaces the instruction sequence with a instruction. The replaced instruction sequence, is kept by the callback construct as a continuation (between curly braces). The new stack only holds the callback construct, which contains an indirect call with the index returned by the callback mapping s_{cbs}. Before the indirect call and the table index, the rule adds the topic and payload of the event to the stack as arguments for that function call. This means that every callback function must have type $\text{i32} \times \text{i32} \times \text{i32} \times 32 \rightarrow \epsilon$. Because we place the arguments on the stack at the same time as the indirect call, the body of the callback as a whole still has type $\epsilon \rightarrow \epsilon$, as specified in Figure 3-6.

step-callback The STEP-CALLBACK rule describes how the code inside the body of the is executed until it is empty.

resume Once a callback is completed, the RESUME rule replace s the empty construct with its stored continuation. From this point onward evaluation resumes normally. We know that the body of the construct will always become empty because its type is $\epsilon \rightarrow \epsilon$.

skip-message When no callback function is registered in the callback mapping s_{cbs} for the event at the top of the FIFO event queue $s_{evt}(0)$, the skip-message rule takes a step by removing the top event from the event queue in the store.

callback-trap Because code within the construct is reduced with the existing WebAssembly reduction rules, it can result in a . In that case, the trap should be propagated upward by replacing the entire callback with it.

register The takes an immediate memory slice, which corresponds with a topic string. The instruction takes a table index j pointing to a function from the stack, and updates the callback mapping so the set of indices returned for the given topic now includes the table index j.

register-trap If the table index that the pops from the stack, does not refer to a WebAssembly function with the correct type ($32 \times \text{i32} \times \text{i32} \times \text{i32} \rightarrow \epsilon$), the instruction will result in a trap as shown in rule REGISTER-TRAP.

deregister Callback functions can be removed from the callback mapping. The function updates the mapping by removing the index j from the set of indices corresponding with the topic immediate.

callbacks Looking up callbacks can be done with the instructions, which returns a vector of the table indices registered for the given topic.

push-event The instruction adds a new event to the global event queue. Analogous to the previous callback instructions, this instruction takes a topic immediate. The payload of the event is taken from the stack. As shown, in Figure 3-6 a payload is a memory slice, which consists of two numeric values: the offset in memory and the length of the slice.

Our formalization closely describes the callback handling system as introduced in Section 3.5.5. It does so with a limited amount of reduction rules. We can keep the formalization small because we reuse the existing instruction when adding a callback to the sequence of instructions. Using a smaller set of rules, means it is easier to reason about the formalization and the impact of the extension on WebAssembly. Furthermore, it means implementing the extension in a WebAssembly runtime is less work, because where the formalization reuses parts of the WebAssembly specification, the existing infrastructure of the runtime can likewise be reused.

So far we have not directly mentioned the interaction between the debugger and callback handling system. The operational semantics as presented here, allow for callback instructions to be introduced at any step. This also holds for the debugging steps. During debugging, WARDuino can jump to a callback function whenever it steps to the next instruction. This can lead to confusing behavior, and is the main reason why debugging concurrent programs is so complicated (Torres Lopez et al., 2019). The implementation of WARDuino features debug instructions to make debugging concurrent programs easier, as described by Lauwaerts et al. (2022). These debug instructions control the callback handling system by choosing when callback functions are executed. This is a powerful tool for debugging concurrent programs, which by design allows developers to explore interleavings of callbacks that are not possible outside the debugger semantics. This means that the observational equivalence no longer holds for the debugger.

3.8.6 Discussion

The small step semantics of WebAssembly precisely defines how a program executes, allowing embedders, such as web browsers or WARDuino, to create different compatible implementations of the same specification. Additionally, the formalization provides a uniform way to propose extensions to the WebAssembly standard. In this section we formalized three extensions to WebAssembly: remote debugging, over-the-air updates and an asynchronous callback handling system. Other runtimes can use our formalizations to implement (some of) these extensions for their embedding of WebAssembly.

Our first extension, remote debugging allows developers to remotely control a WebAssembly runtime. By sending it messages, they can set breakpoints, pause the execution, inspect values and so on. Our formalization is based on a debugging recipe (Torres Lopez et al., 2019) that transforms a language semantics into an observationally equivalent debugger semantics. This means that no execution path observed by the debugger semantics is not observed by the underlying language semantics, and that no execution path observed by the language semantics cannot be observed by the debugger semantics as well. Thanks to the recipe used to construct the debugger semantics, the proof for observational equivalence follows almost directly by construction as shown in our proof sketch.

With our over-the-air update system programmers can safely replace a running WebAssembly module, specific functions or specific locals. We specify that if functions or locals are updated, they must maintain their original type. When uploading entire modules, the new modules must be valid. Over-the-air updates are defined orthogonal to debugging, this allows these two extensions to be used side-by-side or independently of one another. With the addition of over-the-air updates, the debugger semantics are no longer observationally equivalent to the language semantics, since it can now update code arbitrarily. Nevertheless, the semantics are important to show how the system preserves WebAssembly types across updates. Furthermore, we are not aware of any previous attempts to describe over-the-air updates of WebAssembly code, or describe over-the-air updates of binary code with an operational semantic. There is however, a limited body of work that looks into the theoretical aspects of over-the-air updates. We go into further detail on the existing works in Section 3.11.2.

Our last extension shows how WARDuino can handle asynchronous events in WebAssembly. Hardware interrupts or asynchronous network communication is facilitated by our novel callback handling system. This system allows developers to transparently interrupt an executing WebAssembly program to execute a callback that deals with an incoming event. While the semantics presented here are certainly novel, there are several other proposals for supporting asynchronous code in WebAssembly. We discuss these works in Section 3.11.4.

3.9 Tool Support for WARDuino

WARDuino aims to make it easier for programmers to debug applications running on embedded devices. There are several approaches and tools that WARDuino offers in this regard. The virtual machine includes its own remote debugger, while the use of WebAssembly makes it much easier to build emulators. In this section we give a detailed overview of the different tools available for debugging WARDuino applications.

3.9.1 Debugging WARDuino Programs Remotely

Microcontrollers are often not equipped with a screen and keyboard, therefore, we allow programmers to debug their programs remotely. We offer a command line tool and a Visual Studio Code plugin. First, we give an overview of our debugging protocol and the debugger architecture, before we show the VS Code plugin build on top of this debugger.

3.9.1.1 Debugging Protocol

The WARDuino VM facilitates remote debugging by allowing debug messages to be sent over a variety of carriers. We have experimented both with wired (USB) and wireless (Wi-Fi) communication means. In theory any communication channel can be used.

Our protocol consists of a set of instructions sent as messages to the virtual machine. The first byte of each message indicates its type. Depending on the type, the first byte is followed by a byte sequence consisting of the arguments of the messages. When a debug message is received by the microcontroller, it is caught by an interrupt handler. This handler reads the available data and passes it on to the virtual machine. The VM in turn waits for a full debugging package to arrive. Once a package is complete, it is placed in the debugging queue for final processing. The debugging queue is polled before each executed instruction. If a message is present in the queue, the appropriate action is taken.

There are four broad categories of debug messages supported by WARDuino.

1. Basic debugging

The one-byte *play*, *pause* and *step* messages respectively run, pause or step the currently executing program by setting the VM's run state. The debugger keeps track of the run state that is either PAUSE or RUN. When in the PAUSE state, WARDuino waits for a *play* or *step* message to process the next instruction. In the RUN state, the VM executes normally.

2. Breakpoints

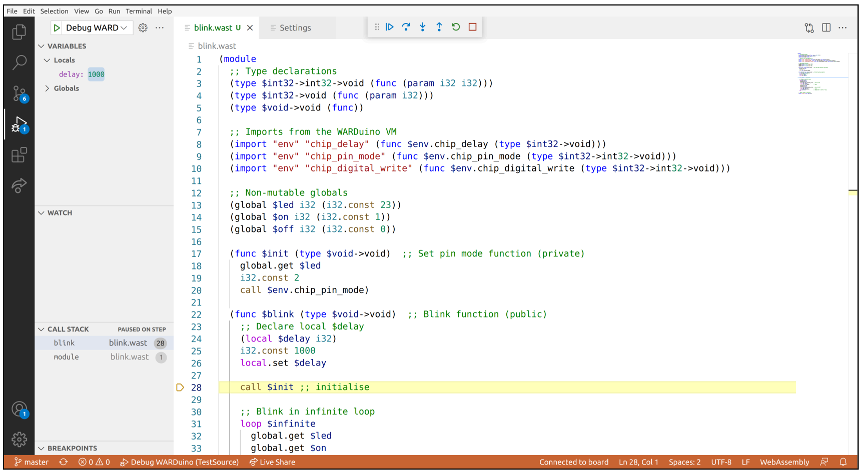
The remote debug messages $break^{+}$ and $break^{-}$ carry a pointer to an instruction where a user wishes to PAUSE execution. These breakpoints are stored in a set. The set is checked before each instruction. When a break point is hit, the run state is set to PAUSE, and an acknowledgment is sent to the remote debugger.

3. Inspection

When a *dump* message is received, the run state is set to PAUSE and a JSON representation of the state of the virtual machine is sent back to the user. The JSON object obtained from a *dump* message contains the call stack, a list of functions, and the current instruction pointer. An example output is shown in

4. Over-the-air updates

The remote debug messages for over-the-air updates, $update_f$ and $update_l$, both contain the ID of the function or local to update, and its



```
1  (module
2    ;; Type declarations
3    (type $int32->int32->void (func (param i32 i32)))
4    (type $int32->void (func (param i32)))
5    (type $void->void (func))
6
7    ;; Imports from the WARDuino VM
8    (import "env" "chip_delay" (func $env.chip_delay (type $int32->void)))
9    (import "env" "chip_pin_mode" (func $env.chip_pin_mode (type $int32->int32->void)))
10   (import "env" "chip_digital_write" (func $env.chip_digital_write (type $int32->int32->void)))
11
12   ;; Non-mutable globals
13   (global $led i32 (i32.const 23))
14   (global $on i32 (i32.const 1))
15   (global $off i32 (i32.const 0))
16
17   (func $init (type $void->void)  ;; Set pin mode function (private)
18     global.get $led
19     i32.const 2
20     call $env.chip_pin_mode)
21
22   (func $blink (type $void->void)  ;; Blink function (public)
23     ;; Declare local $delay
24     (local $delay i32)
25     i32.const 1000
26     local.set $delay
27
28     call $init ;; initialise
29
30     ;; Blink in infinite loop
31     loop $infinite
32       global.get $led
33       global.get $on
```

Figure 3-8. Screenshot of the VS Code debugger extension for WARDuino with a WebAssembly program (blinking LED).

new value. The virtual machine should be in the PAUSE state to process such as change. Updating a local simply updates the appropriate value on the stack. Updating a function on the other hand is slightly more elaborate. First, the bytecode of the function is parsed and the appropriate structures are built. If the new function has an identical type, the pointer in WARDuino's function table is replaced with a reference to the new code. Any running call of the existing function will continue to work with the old code. New calls will use the updated code.

3.9.1.2 Visual Studio Code Debugger

The remote debugging system we presented so far, allows developers to debug WebAssembly code on microcontrollers remotely via a terminal. Debugging via a terminal with memorized commands is something few developers are used to. Instead, most developers use debuggers with a user-friendly interface such as the GUI debugger in an IDE. We created a plugin for the widely used IDE Visual Studio Code (VSCode) that allows developers to remotely debug WARDuino instances.

At its core, a debugging plugin for an IDE sends the debug messages described above on behalf of the developer. This removes the need for them to know our specific debugging API. Having a plugin send the same messages as a developer would in the terminal, is enough to support WebAssembly level debugging in an IDE. Figure 3-8 shows a screenshot of the VS Code plugin debugging a remotely running WebAssembly blink program that is currently paused on the highlighted line (Line 28). The plugin also support provisional source mapping for AssemblyScript, which means most features of the plugin can be used to debug AssemblyScript code directly. The buttons at the top of the screen allow the execution to be resumed and steps to be taken. In the sidebar on the left we can inspect local variables and edit them. These edits are then immediately propagated to the device with a $update_l$ message. At the bottom of the sidebar, we can inspect the call stack.

3.9.2 Building Emulators

A common practice in IoT development is to use emulators to verify, as well as possible, the correctness of the code before running it on the custom hardware (Makhshari and Mesbah, 2021). Emulation is designed to minimize the need for debugging on microcontrollers. Unfortunately, this approach is far from ideal as non-trivial differences between the emulator and the real device will exist. Emulated sensors may for example not produce real-world values. Instead, they might report a fixed value that does not change over time. Furthermore, real changes in sensor values may appear differently to real devices due to physical effects such as contact bounce (chatter). These differences can cause an application that works in an emulator not to work on a real device. That is why the WARDuino project focuses on delivering an alternative approach, where testing can be performed on the custom hardware itself. We argue, that this approach can catch more errors than emulation, and leads to a shorter development cycle.

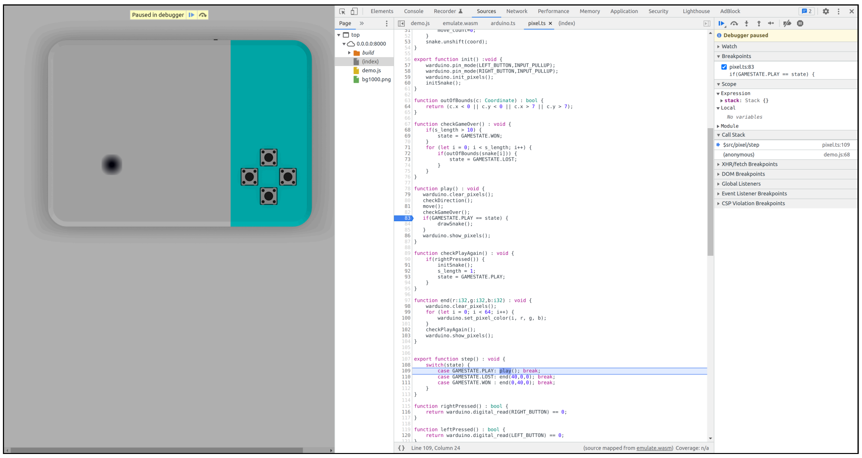
However, emulated verification can still be useful. This is certainly the case when developers work with highly specific hardware, which may not always be at hand. WARDuino also helps when using the emulation approach, providing an easy workflow for implementing emulators for custom hardware.

Developers using WARDuino compile their programs to WebAssembly, which means their code also runs in web browsers. The only missing components are the WARDuino primitives that control the custom hardware. In other words, implementing an emulator for custom hardware comes down to creating a HTML based device and writing a minimal set of hooks to substitute the WARDuino primitives.

Our ESP32-powered game controller is based on Johan Von Konow's Soko-Day design.

We implemented a snake game for a custom game controller that uses an 8x8 LED matrix. To support this LED matrix peripheral, we extended the WARDuino virtual machine with the Arduino Adafruit NeoPixel library[2] using the process described in section Section 3.7.1. We wrote a snake game in AssemblyScript. To implement the emulator we write small JavaScript functions for each primitive we use in the code. The entire JavaScript code for the emulator consists of only 150 lines. It allows us to run the snake game in

[2]https://www.arduino.cc/reference/en/libraries/adafruit-neopixel/



```
            move_count=0;
        }
        snake.unshift(coord);
}

export function init() :void {
    warduino.pin_mode(LEFT_BUTTON,INPUT_PULLUP);
    warduino.pin_mode(RIGHT_BUTTON,INPUT_PULLUP);
    warduino.init_pixels();
    initSnake();
}

function outOfBounds(c: Coordinate) : bool {
    return (c.x < 0 || c.y < 0 || c.x > 7 || c.y > 7);
}

function checkGameOver() : void {
    if(s_length > 10) {
        state = GAMESTATE.WON;
    }
    for (let i = 0; i < s_length; i++) {
        if(outOfBounds(snake[i])) {
            state = GAMESTATE.LOST;
        }
    }
}

function play() : void {
   warduino.clear_pixels();
   checkDirection();
   move();
   checkGameOver();
   if(GAMESTATE.PLAY == state) {
        drawSnake();
   }
   warduino.show_pixels();
}

function checkPlayAgain() : void {
    if(rightPressed()) {
        initSnake();
        s_length = 1;
        state = GAMESTATE.PLAY;
    }
}

function end(r:i32,g:i32,b:i32) : void {
    warduino.clear_pixels();
    for (let i = 0; i < 64; i++) {
        warduino.set_pixel_color(i, r, g, b);
    }
    checkPlayAgain();
    warduino.show_pixels();
}


export function step() : void {
    switch(state) {
        case GAMESTATE.PLAY: play(); break;
        case GAMESTATE.LOST: end(40,0,0); break;
        case GAMESTATE.WON : end(0,40,0); break;
    }
}

function rightPressed() : bool {
    return warduino.digital_read(RIGHT_BUTTON) == 0;
}

function leftPressed() : bool {
    return warduino.digital_read(LEFT_BUTTON) == 0;
```

Line 109, Column 24 (source mapped from emulate.wasm) Coverage: n/a

pixel.ts:83
`if(GAMESTATE.PLAY == state) {`

Expression
stack: Stack {}
Local
No variables
Module
Call Stack
$src/pixel/step pixel.ts:109
(anonymous) demo.js:68

Figure 3-9. Screenshot of a browser-based emulator for custom hardware. The right side of the screenshot shows how the browser debugger can pause and step through the original AssemblyScript code written for WARDuino.

the browser. Additionally, we can use the browser's debugger to step through the AssemblyScript code, as shown in Figure 3-9.

3.9.3 Debugging High-Level Languages

Debugging at the low level of WebAssembly is not workable for any real-world application. Our goal is to debug the high-level source code. So-called "source maps" make this possible. Source maps align the line numbers of the original code to the WebAssembly instructions they compile to. They typically come as a separate file containing only the mapping. By cross-referencing WARDuino's instruction pointer with the source maps, we can derive the line we are executing in our program written in a high-level language.

Figure 3-9 shows source mapping in action, the emulator is executing WebAssembly, but the code view on the right shows AssemblyScript code. The WebAssembly instruction pointer is translated into a line number in the AssemblyScript code by using source maps. AssemblyScript is not the only language with source mapping. Most high-level languages with good support for WebAssembly, can generate a source map during compilation. In fact, any language using the LLVM compiler infrastructure, can generate all the necessary information in DWARF format from which a source map can be derived.

3.10 Evaluation

In this section we evaluate the WARDuino VM in terms of its runtime performance, and conformance to the WebAssembly standard. Section 3.10.1 illustrates the usability of WARDuino in the real world, by presenting a qualitative evaluation of a smart light application written in AssemblyScript (Section 3.10.1). Next, in Section 3.10.2, we evaluate the performance of the virtual machine with a set of microbenchmarks. We measure the runtime speed, as well as, the size of executables. Since WARDuino targets microcontrollers with limited memory, it is important to take into account the number of bytes that get flashed per program. We end the section by looking at WARDuino's conformity to the WebAssembly standard (Section 3.10.3).

3.10.1 Practical Application

Smart light applications are one of the most widely known and practically applied IoT applications. We investigate how well WARDuino performs for programming microcontrollers in practice by implementing a simple smart light application in AssemblyScript. Specifically, we connected an ESP device to a button, and an LED. The microcontroller will toggle the LED, when the button is pressed, or when it receives a certain MQTT message over the internet. To receive MQTT messages, it subscribes to the "LED" topic on an MQTT broker. There are two recognized MQTT payloads: "on" and "off".

Listing 3-9 shows the source code of the software running on the ESP. On the left side, we import the WARDuino primitives, and we define some constants and helper functions. On the right side, we have the main entry point of the program, starting at Line 31. The main function first sets the correct modes of the `LED` and `BUTTON` pins. Next, it connects to the Wi-Fi network and prints the local IP address of the device on success (Lines 36–39). When the microcontroller is connected to the network, it connects to the MQTT broker (Lines 39–47). In Section 3.4.1, we already discussed the code required to set up these connections.

With an established connection, the microcontroller subscribes to the "LED" MQTT topic on Line 47. The supplied callback is defined on Lines 14 to 24. It takes two arguments, the topic and the payload of the incoming message. First, it prints the message to the serial port using `print`. Then, we inspect the payload, if it is the string "on", we turn the LED on by using `digitalWrite`, otherwise we turn the LED off.

After subscribing, the `main` function sends an "on" message to the "LED" topic using the `MQTT.publish` primitive. When the device receives its own message, the `callback` function will make the LED shine.

```
 1  import * from "as-warduino";
 2
 3  const BUTTON = 25; const LED = 26;
 4  const SSID = "local-network";
 5  const PASSWORD = "network-password";
 6  const CLIENT_ID = "random-client-id";
 7
 8  function until(attempt: () => void,
 9                 done: () => boolean): void {
10    while (!done()) {
11      delay(1000); attempt();
12  }}
13
14  function callback(topic: string,
15                    payload: string): void {
16    print("Message [" + topic + "] " + payload);
17
18    // Inspect the payload of the MQTT message
19    if (payload.includes("on")) {
20      digitalWrite(LED, PinVoltage.HIGH);  // On
21    } else {
22      digitalWrite(LED, PinVoltage.LOW);   // Off
23    }
24  }
25
26  function toggleLED(_t: string, _p: string): void {
27      let status = digitalRead(LED);
28      // Toggle LED via MQTT
29      MQTT.publish("LED", status ? "off" : "on");
30  }
31  export function main(): void {
32    pinMode(LED, PinMode.OUTPUT);
33    pinMode(BUTTON, PinMode.INPUT);
34
35    // Connect to Wi-Fi
36    until(() => { WiFi.connect(SSID, PASSWORD); },
37      WiFi.connected);
38    let message = "Connected to wifi with ip: ";
39    print(message.concat(WiFi.localip()));
40
41    // Connect to MQTT broker
42    MQTT.init("192.168.0.42", 1883);
43    until(() => { MQTT.connect(CLIENT_ID); },
44      MQTT.connected);
45
46    // Subscribe to MQTT topic and turn on LED
47    MQTT.subscribe("LED", callback);
48    MQTT.publish("LED", "on");
49
50    // Subscribe to button interrupt
51    interruptOn(BUTTON, InterruptMode.RISING,
52      toggleLED);
53
54    while (true) {
55      until(() => { MQTT.connect(CLIENT_ID); },
56        MQTT.connected);
57      MQTT.poll();
58      delay(500); // Sleep for 0.5 seconds
59    }
60  }
```

Listing 3-9. A smart light AssemblyScript program for WARDuino.

On Line 47 we attach a callback to rising voltage changes of the button pin. We use the `interruptOn` primitive to do this. It takes three arguments: the pin to monitor, the kind of change to trigger for, and a callback to invoke when a change occurs. Here we monitor the pin of the button for a rising edge (`InterruptMode.RISING`). This means our callback, `callback`, will be invoked whenever the `BUTTON` pin goes from low (not pressed) to high (pressed). Lines 26 to 30 define `toggleLED`. It reads the current state of the LED and then sends out an MQTT message with the opposite state. This message will then be received by `callback`, which in turn toggles the LED's state.

The `main` function concludes by ensuring that the connection to the MQTT broker stays alive. To this end, it uses a `while` loop that calls `until` to reconnect to the MQTT broker if the connection is lost.

Note that the two callbacks used in this example, `callback` and `toggleLED`, have different types. However, in Section 3.8.5 we saw that our callback system requires that all stored callbacks have the type $\mathsf{i32} \times \mathsf{i32} \times \mathsf{i32} \times \mathsf{i32} \rightarrow \varepsilon$. This is indeed the case at the WebAssembly level. The primitive behind `interruptOn` requires a callback of that type. Our language interoperability layer abstracts this away and exposes an `interruptOn` that expects a `void` → `void` AssemblyScript callback.

To test the stability of WARDuino, we run the code in Listing 3-9 on an ESP32-DevKitC V4 board with WARDuino. We also create a small web application to control the LED from our phone via MQTT. When testing our setup, we encountered no noticeable delay between pressing the physical button, and the LED changing status. Furthermore, the delay between pressing the button on the web page, and the LED updating was reasonable and mostly influenced by the Wi-Fi connection.

3.10.2 Performance on Microcontrollers

There are three ways in which developers can run programs on microcontrollers. Dynamically typed languages such as JavaScript are run in dynamic runtimes, while statically typed languages can be executed with a byte-code interpreter, as is the case for WebAssembly, or can be compiled to executable byte-code, typically done with C or C++. In this section we compare the general computational performance of the WARDuino virtual machine with each approach. For the dynamic language we used the popular Espruino (Williams, 2014) runtime for JavaScript. For the static runtime we compared WARDuino with another WebAssembly byte-code interpreter that is small enough to run on microcontrollers, namely WASM3 (Massey and Shymanskyy, 2021). Since it is still the most widely used language for microcontrollers, we used C as the compiled language. We use each approach to run the same microbenchmarks on a microcontroller. Since we are interested in comparing the general computational performance and memory occupancy of our approach, our benchmarks consist of standard computational tasks; such as calculating the greatest common divider, factorial, binomials, Fibonacci sequence, or verifying if a number is prime.

Espruino (Williams, 2014) is a commercial JavaScript based microcontroller platform for IoT applications. Like WARDuino, Espruino is a VM that runs on microcontrollers. Instead of running WebAssembly, it interprets JavaScript, a popular programming language. The pins of the device are exposed as global JavaScript objects with methods for adjusting their value, `D14.set()` for example makes pin D14 high. Other features, such as Wi-Fi connectivity, can be imported and present themselves as JavaScript objects as well.

WASM3 (Massey and Shymanskyy, 2021) is a fast WebAssembly interpreter, which uses a special compilation technique rather than JIT compilation (Aycock, 2003) to achieve good performance. It has explicit support for microcontrollers such as the ESP32 (Espressif Systems, 2023b). Similar to WebAssembly it exposes access to the hardware of the microcontroller through a custom WebAssembly module that provides Arduino primitives. Our benchmark consists of six computationally intensive programs implemented in JavaScript (for Espruino), WebAssembly (for WARDuino and WASM3), and C (as baseline) . The WebAssembly code was generated from C code with Clang 13 (Clang contributors, 2021). To ensure an honest comparison this C code is identical in structure to the JavaScript code except for the addition of types. Additionally, we prohibited the compiler to perform loop unrolling and inlining of the benchmark functions. Appendix G.1 describes our microbenchmark functions in detail. Each solves some mathematical problem in a naive way.

We compare the performance of the runtimes in terms of execution speed and program size for each microbenchmark. When measuring the execution speed, we record the execution time of the benchmarks excluding the upload and initialization time of the virtual machines. We include the program size, because the low-end microcontrollers we are targeting have very limited memory. In fact, the measurements were performed on an ESP32-DevKitC V4 board[3]. This board features an ESP32 WROVER IE chip that operates at 240 MHz, with 520 KiB SRAM, 4 MB SPI flash and 8 MB PSRAM. This is a representative board for the kind of resource-constrained microcontrollers targeted by WARDuino. There exist more resource-rich devices that are used for IoT applications, such as the Raspberry Pi devices, but these are so powerful that many of the challenges outlined in this chapter are present to a far lesser extend. For example, as a Raspberry Pi has a full-fledged operating system, it is trivial to adapt the code remotely (with ssh).

3.10.2.1 Espruino

Figure 3-10 shows the results of the benchmarks for Espruino and WARDuino. In each graph the green (right) bars indicate the measurements for

[3]https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html

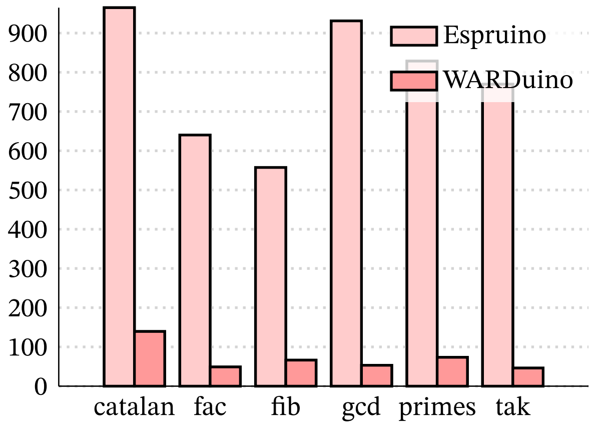


Figure 3-10. The execution times of WARDuino and Espruino. *Top Left*: absolute execution times for the benchmarks. *Top Right*: sizes of the programs uploaded to the VM. *Bottom Left*: execution time normalized to native C execution time. *Bottom Right* execution times normalized to the WARDuino execution time.

Espruino, the red (left) bars show the results for WARDuino. The first graph, on the top left, shows the absolute execution times of each benchmark on a log scale. The overhead of the WARDuino and Espruino implementations compared to execution time of a native C implementation are shown in the graphs at the bottom. We see that WARDuino consistently outperforms Espruino by roughly a factor of 10. In fact, the geometric mean of the overhead relative to WARDuino is 11.66. Note that the difference is even larger for the `tak` benchmark. This may be attributed to the extreme amount of recursion the `tak` function exhibits. This suspicion seems to be confirmed by the (iterative) `fib` benchmark that calculates Fibonacci numbers without recursing. In this benchmark the performance difference is indeed less pronounced as in the `tak` benchmark.

In the last graph (top right) we show the byte code sizes uploaded to the instantiated virtual machines. We see that the WARDuino size is never larger than the JavaScript files. This is not surprising, as WebAssembly programs are saved in a binary format, and were optimized for size by the compiler.

3.10.2.2 WASM3

Figure 3-11 compares the performance of WASM3 and WARDuino. The graph on the right side shows the overhead of the WARDuino virtual machine relative to the WASM3 runtime. We see that WASM3 executes the same WebAssembly program approximately forty times faster than WARDuino, to be precise, the geometric mean of WARDuino's overhead compared to WASM3 is 40.75.

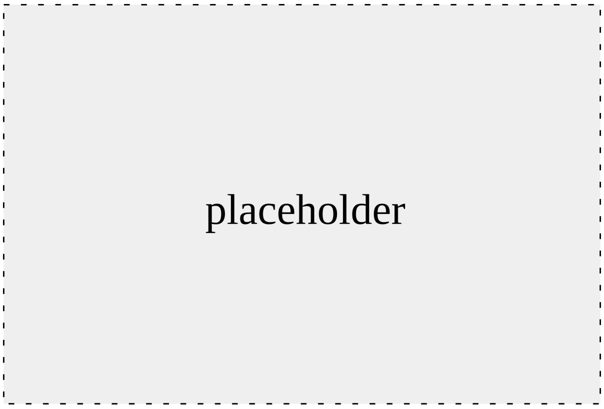


Figure 3-11. The benchmark execution times of WARDuino and WASM3. *Left*: absolute execution times. *Right*: execution times normalized to WASM3 execution time.

name	Espruino (s)	WARDuino (s)	WASM3 (s)	C (s)	$\frac{\text{Espruino}}{C}$	$\frac{\text{WARDuino}}{C}$	$\frac{\text{WASM3}}{C}$
catalan	964.72	139.38	3.765	0.246	3922.54	566.71	15.31
fac	640.11	48.97	1.071	0.122	5248.21	401.49	8.78
fib	557.39	66.33	1.481	0.134	4145.68	493.35	11.02
gcd	931.1	53.21	1.157	0.224	4153.24	237.34	5.16
primes	828.54	73.48	2.279	0.137	6058.18	537.29	16.66
tak	769.6	46.14	—	0.107	7217.92	432.7	—
mean	767.476	65.869	1.736	0.154	4993.28	428.542	10.495

Table 3-7. Left: Absolute execution times in seconds for all tests. Right: Execution time of tests normalized to the native C implementation. The means shown in the table are geometric means.

Although WASM3 is faster than WARDuino, the interpreter's architecture, comes with a significant drawback on memory-constrained devices. It trades memory space for time. Our texttt{tak} benchmark cannot run on the ESP32 with WASM3 because the device runs out of memory. In contrast, this benchmarks runs well on WARDuino. We excluded the texttt{tak} benchmark from the second graph in figure ref{fig:wasm3} for this reason. Note that the WebAssembly program implementing texttt{tak} does run on the same device with WARDuino.

3.10.2.3 Comparison

The complete benchmarks results are shown in table ref{tbl:allbench}. In the first four columns of the table, we report the time that elapses between starting and ending the execution of the microbenchmark ten times for each platform. On the right, we list the execution times normalized to the execution time of the native C implementation. Because the C implementation does not run in a managed environment, it is much faster. To add two numbers for example, no stack access is needed in native C. Since WebAssembly is a stack machine, and our implementation does not yet feature a JIT compiler, memory access is required to perform all basic operations. Taking the geometric mean of the normalized execution times, shows that WASM3 is 11 times slower than its native C, while WARDuino is about 428 times slower and Espruino is 4992 times slower. We note that clang was instructed to optimize for size. Setting the compiler to optimize more at the cost of binary size can have a big impact on the performance of WARDuino at the price of binary size code. Not optimizing for binary size reduces WARDuino's overhead compared to C to 312x.

3.10.3 Conformance to the WebAssembly Standard

The WebAssembly working group provides a test suite for the core WebAssembly semantics[4]. These integration tests are meant to help runtime implementers verify that their implementation follows the official specifications. Each test contains a WebAssembly module to be loaded by the virtual machine, and a sequence of assertions to check. These assertions specify an action to execute on the module, and the expected result.

We used the official specification test suites to test the WARDuino virtual machine extensively. Because WARDuino does no yet support the latest extending proposals, we use the 15295 tests of the latest specification test suite that only test the original core specification. For instance, we leave out the tests for SIMD instructions, since this proposal has not been adopted by the WARDuino virtual machine. Besides the official specifications, we wrote our own specification test for the WARDuino primitives and extension. Analogous to the official tests, we use these tests to verify that our primitives do not cause ill-formed stacks and only throw traps under the right conditions.

3.10.4 Discussion

The computational benchmarks in this section show that WARDuino is roughly ten times faster than the popular Espruino virtual machine. While IoT applications typically do not perform many computationally heavy tasks, we believe that the difference in performance for these benchmarks is significantly large enough to show that WARDuino—and WebAssembly generally —can easily outperform dynamic interpreters for high-level languages. At

[4]github.com/WebAssembly/spec/tree/main/test/core

the very least, we may conclude that WARDuino is certainly fast enough for real-world IoT applications, such as those run with Espruino. This is further illustrated by the smart light application at the beginning of this section, which shows that WARDuino can indeed be used to program embedded devices with AssemblyScript.

The microbenchmarks also show that WARDuino executes programs significantly slower than their native counterparts. The extra execution time allows us to provide the developer with the safety guarantees of WebAssembly and features such as remote debugging and over-the-air updates. Measurements of the WASM3 virtual machine show that WebAssembly program can run faster in WASM3 on microcontrollers than in WARDuino. While WARDuino has a significant overhead in speed compared to WASM3, it does manage to run with a lower memory footprint, as WASM3 is not able to run all our benchmarks without exceeding the memory limits of our microcontroller. Additionally, WASM3 also does not enable remote debugging and over-the-air updates. Nevertheless, we believe that we could use techniques from WASM3 to further improve WARDuino's performance.

Aside from performance, we have shown that WARDuino conforms to most of the core WebAssembly specification.

3.11 Related Work

WARDuino presents a WebAssembly virtual machine for microcontrollers and a collection of extensions to the WebAssembly standard. In this section we discuss the related work for each aspect in turn. We focus first on programming microcontrollers with non-WebAssembly solutions. Then we discuss other WebAssembly embeddings for microcontrollers. After this, we finish our related work by summarizing the alternative methods for handling interrupts in WebAssembly.

3.11.1 Programming Embedded Devices

The world of programming languages for microcontrollers is heavily dominated by the C language (Kernighan and Ritchie, 1989), but an increasing range of programming languages have been ported to various hardware platforms, such as: Forth (Rather and Moore, 1976), BASIC (Kemeny et al., 1968), Java (Gosling et al., 1996), Python (Rossum, 1995), Lua (Ierusalimschy et al., 1996) and Scheme (Yvon and Feeley, 2021). Here we restrict ourselves to compare popular implementation approaches for IoT functionality on ESP-based microcontrollers, the platform on which WARDuino was primarily tested.

The predominant programming language for programming the ESP processor is C (Espressif Systems, 2023a, Kernighan and Ritchie, 1989). The advantage of using C is that the programs execute fast. The downside is that it places the burden of managing memory onto the developer. Another downside of the C language is that once a bug is potentially solved, the programmer needs to re-compile, flash the hardware and restart the device completely. Flashing the chip can take a long time, making the development of microcontroller software a rather slow process.

In recent years the concept of remote debugging has seen its first implementations for embedded systems. The recently released Arduino IDE 2.0 comes with a debugger interface that allows developers to debug C and C++ code with standard debugging operations. It does not support any over-the-air updates (Söderby and De Feo, 2024). Subsequently, developers still need to flash the entire software at every change. In contrast, WARDuino allows both remote debugging and over-the-air updates to ease program development on ESP processors.

The Zerynth Virtual Machine (Zerynth s.r.l., 2021) allows developers to run Python programs on 32-bit microcontrollers, but it mainly targets the ESP platform. Users can send HTTP and MQTT request by using the Zerynth standard library. Like our work, these network primitives are implemented in C and exposed in a (Python) module. Zerynth only supports Python, whereas WARDuino aims to build a common WebAssembly based intermediate representation that allows a multitude of languages to use the networking capabilities of the embedded device. Additionally, WARDuino supports remote debugging with breakpoints, a capability the Zerynth VM does not offer.

Espruino (Williams, 2014) allows programmers to use a dialect of JavaScript by running a JavaScript interpreter on the chip. The VM is unfortunately too slow to program the device drivers in JavaScript. Therefore, most support for displays and sensors is hard-coded in the Espruino VM. Espruino has MQTT and HTTP modules that can be used in the traditional callback-based style of JavaScript. The VM offers both a web IDE, and a command-line tool to program microcontrollers. Both applications offer roughly the same functionalities, and can connect to a remote device over many connection types, such as serial, Wi-Fi, or Bluetooth. Once connected to a device, Espruino can provide the developer with a REPL to execute JavaScript code directly on the device. This way Espruino does support over-the-air updates. The Espruino runtime contains a built-in remote debugger, which uses the same commands as GDB.

MicroPython (George, 2021) is a highly optimized subset of the Python programming language. It provides on the chip compilation of Python programs. MicroPython supports HTTP requests through its `urequests` module and

MQTT with the `micropython-mqtt` community package. The MicroPython project does not provide any means for remote debugging itself, but does offer a REPL in the browser that can connect with embedded devices over serial or Wi-Fi (George, 2021). However, there are a few integrated development environments for Python that can use MicroPython, such as the Mu (Tollervey, 2022) and Thonny (Annamaa, 2015) editors, which do support minimal remote debuggers. Unfortunately, both debuggers only supports larger Raspberry Pi devices, and do not appear to support smaller microcontrollers targeted by WARDuino.

There are multiple projects for using Ruby on embedded devices, the most widely used and actively maintained is mruby (Yukihiro and others, 2023). The mruby project partially implements the ISO standard for the Ruby language. Unfortunately, mruby does not support a remote debugger for embedded devices and developers are forced to rely on print-statement debugging. The project does include its own package manager, which gives developers access to a variety of libraries for accessing hardware and using IoT protocols (Koji and others, 2023, McDonald and others, 2023, Yukihiro and others, 2023). However, most libraries are open-source projects, and given the small community, many libraries are no longer being actively maintained.

3.11.2 Over-the-air Programming

The high-level languages described so far have varying support for over-the-air updates, mostly in the form of remote REPLs. However, the idea of updating low-end embedded devices over-the-air is not new, and the idea has received considerable attention in the context of sensor networks. For instance, already in 2002 Levis and Culler (2002) created a byte-code interpreter for tiny microcontrollers called Maté. Maté was designed to reprogram sensor networks through self-replicating packages of just 24 instructions. More recently, Baccelli et al. (2018) looked at reprogramming low-end devices with a low-code approach, where Business Process Modelling Notation (BPMN) (Rospocher et al., 2014) is translated into JavaScript code by a central server and sent to the devices of the sensor network over the air. Similar to a lot of systems for over-the-air updates, in this work the software running on the low-end device is updated in its entirety. The functional approach was also explored by Lubbers et al. (2021) in the Clean language (Brus et al., 1987), specifically, task oriented programming was adopted for tiny low-end microcontrollers. Task oriented programming is a programming paradigm for distributed systems, where tasks represent units of computations, which —like monads—can be constructed with combinators, and which share data via their observable values (Plasmeijer et al., 2012). Individual tasks can be compiled to bytecode and sent to devices to be executed, enabling partial updates of the code. While these three approaches are very different, each focuses on low-end microcontrollers similar to WARDuino. By contrast de Troyer et al. (2018), developed a reactive programming approach for the more powerful Raspberry Pi computers. Raspberry Pi's are far bigger than the low-end devices targeted by WARDuino, and have subsequently much more resources, but they are still used considerably for the Internet of Things (Maksimovic et al., 2014). The reactive language allows the entire life-cycle of a device to be programmed, including the deployment of software and over-the-air updates. Again, the over-the-air updates are limited to the entire program.

In contrast to these works, the main motivation behind WARDuino is to simplify development of IoT applications in a way that is widely applicable. In this spirit, the idea of over-the-air updates is adopted by WARDuino as an extension to the classic debugging operations. This provides developers with powerful operations during debugging; partial code updates, and changing variable values. Additionally, we believe the small-step semantics of the over-the-air updates present a novel contribution, which in future work can form the basis for proving the correctness of updates by showing that programs remain well-typed.

While we are not aware of any other attempts to describe over-the-air updates of binary code through a small-step semantic, there is some theoretical work on live updates. For instance, in 1996, Gupta et al. (1996) showed that the validity of live updates is generally undecidable. However, most work has been focused on distributed systems specifically, and the issues that arise due to the distribution of nodes. identified the important problem of; when is a system in the appropriate state for a live update? The proposed solution was later improved by Vandewoude et al. (2007). In WARDuino this problem is largely circumvented because the updates are integrated in the debugger.

3.11.3 WebAssembly on Embedded Devices

Since the start of the WARDuino project, many others have started looking into running WebAssembly on embedded devices. These projects range widely in scope and focus. Here, we give an overview of some projects bringing WebAssembly to IoT and Edge Computing.

The WebAssembly Micro Runtime (Huang and Wang Xin, 2021) and WASM3 (Massey and Shymanskyy, 2021) are WebAssembly runtimes with a small memory footprint like WARDuino. The WebAssembly Micro Runtime specifically aims to have a tiny memory footprint such that it can be used in constraint environments, such as small embedded devices. The runtime largely supports the WASI standard (Hickey et al., 2020), including the `pthreads` API that allows developers to use multithreading. However, it does not support the WASI `sockets` API providing internet connection. WASM3 can run on microcontroller platforms, such as the ESP32. In the first place, the microcontroller support is a research project to test and showcase their novel interpreter that uses heavy tail-call optimizations rather than JIT compilation to improve performance (Massey and Shymanskyy, 2022). Not using JIT compilation is the main reason the WASM3 interpreter has such a small footprint and can run on microcontrollers. WASM3 supports most of the new WebAssembly proposals and can run many WASI apps, but it does not fully support the `pthreads` or `sockets` API. WARDuino brings a more general mechanism to WebAssembly that allows both synchronous and asynchronous network communication without the need for a full-fledged operating system. Unlike the WebAssembly Micro Runtime, WASM3 has explored remote debugging. Specifically, the project examined the remote debugging protocol of GDB to try and debug source-level WebAssembly (Shymanskyy, 2023). This effort was not targeted at microcontrollers, but could work on embedded devices with a JTAG hardware debugger. By contrast, WARDuino can remotely debug microcontrollers without the need for a dedicated hardware debugger. Additionally, WARDuino supports over-the-air updates, something neither the WebAssembly Micro Runtime nor WASM3 allow.

Wasmer (Wasmer, Inc., 2022) is another WebAssembly runtime that reports to be fast and small enough to run on Cloud, Edge and IoT devices. The runtime supports WASI programs, but it does not support threading and is waiting for the official Threads Proposal for WebAssembly to reach the implementation phase, which it has not at the time of writing. However, it does support Emscripten's pthread API. Unfortunately, the project does not provide a list of supported microcontroller platforms and does not seem to target devices with limited memory. Neither does the project provide clear instructions on how to execute the Wasmer runtime on embedded devices. While the project developed their own WebAssembly package manager (wapm), there are currently no packages for IoT protocols such as MQTT, or for interacting with hardware peripherals. Wasmer is currently working on its debugging support, which is limited at the time of writing. Moreover, the project does not seem to target remote debugging of embedded devices at this stage.

3.11.4 Interrupt Handling in WebAssembly

There are different efforts in the WebAssembly community to add support for handling asynchronous to the standard. As WebAssembly is still primarily used on the web, most of the new proposals to the standard are made because of certain needs arising from the web. The WASI `pthreads` API, The threads and stack switching proposals (WebAssembly Community Group, 2022) are no exception.

The threads and stack switching proposals allow WebAssembly to run asynchronous code, this could then be used to add interrupts to WebAssembly. These proposals themselves do not provide a dedicated system for interrupts. Developers would have to implement a complete callback handling system in WebAssembly themselves. Without a dedicated system for interrupts, everything would have to be implemented directly into WebAssembly, which is not a trivial task. Additionally, both proposals allow the space taken by the stack(s) to grow fast, an unwanted side effect on memory constrained devices. WARDuino only executes one callback at a time keeping the stack size as low as possible.

A very recent chapter by Phipps-Costin et al. (2023), alternatively proposes a universal target for non-local control flow that relies on effect handlers (Plotkin and Pretnar, 2009). In our opinion this solution is more attractive than the threads and stack switching proposal, primarily due to its simplicity —it only adds three new instructions—and its universality. Similar to the stack switching proposal, a callback handling system comparable to the one described here, could most likely be built on top of this system. However, continuations are still expensive, since they also need to save the entire stack. Furthermore, the proposal is again not enough to support asynchronous primitives. The effect handlers would only allow us to create a system for handling interrupts with callback functions, directly in WebAssembly code. In this case, we arrive at the same solution we have outlined in this chapter, except the implementation has moved from the virtual machine to WebAssembly code. It is not clear whether this approach would have any benefits.

The WebAssembly System Interface (WASI) (Hickey et al., 2020) is a collection of standardized APIs for system level interfaces. It is not part of the official WebAssembly standard, but is widely used. The WASI `pthreads` API could be used to implement interrupts in WebAssembly. WASI provides a means to access system level APIs in WebAssembly. As with our approach, these API functions can be imported from a module named `env`. The expectation is that the WASI `pthreads` API could be used to implement a callback handling system. Again, this is currently only an idea, as we are not aware of anyone actually realizing such an implementation. Building a callback handling system on top of WASI already has its own challenges, but using it on embedded systems adds an additional layer of constraints. As a start, simply supporting the full WASI specification on embedded devices has proven to be complicated in practice (Massey and Shymanskyy, 2021). To the best of our knowledge there is no WebAssembly runtime for constrained devices that fully supports WASI. Moreover, this approach does not seem to have much traction in the community, as there are several online discussion threads to add a more dedicated API for asynchronous interrupts to WASI[5]. However, not much work has been done around these discussions, and it seems WASI is waiting on the official proposals before they create their own API.

3.12 Conclusion

This chapter presents the design and implementation of WARDuino that addresses key challenges associated with developing IoT applications: *low-level coding*, *portability*, *slow development cycle*, *debuggability*, *hardware limitations*, and *bare-metal execution environment*. The WARDuino virtual machine enables programmers to develop IoT applications for microcontrollers in high-level languages—compiled to WebAssembly—rather than low-level languages such as C. Higher-level languages can help developers by providing automatic memory management and by giving extra guarantees via type systems. Additionally, using a universal compile-target such as WebAssembly, WARDuino can greatly improve the *portability* of microcontroller programs. The virtual machine supports the WebAssembly core specification and several important extensions to support common aspects of IoT applications.

Access to device peripherals and common M2M protocols is provided by WebAssembly primitives embedded in the virtual machine. These primitives include functions for synchronous (HTTP) and asynchronous (MQTT) communication protocols. To support asynchronous code, WARDuino allows developers to assign callback functions as handlers for asynchronous events, such as incoming MQTT messages or button presses. Whenever a subscribed event occurs, WARDuino will transparently execute the callbacks in isolation of the running program as shown by the small step reduction rules for the callback handling system.

Language integration for high-level languages, exposes the WARDuino primitives as a library with an interface that is conventional for the host language. We have presented different levels of integration for the AssemblyScript language, with higher integration bringing more of the advantages of high-level coding to WARDuino. Our AssemblyScript library, for example,

[5]In particular the “Alternative to a “conventional” poll api?” (Issue 79) discussion is interesting here, it describes almost exactly the use-case our callback handling system enables. Other relevant discussions are “Execution Environment for Asyncify Lightweight Synchronize System Calls” (Issue 276) and “Poll + Callbacks” (Issue 283). These discussions can be found on https://github.com/WebAssembly/WASI/issues.

exposes primitives that accept strings although WebAssembly does not have a *string* type. Internally, our library translates the AssemblyScript strings to WebAssembly memory slices. Developers can thus use WARDuino without having to worry about these kinds of implementation details, or deal with the headaches of *low-level coding*.

Another important contribution is the improved *debuggability* of microcontrollers provided by the WARDuino remote debugger. Developers can send debug messages over any communication channel to the virtual machine and mandate it to pause, resume, step or dump its state. In the paused state, developers can use the same mechanism to reprogram a running application. WARDuino can update local variables, functions, and even the entire program over the air. This speeds up the *slow development cycle*, as developers no longer need to wait while their program is re-flashed to the device. To further ease debugging we created a VSCode plugin that allows remote debugging of a WARDuino instance in a graphical user interface. Thereby, creating a development experience which is much closer to conventional computer programming.

Uniquely, the debugging, over-the-air programming, and callback handling system have been described formally as extensions to the operational semantics of WebAssembly. The small-step reduction rules provide a precise description of these systems, and allow them to be easily implemented by other WebAssembly virtual machines. Furthermore, the semantics allow us to prove desirable properties over the debugger and callback handling system. We prove that the debugging semantics are observationally equivalent to the underlying WebAssembly semantics. In future work, we want to explore other desirable properties, for example that the over-the-air updates cannot break a well-typed WebAssembly program.

We evaluate our work by demonstrating that it is suitable and stable enough to program traditional long-running IoT applications with a smart lamp application in AssemblyScript, a snake game and a whole suite of microbenchmarks. Additionally, we compare WARDuino's performance to that of WASM3 and Espruino. We conclude that we are on average 428 times slower than a native C implementation of computationally intensive microbenchmark. By comparison, WASM3 is 10 times slower and Espruino is 4.991 times slower. Although performance improvements are likely possible, we believe that WARDuino is fast enough for IoT applications as the much slower Espruino is widely and successfully used for this goal.

Chapter 4

Stateful Out-of-place debugging

Some problems are better evaded than solved.
— Tony Hoare

Today, remote debuggers—like the one presented in the previous chapter —are commonly used to debug microcontrollers, however, there are severe disadvantages. Luckily, a novel technique, called out-of-place debugging, can be adopted to largely evade these disadvantages by moving the debugging session to another more powerful device.

During the writing of this dissertation we explored two new concepts for out-of-place debugging, which are essential for microcontrollers. Initially, we explored how to support event-driven applications, which are common in microcontrollers. This lead to an early publication at MPLR 2022 (Lauwaerts et al., 2022). Subsequently, we explored how to support stateful actions on non-transferable resources, such as memory-mapped I/O devices. As part of this work, we developed the first formal model for out-of-place debugging, and proved its soundness and completeness.

4.1 Introduction

Remote debuggers are commonly used to debug various kinds of applications (Högl and Rath, 2006, Li et al., 2009), such as real-time systems (Skvařc Bŏzĭc et al., 2024), containerized applications for edge computing (Ozcan et al., 2019), and Internet of Things applications (Lauwaerts et al., 2024, Pötsch et al., 2017). Yet, remote debuggers suffer from three severe disadvantages. Firstly, the debugger is run on the remote device. In the context of constrained devices, this additionally limits the resources available to the debugger. Secondly, the communication channel can be slow, and can introduce latency in the debugging process. Thirdly, the delays introduced by the remote communication can exasperate debugging interference.

These problems can be addressed using out-of-place debugging. It combines local online and remote online debugging, reducing communication latency and overcoming the resource constraints of the remote device. However, out-of-place debugging is a very new idea, and there are still several open questions and challenges that come with the technique, that have not been addressed yet. Additionally, the technique is without formal foundations. In this chapter, we present the first formalisation of the technique, and attempt to address some important gaps in the existing literature.

Naturally, our work builds on the preceding out-of-place debugging works, and these deserve a proper introduction. Therefore, we first provide an overview of how out-of-place debugging works, and discuss how our contributions relate to previous work.

4.1.1 The origin of out-of-place debugging

Out-of-place debugging was originally devised to minimize debugging interference of remote debuggers for big data applications (Marra et al., 2018), by moving the debugging session to another device. The first out-of-place debugger, IDRA, was developed for the Pharo language, and allowed for debugging of live distributed big data applications. By moving the debugging session out of place, IDRA could debug a node in the network without effecting the live execution of the distributed software. The prototype showed how out-of-place debugging can reduce the debugging latency significantly in the context of large clusters.

4.1.2 Out-of-place debugging for microcontrollers

In the context of embedded applications, out-of-place debugging has great potential for improving the debugging experience offered by remote debuggers. This is due to the techninque being ideally suited for tackling the three main drawbacks of remote debuggers. Firstly, the debugger is run on the remote device. In the context of constrained devices, this additionally limits the resources available to the debugger. Secondly, the communication channel can be slow, and can introduce latency in the debugging process. Thirdly, the delays introduced by the remote communication can exasperate the debugging interference.

An initial investigation by Rojas Castillo et al. (2021) looked at out-of-place debugging as a solution for live debugging of *in-production* embedded applications. The work paved the way for using out-of-place debugging on microcontrollers, and while its topic is very interesting, there are many questions around the idea of debugging in production. In-production debugging is rarely seen in practice, and considered by some to be undesirable. Regardless, out-of-place debugging can provide numerous other benefits to debuggers for microcontrollers. In this dissertation, we will therefore not concern ourselves with the problem of in-production debugging, and instead present how we adapted—and extended—out-of-place debugging to work for microcontrollers during the traditional development stage.

4.1.3 The gaps in out-of-place debugging

Since out-of-place debugging is still a very young technique, it is not surprising that there are some important gaps in the existing work around it. There are three important gaps that we attempt to fill in this chapter. While these gaps are not specific to microcontrollers, they are especially relevant in this context.

First, out-of-place debugging currently lacks a sound formal foundation. We therefore developed the first formalisation of the technique based on WebAssembly. However, our formalisation illustrates and captures the essence of out-of-place debugging without many WebAssembly specifics—and so we argue, is more broadly applicable.

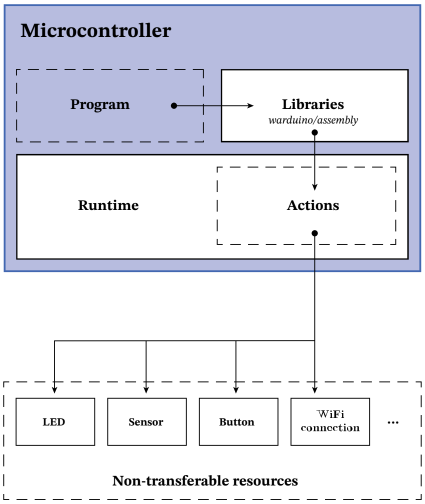
Second, existing work fails to address possible state desynchronization between the remote and local device. Existing solutions typically limit internal state changes to the local debugging environment, making it difficult to debug essential operations like MQTT communication in Internet of Things (IoT) systems. In our formalisation, we show how to handle stateful operations on non-transferable resources through limited synchronization of the state between the local and remote devices.

Third, non-transferable resources are only accessed in a synchronous request-driven way. The client debugger will request information about non-transferable resources from the server, and *pull*, or *transfer*, the information to the local client session. However, some non-transferable resources may act in an asynchronous way, *pushing* information at arbitrary times. To solve this, we extend out-of-place debugging to support *event-driven* access to non-transferable resources, alongside the typical *request-driven* access.

In the original publication (Lauwaerts et al., 2022), we used the terms *pull* and *push*, here we use *request-driven* and *event-driven* instead.

4.2 Background: Out-of-place debugging

Before delving into the details of our contributions, we first provide an overview of how out-of-place debugging works, and discuss the general out-of-place debugger architecture. Out-of-place debugging provides the debugging experience of a remote debugger, while running most of the code on a local device, thereby reducing debugging latency and interference. It allows for debugging live applications, as the debugging session is isolated from the live execution of the program. Additionally, by running the debugging session out-of-place, the debugger can have access to more computational power and



```
import {pinMode, PinMode,
  Voltage, digitalWrite, delay}
  from "warduino/assembly";

export function main(): void {
  const led: u32 = 26;
  const pause: u32 = 1000;

  pinMode(led, PinMode.OUTPUT);

  while (true) {
    digitalWrite(led, Voltage.HIGH);
    delay(pause);
    digitalWrite(led, Voltage.LOW);
    delay(pause);
  }
}
```

Figure 4-12. Typical blinking LED program for microcontrollers, illustrating non-transferable resources in out-of-place debugging. *Left:* A schematic of the microcontroller. *Right:* The AssemblyScript code for the program.

memory, or other resources, enabling more complex debugging techniques. We will illustrate the various concepts involved using our prototype implementation build on top of the WARDuino (Lauwaerts et al., 2024) virtual machine.

4.2.1 Example: a blinking LED

Figure 4-12 shows the typical blinking LED example for microcontrollers in AssemblyScript. The application uses the WARDuino actions, imported on the first line of the program. After the correct mode for the LED's pin has been set, the program will turn it on and off in an infinite loop with a small delay. The left side of the figure shows a schematic representing the setup of the microcontroller. The AssemblyScript program is compiled to WebAssembly and run on the microcontroller using a WebAssembly runtime. The runtime provides a series of functions to access the non-transferable resources of the microcontroller, such as the LED, buttons, and sensors—we call these functions *actions*.

4.2.2 Debugging with Out-of-place debugging

A developer can use an out-of-place debugger to debug the example application locally on their own machine, while still maintaining the effects on the

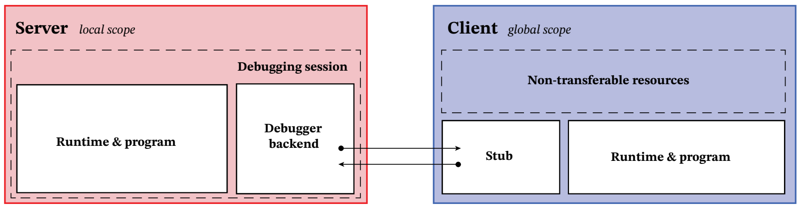


Figure 4-13. Schematic showing the concept of out-of-place debugging with all the involved components.

remote microcontroller, in this case the LED can still turn on or off. Often microcontrollers do not have enough memory to run an additional debugger alongside the application. By using out-of-place debugging, this is no longer necessary. The microcontroller only needs to run a minimal stub to receive a handful of debugging instructions to instrument the runtime.

Figure 4-13 shows the components involved in out-of-place debugging, the developer's local *client* on the left, while the right side shows the remote *server*. The remote server is the device where the software is intended to run. In the case of the blinking light application, this would be the microcontroller that controls the LED. Uniquely in out-of-place debugging, the entire debugging session—consisting of the runtime and the program being debugged —lives on the client.

Despite their small size, we refer to the microcontrollers as the *server*, because they *serve* the *requests* for information from the local debugger.

Note that the server may possess *non-transferable* resources, such as the LED in the example, which cannot be relocated along with the runtime and program to the client. We differentiate between two types of non-transferable resources—based on the way they are accessed or produce information—*synchronous* and *asynchronous*. Synchronous resources, are those accessed by the program synchronously, such as the LED in the example. Asynchronous non-transferable resources on the other hand can produce data at any point in the program, such as hardware interrupts for buttons or motion detectors.

To maintain the benefits of remote debugging, the client does not simulate the non-transferable resources. Instead, the server maintains a small stub which instruments its runtime, and can receive debug instructions from the debugger backend (client). Specifically, the stub supports direct access to synchronous non-transferable resources through remote function calls. For asynchronous non-transferable resources, the stub (server) can send messages to the client through the same connection.

In the case of our example, the only non-transferable resource is the LED light. We consider the action for controlling the LED stateless because it does not change the internal state of the runtime, and does not depend on any internal state other than its own arguments. Such stateless operations can

Reversible debugging can make external state inconsistent. We address this in Chapter 5.

still effect external state. However, since external state is part of the non-transferable resources it only exists on the remote server. As out-of-place debugging still accesses those resources through the server, we assume that their state remains consistent during debugging.

4.3 Problem statement

Since out-of-place debugging runs a program on a pair of two devices forming a distributed system, executing code can lead to diverging states between the two devices, thereby affecting the proper execution of the program. This can lead to inconsistent, and incorrect observations of the program's behavior, making it difficult to identify the root cause of a bug. This is even more problematic when part of the program's execution is asynchronous. To further clarify the problem of state desynchronization, we look at a use case of out-of-place debugging on an Internet of Things application for microcontrollers.

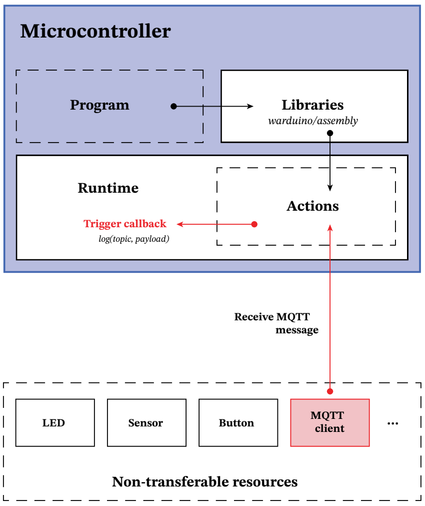
4.3.1 Example: asynchronous logging of a sensor

Consider the previous LED example, in an Internet of Things setting we would like to control the LED through some communication protocol such as MQTT. Figure 4-14 shows how this can be done in AssemblyScript code. The example is written for the WARDuino virtual machine, a WebAssembly runtime for microcontrollers. The virtual machine provides actions for the typical MQTT operations, such as subscribe, publish, and keep-alive.

The code in Figure 4-14 works as follows, at the start of the program the necessary MQTT configuration is set up, and the microcontroller connects to the local Wi-Fi network. After the connection is established, the microcontroller subscribes to the topic "SENSOR" and wait for incoming messages. At any point, a message can be received on the topic "SENSOR", at which point the virtual machine schedules the callback function and log the sensor value. On the right-hand side of Figure 4-14, we show a schematic of the microcontroller connected to the MQTT broker. The connection with the MQTT broker is an example of an asynchronous non-transferable resource, which can produce new MQTT messages at any point in the program, and subsequently triggering a callback, such as the *log* function in the example.

4.3.2 Out-of-place debugging for event-driven applications

Previous out-of-place debuggers (Marra et al., 2018, Rojas Castillo et al., 2021), did not support asynchronous non-transferable resources, such as the MQTT connection in the example. Access to non-transferable resources



```
import {MQTT} from "warduino/
assembly";
import {init} from "./util";
import * as config from "./config";

export function main(): void {
    // connect to WiFi & MQTT broker
    init();

    // Subscribe to MQTT topic
    MQTT.subscribe(
        "SENSOR",
        // and log value
        log(topic: string,
            payload: string) => {
            // ...
    });
    // ...
}
```

Figure 4-14. Small example application illustrating the state desynchronization problem in out-of-place debugging, when receiving MQTT messages. The application simply logs sensor values received via MQTT messages.

from the client was only possible by request of the server, such as through synchronous remote function calls in the case of Rojas Castillo et al. (2021). This already allowed for continual synchronous polling of a sensor value.

However, such a *request-driven* strategy to access sensor data is often too resource-intensive for embedded applications, and so does not capture all use cases for debugging IoT devices. Many of the peripheral devices attached to a microcontroller use an interrupt-driven interface instead. Such interrupts are generated when certain external events happen, for example when an input-pin changes from low to high. This prevents microcontrollers from having to poll the state of the pin constantly, and save resources by only reacting to changes in the environment when they occur.

4.3.3 Out-of-place debugging for stateful resources

Unfortunately in out-of-place debugging, since the entire debugging session is moved to the local machine, the scheduling of the callbacks happens on the local client, while subscribing callbacks on MQTT topics, and receiving the messages happens on the server. This presents two types of state desynchronization from the perspective of the client, *synchronous* and *asynchronous*.

4.3.3.1 State Desynchronization between two Devices

Harmful synchronous state desynchronization can occur whenever the client instructs the server to execute a piece of code. This code can potentially change the memory state on the server leading to state desynchronization. This is especially problematic when these state changes are needed for the program to continue running. In such cases, the server will use outdated values from memory instead of the updated values from the client.

While synchronous state desynchronization is triggered by the server and happens at specific, well-defined moments, asynchronous desynchronization can occur at any time. Relevant state on the client can change asynchronously, outside of the local context of the debugger. These changes are caused by asynchronous events which can occur at any time in the program, such as receiving MQTT messages.

The example in Figure 4-14, illustrates both synchronous and asynchronous state desynchronization. First, the MQTT.subscribe function, illustrates synchronous state desynchronization. It modifies the internal state of the runtime on the server by storing the callback to be triggered upon receiving the “SENSOR” messages, which the subsequent out-of-place code depends on. Second, whenever an MQTT message is received, this message is stored in memory (on the server) and should be executed as soon as the currently executing instruction is finished.

4.3.4 The abstract model of out-of-place debugging

The concept of out-of-place debugging still lacks a sound formal foundation, that captures the entire spectrum of its implementations. Furthermore, the existing work fails to address the full range of side effects of executing code involving non-transferable resources, and how it can lead to *state desynchronization* between the local and remote device. Existing solutions typically limit internal state changes to the local debugging environment, making it difficult to debug essential operations like MQTT communication in Internet of Things (IoT) systems.

Many non-transferable resources feature stateful operations, which can impact a program’s behavior. Without those changes being reflected on the local server, the debugger can never provide an accurate representation of the program’s execution. In scenarios where asynchronous events or interactions with external systems occur (for example receiving an MQTT message), desynchronization of the program state between the two devices can occur at any point in the program’s execution. Traditional methods that strictly scope side effects to the local server fail to account for the dynamic nature of state changes occurring on remote devices.

This chapter introduces stateful out-of-place debugging that bridges the gap between local and remote debugging paradigms more completely. Our method ensures that while the majority of the debugging code executes locally, stateful operations on the remote device are consistently managed and reflected on the local device. In our solution we adopt a minimal synchronization strategy, where synchronous operations transfer the minimal state required for their execution at the point they are invoked. Asynchronous resources send their changes to the internal state to the debugging session as soon as they become available, providing a debugging experience where debugger interference is minimized.

In order for this synchronization to work, our solution identifies the specific requirements non-transferable resources and their operations must satisfy. We demonstrate that meeting these requirements is not very restrictive and show how real-world examples can be implemented using our approach. We further provide proofs that this approach is sound and complete.

4.4 Stateful out-of-place Debugging for WebAssembly

In this section, we present the semantics of out-of-place debugging for WebAssembly. In order to have sound and complete semantics, we need to make two key assumptions about the stateful operations in the system. These assumption are not specific for WebAssembly and can be ensured for a wide range of stateful operations expressed in any programming language:

1. **Statically known state dependency.** Given the argument values for any *synchronous* stateful operation in the system, it must be possible to define a function which identifies all parts of the internal state that the operation depends on, and the state that the operation changes.

2. **Instantaneous partially ordered events.** We assume that for all *asynchronous* non-transferable operations during a debugging session, there exists a *partial* order over the asynchronous events they produce. In addition we assume that events do not have real-time dependencies.

Without the first requirement we would have to rely on possibly time consuming static analysis or conservatively copy the whole state, defeating many of the advantages of out-of-place debugging. This requirement does exclude certain complex operations, where changes to the state are calculated based on some implicit state. However, we believe that for most of such operations the implicit state can be made explicit by passing it to the stateful operation as an argument.

$$\begin{array}{lrl}
\text{(WebAssembly program state)} & K & ::= \{s, v^*, e^*\} \\
\text{(Action table)} & A & ::= a^* \\
\text{(Action)} & a & ::= \{\text{code cl}, \text{transfer } t, \text{transfer}^{-1}\ r\} \\
\text{(Backward transfer)} & t & ::= v^* \times s \to s', \text{where } s' \subseteq s \\
\text{(Forward transfer)} & r & ::= s \to s', \text{where } s' \subseteq s
\end{array}$$

Figure 4-15. The configuration for WebAssembly with embedded actions, supporting transfer and syncing of state based on the semantics of the original paper (Haas et al., 2017).

WebAssembly evaluation $\boxed{\{s, v^*, e^*\} \hookrightarrow_i \{s', v'^*, e'^*\}}$

$$\frac{\{s, v^*, e^*\} \hookrightarrow_i \{s', v'^*, e'^*\}}{\{s, v^*, L^k[e^*]\} \hookrightarrow_i \{s', v'^*, L^k[e'^*]\}}\ \textsc{Label} \qquad \frac{\{s, v^*, e^*\} \hookrightarrow_i \{s', v'^*, e'^*\}}{\{s, v^*, L^k[e^*]\} \hookrightarrow_i \{s', v'^*, L^k[e'^*]\}}\ \textsc{Local}$$

$$\frac{s_{\text{func}}(i, j) \neq \text{cl} \qquad A(j) = a \qquad \{s, v^*, \text{call } a_{\text{code}}\} \hookrightarrow_i^* \{s', v'^*, v\}}{\{s, v^*, \text{call } j\} \hookrightarrow_i \{s, v^*, v\}}\ \textsc{Action}$$

Figure 4-8. The semantics of actions, and invoking instructions in WebAssembly.

Without the second requirement, the debugger cannot know which of the received events can be (safely) handled next. In this chapter, we focus solely on the order in which they are processed. Other considerations around the exact timing of the events are important for real-time systems, but are impossible to handle in an online debugger context where execution can be paused for arbitrary periods. Although these requirements impose some limitations on the types of stateful operations we can support, we believe that a broad range of stateful operations align with these assumptions.

4.4.1 WebAssembly Semantics with Embedded Actions

WebAssembly is a portable, low-level binary code format built for secure and efficient execution across different platforms. Its formal semantics are based on a stack-based virtual machine, where instructions and values interact with a single, strictly typed stack, enabling fast static validation. The core semantics encompass structured control flow constructs, such as blocks, loops, and conditionals, along simple linear memory. To remove platform-specific dependencies, WebAssembly deliberately omits external interface definitions, including input and output operations. We have extended WebAssembly with a set of non-transferable actions which are clearly separated from regular code execution. This design choice enables us to have a clear and easy division between transferable and non-transferable code.

We build on the semantics of WebAssembly as defined by Haas et al. (2017). Figure 4-15 shows data extensions to WebAssembly needed to support non-transferable resources. The WebAssembly program state is defined as a configuration K, with global store s, local values v^*, and the current stack of instructions e^*. The *global* action table A contains all actions, each action a is a named pair of a closure cl and a transfer functions t and r. The closure consists of the code which performs the action over the non-transferable resource. The backward transfer function t, returns the state s' needed to perform the action, given the arguments v^* and the current state s of the server. The forward transfer function r, produces the state s' that has been altered by execution the action given the state s after executing the action on the client. We refer to elements of named tuples, such as the transfer function as a_{transfer}.

We use the terms *forward* and *backward* transfer to refer to the direction of the state changes, similar to program slicing.

The execution of a WebAssembly program is defined by a small-step reduction relation over the configuration $\{s; v^*; e^*\}$, denoted as $\hookrightarrow_i$, where i refers to the index of the currently executing module as shown in figure Figure 4-8. The WebAssembly semantics makes use of administrative operators to deal with control constructs, for example call i denotes a call to a function with index i . To mark the extend of an active control struct, expressions are wrapped into labels. Evaluation context L^k are used in the LABEL rule to unravel the nesting of k labels, allowing to focuses on the currently evaluation expressions $e*$. This rule, as defined in the WebAssembly semantics, is important for defining the out-of-place debugger semantics because it allows capturing the current continuation, (i.e. $L^k[]$) , just before invoking a remote call.

Naturally, the semantics of actions needs to be define both for when the debugger is active and during normal execution. On the bottom of Figure 4-8, we show how actions are executed during normal execution. Actions are defined in a global action table A, separate from WebAssembly functions. Whenever a function is called that is not present in the current instance i, the action rule finds the closure in the action table and shifts execution to evaluating the closure. This is similar to how the WebAssembly semantics handles function calls. Important to note is that actions are atomic, and reduce to a single value in one step, $\{s; v^*; \text{call } a_{\text{code}}\} \hookrightarrow_i^* \{s'; v'^*; v\}$. While we could relax this condition for synchronous events, the semantics to support asynchronous events would become more complex as shown in Section ref{oop:Asynchronous}. In the next section we give an overview of the semantics of actions during debugging.

Global syntax rules

$$\text{(Global configuration) } D ::= (C \mid S)$$

Client syntax rules

$$\begin{array}{lrl}
\text{(Client configuration)} & C & ::= \text{es}, [\![m_{\text{in}}]\!] \;;\; K \;;\; [\![m]\!] \\
\text{(Execution state)} & \text{es} & ::= \text{running} \mid \text{halted} \mid \text{invoked}\langle \text{es} \rangle \\
\text{(Debug commands)} & m & ::= \text{play} \mid \text{pause} \mid \text{step} \\
\text{(Internal messages)} & m_{\text{in}} & ::= \varnothing \mid \text{sync}\langle s, v \rangle
\end{array}$$

Server syntax rules

$$\begin{array}{lrl}
\text{(Server configuration)} & S & ::= \overline{\text{es}}, [\![\overline{m_{\text{in}}}]\!] \;;\; K \\
\text{(Execution state)} & \overline{\text{es}} & ::= \text{running} \mid \text{halted} \mid \text{invoking}\langle a \rangle \\
\text{(Internal messages)} & \overline{m_{\text{in}}} & ::= \varnothing \mid \text{invoke}\langle s, e^* \rangle
\end{array}$$

Figure 4-9. The syntax rules for a stateful out-of-place debugger, on top of the WebAssembly semantics shown in Figure 4-15. The rules are split into three groups, the global rules, the client rules, and the server rules. Elements in the server configuration are overlined whenever they need to be differentiated from the client configuration.

4.4.2 Configuration of the Stateful Out-of-place Debugger

Out-of-place debugging distributes a single program over two distributed entities, the local debugger which acts as the *client*, and the remote micro-controller which acts as the *server*. Recall that the idea is to execute most of the program on the client and only execute code attached to non-transferable resources on the server. We define the configuration for stateful out-of-place debugging in Figure 4-9.

The debugger configuration D is split into the client and server sides, $(C \mid S)$, which each hold a WebAssembly configuration K—the program state. The client side represents the main component of the debugger, which is responsible for receiving debug commands from the debugger's user interface. The server side represents only a small stub running on the remote microcontroller, which must facilitate access to non-transferable resources.

The client syntax rules shown in Figure 4-9, divide the configuration into three main parts divided by a semi-colon. The first component is the internal debugger state, consisting of the execution state es, and the internal message box $[\![m_{\text{in}}]\!]$ which receive messages from the server. Execution state es can be either *running*, *halted*, or *invoked*. The last state is used to indicate that the client is currently executing a remote invocation, and keeps track of the state of the execution state before the invocation. There is only one possible internal message that can be received from the server, *sync*, which is used to synchronize the state of the client after a remote function invocation. The second component is the WebAssembly configuration K, and the third component is the external-facing message box $[\![m]\!]$ for debug commands received from the user interface. For brevity, we have limited the supported debug commands to *play*, *pause*, and *step*.

The server configuration consists of the two components; the internal debugger state and the WebAssembly program state K. The internal debugger state contains the execution state $\overline{\mathrm{es}}$, and the internal message box $\overline{m_{\mathrm{in}}}$, which receives messages from the client. The (invoking a) state is used to indicate that the server is currently executing a remote invocation of the action a. The server can receive just one internal message, *invoke*, which is used to invoke an action. In order for the right state to be synchronized, the messages passes along the WebAssembly global store s and list of instructions e^*. We go into more details when discussing the server-side semantics. For clarity, we place a line over these components to differentiate them from the similar components in the client configuration.

In our implementation we also have an outgoing message box used to communicate all the information needed to update the debugger frontend, in order not to clutter the semantics we omit this message box in the semantics.

4.4.3 Stepping in a Stateful Out-of-place Debugger

Given the syntax rules for the out-of-place debugger, we can now define the evaluation rules, which are of the form $(C \mid S) \hookrightarrow^*_{d,i} (C' \mid S')$. Figure 4-10 shows the rules for the client side of the out-of-place debugger. The debugger message box $[\![m]\!]$ in C receives debug messages from the debugger frontend, which are processed in the order they are received in. Conceptually, step operations may involve either the server or the client taking a step. The determining factor is whether the step requires access to a non-transferable resource. Below, we outline the step and run rules.

step-client When the client is halted, and receives the debug command *step* in the external-facing message box, and the next instruction is not an action, then the execution takes a single step in the underlying WebAssembly semantics from K to K'.

play Whenever the halted server receives a *play* message, it will move the server to the *running* state.

pause Whenever the running server receives a *pause* message, it will move the server separator to the *halted* state.

Client evaluation rules

$$\frac{\neg\big(K = \{s; v^*; L^k[\text{call } i]\} \wedge a = A(i)\big) \qquad K \hookrightarrow_i K'}{(\text{halted}, [\![\varnothing]\!] \,;\, K \,;\, [\![\text{step}]\!] \mid C) \hookrightarrow_{d,i} (\text{halted}, [\![\varnothing]\!] \,;\, K' \,;\, [\![\varnothing]\!] \mid C)}\ \text{step-client}$$

$$\frac{}{(\text{play}, \text{bp}, (\text{halted}, [\![\varnothing]\!] \,;\, K \,;\, [\![\text{play}]\!]) \mid C) \hookrightarrow_{d,i} ((\text{running}, [\![\varnothing]\!], K) \mid C)}\ \text{play}$$

$$\frac{}{(\text{pause}, \text{bp}, (\text{running}, \varnothing, K) \mid C) \hookrightarrow_{d,i} ((\text{halted}, [\![\varnothing]\!] \,;\, K) \mid C)}\ \text{pause}$$

$$\frac{\neg\big(K = \{s; v^*; L^k[\text{call } i]\} \wedge a = A(i)\big) \qquad K \hookrightarrow_i K'}{(\text{running}, \varnothing, K \mid C) \hookrightarrow_{d,i} (\text{running}, \varnothing, K' \mid C)}\ \text{run-client}$$

$$\frac{K = \{s; v^*; L^k[v^n \text{ call } i]\} \qquad a = A(i) \qquad a_{\text{transfer}(v^n, s)} = s'}{\begin{array}{c}(\text{halted}, [\![\varnothing]\!] \,;\, K \,;\, [\![\text{step}]\!] \mid \text{halted}, [\![\varnothing]\!] \,;\, K^c) \hookrightarrow_{d,i} \\ (\text{invoked}\langle\text{halted}\rangle, [\![\varnothing]\!], K \,;\, [\![\varnothing]\!] \mid \text{halted}, [\![\text{invoke}\langle s', v^n \text{ call } i\rangle]\!] \,;\, K^c)\end{array}}\ \text{step-invoke}$$

$$\frac{K = \{s; v^*; L^k[v^n \text{ call } i]\} \qquad a = A(i) \qquad a_{\text{transfer}(v^n, s)} = s'}{\begin{array}{c}(\text{halted}, [\![\varnothing]\!] \,;\, K \,;\, [\![\text{step}]\!] \mid \text{halted}, [\![\varnothing]\!] \,;\, K^c) \hookrightarrow_{d,i} \\ (\text{invoked}\langle\text{running}\rangle, [\![\varnothing]\!], K \,;\, [\![\varnothing]\!] \mid \text{halted}, [\![\text{invoke}\langle s', v^n \text{ call } i\rangle]\!] \,;\, K^c)\end{array}}\ \text{run-invoke}$$

$$\frac{K = \big\{s; v^*; L^k[v^n \text{ call cl}]\big\} \qquad \text{update}(s, \Delta) = s' \qquad K' = \big\{s'; v^*; L^k[v]\big\}}{(\text{invoked}\langle\text{es}\rangle, [\![\text{sync}\langle\Delta, v\rangle]\!] \,;\, K \,;\, [\![m]\!] \mid C) \hookrightarrow_{d,i} (\text{halted}, [\![\varnothing]\!] \,;\, K' \mid C)}\ \text{sync}$$

Figure 4-10. The semantics of remote action invocation in out-of-place debugging.

run-client Similarly, when the server can take a (local) step in the underlying semantics, and is in the *running* state, the server takes a step through the underlying semantics.

These rules allow the debugger to execute a WebAssembly program that does not contain any actions. When during the execution of the program an action is encountered execution needs to be transferred to the server. For fullness, Figure 4-10 already contains the rule for handling the forward transfer of state by the client—as it is received from the server after a remote action call.

step-invoke When during stepping, the next instruction is a call to an action, the execution is transferred to the server device. The transfer function a_{transfer} calculates the state s' required to execute the action on the client. This state is passed to the server through the *invoke* message, along with the arguments of the call v^n, and the function id i to call.

Note that the client's execution state transitions to invoked⟨halted⟩. Before executing code on the server, we must remember that the execution was halted to restore it after the call. This is crucial because execution

Server evaluation rules

$$\frac{s'' = \text{update}(s, s')}{\begin{array}{l}(\text{invoked}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K \mid \text{halted}, \llbracket\text{invoke}\langle s', v^n \text{ call } i\rangle\rrbracket\ ;\ \{s; \varepsilon; \varepsilon\}) \\ \hookrightarrow_{d,i} (\text{invoked}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K \mid \text{invoking}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ \{s''; \varepsilon; v^n \text{ call } i\})\end{array}}\ \text{invoke-start}$$

$$\frac{K \hookrightarrow_i K'}{(S \mid \text{invoking}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K) \hookrightarrow_{d,i} (S \mid \text{invoking}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K')}\ \text{invoke-run}$$

$$\frac{\Delta = a_{\text{transfer}^{-1}}(s) \qquad K' = \{s; \varepsilon; v\}}{\begin{array}{l}(\text{invoked}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K \mid \text{invoking}\langle a\rangle, \llbracket\varnothing\rrbracket\ ;\ K') \hookrightarrow_{d,i} \\ (\text{invoked}\langle a\rangle, \llbracket\text{sync}\langle\delta, v\rangle\rrbracket\ ;\ K \mid \text{halted}, \llbracket\varnothing\rrbracket\ ;\ \{s; \varepsilon; \varepsilon\})\end{array}}\ \text{invoke-end}$$

Figure 4-11. The semantics of out-of-place execution, i.e., on the server, in stateful out-of-place debugging.

on the server can be triggered both while the client is halted and while it is running.

run-invoke When the client is in the running state and the next instruction is a call to an action, the execution is transferred to the server device. This rule is entirely analogous to the step-invoke rule, with that difference that the server will transition to the invoked⟨running⟩ state. This is important to be able to restore the running state after the call.

sync The synchronization rule updates the state of the client, with the difference received from the server after an invocation. This is identical to the update in the *invoke-start* rule. Finally, the execution state of the client is restored to es.

Figure 4-11 shows how the *invoke* message is handled by the debugger stub on the server side. The process is split into three steps, corresponding to four evaluation rules, first the server synchronizes the state based on the backward transfer and prepares the action call, second the action is performed, and finally the changes in state are transferred back to the client.

invoke-start When the client receives an invoke message, it updates the local state s with the snapshot s' of the *invoke* message. The *update* function simply overrides the current state s with those parts that are present in s'. To keep track that the client device is invoking an action, the execution state of the client is set to *invoking*. The status also stores which action a, is currently being executed in order to be able to get access to the transfer function after the invocation .

(Extended WebAssembly store)	s	::= {..., callbacks Cbs, events evt*}
(Callback environment)	Cbs	::= ∅
		Cbs, $x \mapsto i$
(Event)	evt	::= {topic memslice, payload memslice}
(Memory slice)	memslice	::= {start i32, length i32}
(Extended instructions)	e	::= ... | callback.set | callback.drop

Figure 4-16. The extended WebAssembly configuration, and the typing rules for the new concurrent callback system.

invoke-run When the client is invoking an action, it executes it simply by executing the underlying WebAssembly semantics.

invoke-end When invocation ends, there is only a single value v left on the stack on the client C. At this point, the client makes use of the $a_{\{\text{transfer}^{\{-1\}}\}}$ function of the action to compute which state needs to be synchronised. This difference is then transferred back to the server in a SYNC message, along with the return value v of the action.

4.5 Modeling Asynchronous Non-transferable Resources

The semantics so far, allow for the out-of-place debugger to handle programs with synchronous operations that are both stateless and stateful. However, in microcontroller systems, actions can be triggered asynchronously by elements such as sensors, hardware interrupts, and asynchronous communication protocols like MQTT. Pure WebAssembly does not have support for callbacks, therefore, we extend the WebAssembly semantics with a lightweight callback handling system as proposed by Lauwaerts et al. (2024). While our semantics largely follow their approach, we made minor adjustments to better align with our stateful out-of-place debugger. Our contribution lies in extending this semantics to support stateful out-of-place debugging, as shown in Section 4.6.

The required extension to support asynchronous events are shown in Figure 4-16 and Figure 4-12. Note that in this section we focus on explaining these extensions separately from our debugging semantics.

Figure 4-16 shows how the store s is extended with a callback table Cbs, which maps event topics to WebAssembly function indices i. The event system captures asynchronous events, such as hardware interrupts, and reifies them into a universal event queue. We further extend the global store s with an event queue evt*. All asynchronous events evt are captured in the event queue, similar to the debugging message queue shown before. Each event has a topic stored as a slice of WebAssembly memory, we keep track of these slices

Evaluation rules

$$\frac{s_{\{\text{callbacks}\}}[\text{topic} \mapsto \text{nil}] = s'_{\{\text{callbacks}\}}}{\{s; v^*; (\text{callback.drop topic})\} \hookrightarrow_{d,i} \{s'; v^*; \varepsilon\}}\ \text{deregister}$$

$$\frac{s_{\{\text{callbacks}\}}[\text{topic} \mapsto j] = s'_{\{\text{callbacks}\}}}{\{s; v^*; (\text{i32.const } j)(\text{callback.set topic})\} \hookrightarrow_{d,i} \{s'; v^*; \varepsilon\}}\ \text{register}$$

$$\frac{\begin{array}{c}\xi = s_{\{\text{events}\}}(0) \\ s'_{\{\text{events}\}} = \text{remove}\big(s_{\{\text{events}\}}, 0\big) \\ s_{\{\text{callbacks}\}}\big(\xi_{\{\text{topic}\}}\big) = \text{nil}\end{array}}{\{s; v^*; e^*\} \hookrightarrow_{d,i} \{s'; v^*; e^*\}}\ \text{drop} \qquad \frac{\begin{array}{c}\xi = s_{\{\text{events}\}}(0) \\ s'_{\{\text{events}\}} = \text{remove}\big(s_{\{\text{events}\}}, 0\big) \\ e'^* = \text{construct\_call}(s, \xi)\end{array}}{\{s; v^*; e^*\} \hookrightarrow_{d,i} \{s'; v^*; \text{Clb}[e'^*]e^*\}}\ \text{interrupt}$$

$$\frac{\{s; v^*; \text{Clb}[e^*]\} \hookrightarrow_{d,i} \{s'; v'^*; \text{Clb}[e'^*]\}}{\{s; v^*; e^*\} \hookrightarrow_{d,i} \{s'; v'^*; e'^*\}}\ \text{callback} \qquad \frac{}{\{s; v^*; \text{Clb}[\varepsilon]\} \hookrightarrow_{d,i} \{s; v^*; \varepsilon\}}\ \text{resume}$$

Figure 4-12. The reduction rules describing the event and callback handling in our concurrent callback system for asynchronous events in WebAssembly, based on the lightweight callback system of the WARDuino virtual machine (Lauwaerts et al., 2024).

by their *start* address and *length*. The topic of an event corresponds to the unique identifier of a category of events, to which callbacks can subscribe. Additional data of the event can be stored as a second slice of memory, which we refer to as the event's payload.

Figure 4-12 shows the reduction rules describing the event and callback handling, we discuss each of the rules in detail below.

register The register rule adds a new callback to the callback map, which maps a topic to a WebAssembly function index. The *callback.set* instruction takes an immediate memory slice, which corresponds to the topic string. The instruction updates the callback map for the topic with a function index j, which it takes from the stack.

deregister Callback functions can be removed from the callback map, with the *callback.drop* instruction, which simply takes a memory slice immediate, and removes the entry for the topic corresponding to the memory slice.

drop Whenever the event queue is not empty, the first event is taken from the queue, and its topic is looked up in the callback map. If no callback is registered for the topic, the event is simply dropped by this rule.

interrupt Whenever a popped event does correspond to a registered callback, its topic and payload are placed on the stack as arguments for the callback function. The callback function is called indirectly with its function index. For brevity, we leave out the details of this call construction in this figure, the detailed construction created by the *construct-call* function can be found in the appendix. The call is placed in the dedicated *Clb* label before the current series of instructions, so that the callback can be executed concurrently with the main program.

callback The callback rule is similar to the invoking rule, and allows the WebAssembly runtime to execute the callback within the *Clb* label.

resume The resume rule is triggered when the callback has finished executing, its label is empty, and the WebAssembly runtime can continue executing the program. Callback labels must always reduce internally to ε, since no callbacks can have a return value.

Whenever an event arrives in the event queue, the WebAssembly runtime will interrupt the current execution, and invoke the callback function associated with the event topic. Such callbacks cannot have a return type, to ensure that callbacks do not break a well-typed WebAssembly program. However, callbacks can update other internal state, such as global variables, or linear memory. Asynchronous events and callbacks introduce non-determinism into the WebAssembly languages, which can seriously complicate debugging of programs. However, simplifying debugging of non-deterministic bugs is beyond the scope of this chapter, and is an orthogonal problem to that of state desynchronization in out-of-place debugging. In fact, some recent work on debugging non-deterministic programs in WebAssembly uses some resource-heavy program analysis, which can benefit from out-of-place debugging to reduce overhead, and support resource-constraint microcontrollers.

4.6 Debugging Asynchronous Non-transferable Resources

The callback system and the asynchronous non-transferable resources it enables, present a second challenge for handling state desynchronization in out-of-place debugging. Identical to the other parts of the program's runtime, we wish to have the callback system run on the client. Unfortunately, events are generated on the side of the server. Building on the semantics we discussed so far, we show how out-of-place debuggers can deal with these kind of asynchronous state changes in the following sections.

4.6.1 An Example of Asynchronous Resources

To illustrate the challenges introduced by asynchronous resources to stateful out-of-place debugging, we take another detailed look at our running MQTT example (Figure 4-14). Developers familiar with the MQTT subscribe opera-

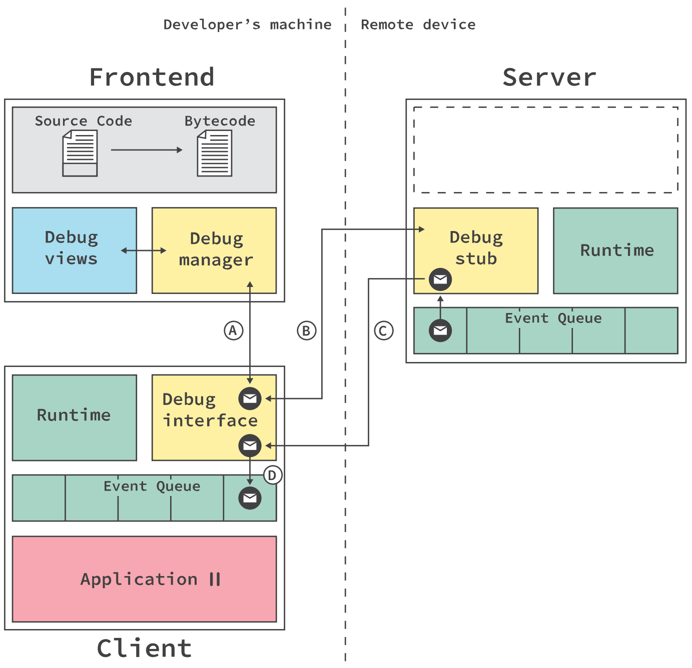


Figure 4-17.

tion, would expect it to send a message to the MQTT broker, indicating that the client is interested in a specific topic. From that moment on, the broker will forward messages of that topic to the client, where they will be handled by a callback function. To implement this action in our system, we expect it to send the appropriate message to the MQTT broker, and register a callback function for those MQTT message in the runtimes callback handling system.

In stateful out-of-place debugging, the update of the callback system by an action is a clear example of synchronous desynchronization, which is handled at the end of the actions invocation. However, whenever the client receives MQTT messages, it places these in the event queue of the callback handling system. This is a clear example of asynchronous desynchronization, which needs to be handled differently.

4.6.2 The Callback System in Out-of-place Debugging

We revisit the semantics of stateful out-of-place debugging entirely, since the current semantics have no way of dealing with events produced by non-transferable resources. We will define a new semantics $\hookrightarrow^{\alpha}_{d,i}$ that encapsulates the previous syntax and evaluation rules, but adds support for synchronization and control of event-driven non-transferable resources.

Our callback system adds two instructions to WebAssembly, which can change the callback map in the global store, DEREGISTER and REGISTER. As our example with MQTT subscribe illustrates, actions can use the instructions to change the map on the server. It is crucial to have these changes reflected on the client, since it has sole control of the callback system. However, these changes are still synchronous, so can be dealt with through the invocation rules already presented in Section 4.4.3.

It is important to note, that this synchronization is only necessary from server to client, since control over the callback system lies entirely with the latter. This means, that the server is not able to start a new WebAssembly callback execution autonomously, and therefore does not its callback map synchronized with any local changes on the client side.

The events are another matter, these are generated asynchronously, and so need to be synchronized asynchronously as well. However, in this case too, synchronization is only necessary from server to client.

4.6.3 Controlling the dispatching of Asynchronous Events

While it is important for microcontroller applications to interrupt a program's execution to handle asynchronous events, during debugging this is extremely distracting and confusing. Debugging relies on giving the developer control over the program's execution, but asynchronous code takes away this control. Furthermore, there are many non-deterministic bugs that depend on a certain order of events, or only appear when events are processed at certain points in the program (Li et al., 2023). We therefore want full control over the impact that asynchronous events have on the control flow of the program.

Figure 4-13 shows the extended semantics of the out-of-place debugger for handling and controlling asynchronous events, defined as the relation ($\hookrightarrow^{\alpha}_{d,i}$) which extends ($\hookrightarrow_{d,i}$). To provide developers with control over the event and callback system, the out-of-place debugger disables the automatic dispatching of events, as shown at the top of Figure 4-13. Specifically, the debugger will never take the *interrupt* step. Instead it provides a new debug message *trigger*, which takes the index of a event in the queue to be dispatched. However, some events cannot occur before other events, the most straightforward case is where one MQTT message is the consequence of another. In such cases, reordering the events may result in execution paths that are impossible without the interference of the debugger. To prevent the debugger from causing such impossible scenario's, the semantics assumes there is a partial order relation $<$ for the events in the queue. At any point in the debugging session, an event can only be dispatched if there is no undispatched event that is smaller under this relation. The TRANSFER-EVENTS rule describes

Evaluation rules

$$\boxed{\frac{(\hookrightarrow_i) \neq \text{interrupt}}{\text{dbg} \hookrightarrow_{d,i} \text{dbg}'}}$$

$$\frac{\xi = s_{\text{events}(j)} \quad s'_{\text{events}} = \text{remove}(s_{\text{events}}, j) \quad e'^{*} = \text{construct\_call}(s, \xi) \quad K' = \{s', v^{*}, \text{Clb}[e'^{*}]e^{*}\}}{(\text{halted}, [\![\varnothing]\!] ; \{s; v^{*}; e^{*}\} ; [\![\text{trigger } j]\!] \mid C) \hookrightarrow_{d,i} (\text{halted}, [\![\varnothing]\!] ; K' ; [\![\varnothing]\!] \mid C)} \text{trigger}$$

$$\frac{\text{length}(s_{\text{events}}) \leq j \vee \exists\, \text{evt} \;:\; \text{evt} < s_{\{\text{events}\}}(j)}{(\text{halted}, [\![\varnothing]\!] ; \{s; v^{*}; e^{*}\} ; [\![\text{trigger } j]\!] \mid C) \hookrightarrow_{d,i} (\text{halted}, [\![\varnothing]\!] ; \{s; v^{*}; e^{*}\} ; [\![\varnothing]\!] \mid C)} \text{trigger-invalid}$$

$$\frac{K'_{\text{events}} \neq \varnothing \quad K''_{\text{events}} = \varnothing \quad s = \{\text{events } K'_{\text{events}}, \text{memory memslice}^{*}\}}{(\text{es}, [\![\varnothing]\!] ; K ; [\![\varnothing]\!] \mid \overline{\text{es}}, \varnothing ; K') \hookrightarrow_{d,i} (\text{es}, [\![\text{sync}\langle s\rangle]\!] ; K ; [\![\varnothing]\!] \mid \overline{\text{es}}', \varnothing, K'')} \text{transfer-events}$$

$$\frac{K = \{s; v^{*}; e^{*}\} \quad \text{update}(s, \Delta) = s' \quad K' = \{s'; v^{*}; e^{*}\}}{(\text{es}, [\![\text{sync}\langle\Delta, v\rangle]\!] ; K ; \mid C) \hookrightarrow_{d,i} (Q, \text{bp}, (\text{es}, \varnothing, K')_{S} \mid C)} \text{sync-events}$$

Figure 4-13. The semantics of push-based asynchronous non-transferable resources in out-of-place debugging $\hookrightarrow^{\alpha}_{d,i}$, which encapsulates the relation $\hookrightarrow_{d,i}$, and provides control over the non-determinism of events to the developer.

how the client sends events to the server, as soon as the events are received. Since the event queue is an extension of the WebAssembly state, the same synchronization and updating mechanism is used as before. We provide a summary of each rule below.

trigger When the client receives a trigger message for event at index j, it pops the event from the event queue, and identical to the *interrupt* rule in Figure 4-12, it calls the corresponding callback function.

trigger-invalid If the index of the event in the trigger message is out of bounds, or the event is invalid, because there are still undispatched events that are smaller under the partial order relation, the server will return an error message.

transfer-events This rule shows how all events arriving on the client are forwarded to the server through the same synchronization message we used before. The message includes the events in the queue, and slices of memory containing the events' topic and payload.

4.7 Correctness of Out-of-place Debugging

Given the presented formalization of out-of-place debugging, we can now proof several interesting properties showing the soundness of the approach. Before we give the proofs of general correctness for our debugger, we first proof two lemmas showing that invoking a non-transferable resource from the server does not change the observable behavior of the program, compared to its normal execution on the client.

Specifically, we proof that given the execution of an action in the non-debugger semantics, there must exist a path in the debugger semantics that moves the client from the same starting state to the same end state.

Lemma 4-1. (Invoking completeness) The remote invoking of an action in the debugger semantics is *complete*, when for every debugging configuration dbg with K in S and, where the next instruction in K is a call to an action a, and K' the result of that action ($K \hookrightarrow_i K'$), the following holds:

$$\exists \mathrm{dbg}' : \mathrm{dbg} \hookrightarrow_{d,i}^{\alpha,*} \mathrm{dbg}' \wedge (K' \in S \text{ of } \mathrm{dbg}')$$

Proof. (Invoking completeness for $\hookrightarrow_{d,i}^{\alpha,*}$) Since we know that $K \hookrightarrow_i K'$ exists, we can construct a path in the debugger semantics that brings the client to the same state. The sequence is as follows:

1. *invoke-start* sets up the server-side execution of the action a, updating the server state to K'' using the backward transfer function.
2. *invoke-run* performs the action $K'' \hookrightarrow_i K'''$, thereby taking the program state to K''' in the server.
3. *invoke-end* sends all changes to the program state from the server to the client using the forward transfer function, and *sync* message.
4. *sync* applies the data from the forward transfer function to update the client's program state to K''''.

By definition, after *invoke-start*, K'' is equivalent to the client's original state K with respect to the semantics of the action a. Therefore, the execution $K'' \hookrightarrow_i K'''$ mirrors the direct semantics $K \hookrightarrow_i K'$. Since the forward transfer function transmits all changes to the program state, we have K'''' on the client equivalent to K' under the underlying language semantics.

□

This lemma shows that the state synchronization between the server and the client is correct when invoking an action. It is therefore crucial to proving the correctness of the debugger. The lemma in the other direction is likewise important. The invoking of actions is sound when for any action invocation that takes dbg to dbg′, there is also a path in the underlying language semantics that takes program state K to K', respectively the program states in S for dbg and dbg′.

Lemma 4-2. (Invoking soundness) Given any dbg $\hookrightarrow_{d,i}^{\alpha,*}$ dbg′, where the next instruction in K of C in dbg is a call to an action a, the sequence dbg $\hookrightarrow_{d,i}^{\alpha,*}$ dbg′ $=$ dbg$_0$ $\hookrightarrow_{d,i}^{\alpha}$ ... $\hookrightarrow_{d,i}^{\alpha}$ dbg$_n$ exists, and:

$$\forall i < n : K \in C \text{ of dbg}_i = K \in C \text{ of dbg}_0$$

Then:

$$\exists K' : K \hookrightarrow_i K' \wedge K' \in C \text{ of dbg}_n$$

The precise formulation is quite involved, but informally, the lemma states that given any sequence dbg $\hookrightarrow_{d,i}^{\alpha,*}$ dbg′ that starts from the call of an action a in the program state k of the client C, and ends in the first state dbg′ where this program state has been changed, we there must exist a sequence of steps in the underlying language semantics that takes K to K' (the new program state in dbg′). The proof follows from the construction of the debugging semantics.

Proof. Consider the initial debugger state dbg with program state K. To reach a state dbg′ where the client's program state has changed, the sequence of steps in the debug semantics, must eventually apply either the *step-invoke* or *run-invoke* rule (by construction of the semantics; see supporting lemma in Appendix G.2).

Once step-invoke or run-invoke are applied, the only possible sequence of rules is again:

1. *invoke-start* sets up the server-side execution of the action a, updating the server state to K'' using the backward transfer function.
2. *invoke-run* performs the action $K'' \hookrightarrow_i K'''$, thereby taking the program state to K''' in the server.
3. *invoke-end* sends all changes to the program state from the server to the client using the forward transfer function, and *sync* message.
4. *sync* applies the data from the forward transfer function to update the client's program state to K''''.

We know there is a step $K'' \hookrightarrow_i K'''$ in the underlying semantics, which is the result of the action a on the server, but on the client the program state moves from K to K''. By definition of the backward transfer function, K'' is semantically equivalent to K with respect to a. Therefore, the execution $K''toK'''$ has the same effect on program state as the transition $KtoK'$. Since the forward transfer function precisely transmits these changes back to the client, the updated client state K'''' is equivalent to K'.

Thus, a step $K \hookrightarrow_i K'$ exists, with K' in C of dbg_n. □

Since debuggers are used to inspect a programs execution to find the causes of errors, we define the correctness of the debugger semantic in terms of its observation of the underlying language semantics. We consider a debugger to be correct if it can observe any execution that can be observed by the underlying language semantics, and conversely, that any execution observed by the debugger semantics can be observed by the language semantics. We call these two properties respectively soundness and completeness.

Theorem 4-1. (Debugger soundness) Let K be the start WebAssembly configuration, and dbg the debugging configuration, where S contains the WebAssembly configuration K'. Let the debugger steps $\hookrightarrow_{d,i}^{\alpha,*}$ be the result of a series of debugging messages. Then:

$$\forall\ \text{dbg} : \text{dbg}_{\text{start}} \hookrightarrow_{d,i}^{\alpha,*} \text{dbg} \implies K \hookrightarrow_i^* K'$$

Proof. □

Theorem 4-2. (Debugger completeness) Let K be the start WebAssembly configuration for which there exists a series of transition $\frac{\hookrightarrow}{g_i^*}$ to another configuration K', and $\text{dbg}_{\text{start}}$ the corresponding starting debugger configuration with K in S. Let the debugging configuration with K' be dbg. Then:

$$\forall K' : K \hookrightarrow_i^* K' \implies \text{dbg}_{\text{start}} \hookrightarrow_{d,i}^{\alpha,*} \text{dbg}$$

Proof. □

4.8 Implementation

We have implemented the stateful out-of-place debugger formalized above in a prototype debugger, called *Edward*. The *Edward* debugger is built on top of the WARDuino runtime (Lauwaerts et al., 2024), a WebAssembly runtime for microcontrollers. The prototype provides the features described in the previous section. Additionally, we created a new high-level interface for defining actions which integrates the state synchronization interface described in Section 4.4. The stateful debugger can be used in VS Code to debug AssemblyScript programs running on an instance of the WARDuino runtime, thanks to a dedicated extension to the VS Code IDE.

4.8.1 Virtual Machine Requirements

In order to implement out-of-place debugging, any candidate VM must support the following:

1. Standard instrumentation required for halting and stepping through a program.
2. Support forward and backward state transfer for non-transferable resources.
3. Update the state provided the data received from the transfer functions.
4. Capture and serialize all asynchronous events produced by non-transferable resources.

First, the virtual machine needs to support the elementary debug commands of any online debugger, which at least include halting and stepping through a program. Second, as we have demonstrated, state synchronization is equally fundamental for out-of-place debugging. The virtual machine should be able to support forward and backward state transfer for non-transferable resources. Third, given the data provided by the transfer functions, the virtual machine must be able to update its own state. Four, the virtual machine must be able to capture and serialize all asynchronous events produced by non-transferable resources. Thereby allowing them to be forwarded from server to client.

4.8.2 Example: the MQTT Subscribe Action

To illustrate how the new interface for defining stateful actions works, we will discuss the implementation of the MQTT *subscribe* action. The subscribe action is exposed in the runtime as a WebAssembly function that takes three unsigned 32-bit integers as arguments, corresponding to the location of the topic string in WebAssembly memory (offset and length), and the function index of the WebAssembly, which will act as callback function for events

```
1  void subscribe_internal(Module *m,
2      uint32_t topic_param,
3      uint32_t topic_length,
4      uint32_t fidx) {
5    const char *topic =
6      parse(m, topic_length,
       topic_param);
7
8    mqttClient.subscribe(topic);
9
10   Callback c = Callback(m, topic,
     fidx);
11   CallbackHandler::add_callback(c);}
12 def_action(subscribe, threeToNoneU32)
   {
13   uint32_t topic = arg2.uint32;
14   uint32_t length = arg1.uint32;
15   uint32_t fidx = arg0.uint32;
16
17   subscribe_internal(
18     m, topic, length, fidx);
19
20   pop_args(3);
21   return true;
22 }
```

Listing 4-12. The implementation of the MQTT *subscribe* action in the WARDuino runtime, without stateful out-of-place support.

from the topic. Actions are implemented directly in the WARDuino virtual machine using C macros. In order to implement stateful actions, we have extended the existing macros with two new macros. We will discuss each macro in turn.

Listing 4-12 shows the standard implementation of the subscribe action using the *def_action* macro. We have split the definition into an internal function, shown on the left, and the interface definition on the right. The internal function implements the behavior of the subscribe action, it receives as parameters the WebAssembly *m* module in which it is executed, and the offset and length of the topic string, and the function index. On line 6, the parse function will extract the topic string from the WebAssembly memory and parse it as an UTF8 string. Using the MQTT client the action will subscribe the microcontroller to the given topic. For brevity, we have left out the exception handling, in case the MQTT client has not been initialized, or the communication with the MQTT broker fails. After subscribing successfully, any messages from the MQTT broker will be routed to the concurrent callback system in the runtime. On lines 10 to 11, the action registers the WebAssembly function with index *fidx* to the callback environment. This way, the callback system will concurrently call the function whenever a new message arrives.

The interface definition on the right side of Listing 4-12 defines the action as a proper WebAssembly function using the *def_action* macro. The macro takes the name of the action, and its type as arguments. In the example, the subscribe action takes three arguments and returns nothing. The body of the macro takes the arguments from the stack and passes them to the internal function which performs the action. At the end the macro lifts the consumed arguments from the stack, and returns true. The boolean value returned by

```
// on server
def_to_client(subscribe) {
  uint32_t topic = arg2.uint32;
  uint32_t length = arg1.uint32;

  // add transfer to be send with invoke
  sync_memory(m, topic, length);
}
```

```
// on client
def_to_server(subscribe) {
  // add transfer to update callback env
  sync_callback();
}
```

Listing 4-13. The implementation of the transfer functions for the MQTT *subscribe* action in the WARDuino runtime, to enable stateful out-of-place support.

actions is used to indicate failure, and are used by the runtime to throw WebAssembly traps in case something goes wrong.

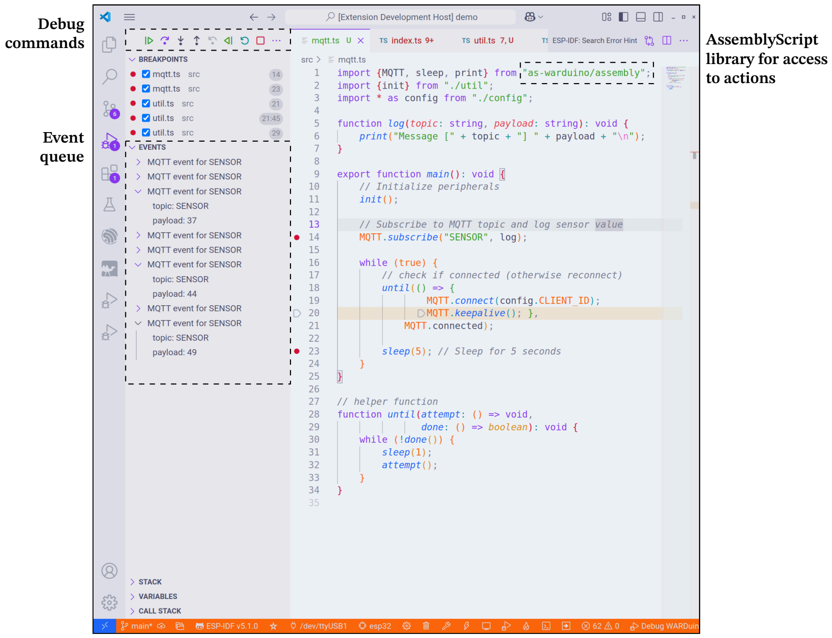
4.8.3 Stateful Actions for Non-transferable Resources

In order to enable stateful out-of-place debugging with the subscribe action, we need to define both the transfer from server to client, and vice versa. Analogous to the action definition this can be done in WARDuino using C macros, however, we also provide a number of primitives for constructing transfers. The primitives hide the specifics of the debugger's communication protocol, and allow library implementers to focus exclusively on what state needs to be transferred for a given action. In essence, each primitive will extend a hidden transfer object with the necessary information, and when the debugger sends the invoke message, it will serialize the constructed transfer and include it in the message.

Listing 4-13 shows the implementation of both transfer functions for the subscribe action. The left side of the figure, shows how the server must transfer state to the client, and the right side how the client must perform the action and return the state changes to the server.

Consider first the server, the *def_to_client* macro defines the transfer function for the subscribe action, which in our semantics is used in the *step-client* rule. Since the transfer is created right before the action should be performed, it can easily look at its arguments on the stack. The subscribe action only relies on the topic string in WebAssembly, so the transfer only needs to sync this slice of memory. This can be done with one of the primitives we provide to help with the state synchronization, in this case *sync_memory*. The primitive takes as arguments, a WebAssembly module, offset, and length, and will add the slice of memory to the transfer.

For the client, the *def_to_server* macro works slightly differently, it is executed after the action has been performed, and needs to define which state needs



BREAKPOINTS
- mqtt.ts src 14
- mqtt.ts src 23
- util.ts src 21
- util.ts src 21:45
- util.ts src 29

EVENTS
- MQTT event for SENSOR
- MQTT event for SENSOR
- MQTT event for SENSOR
  - topic: SENSOR
  - payload: 37
- MQTT event for SENSOR
- MQTT event for SENSOR
- MQTT event for SENSOR
  - topic: SENSOR
  - payload: 44
- MQTT event for SENSOR
- MQTT event for SENSOR
  - topic: SENSOR
  - payload: 49

```
import {MQTT, sleep, print} from "as-warduino/assembly";
import {init} from "./util";
import * as config from "./config";

function log(topic: string, payload: string): void {
    print("Message [" + topic + "] " + payload + "\n");
}

export function main(): void {
    // Initialize peripherals
    init();

    // Subscribe to MQTT topic and log sensor value
    MQTT.subscribe("SENSOR", log);

    while (true) {
        // check if connected (otherwise reconnect)
        until(() => {
                MQTT.connect(config.CLIENT_ID);
                MQTT.keepalive(); },
            MQTT.connected);

        sleep(5); // Sleep for 5 seconds
    }
}

// helper function
function until(attempt: () => void,
               done: () => boolean): void {
    while (!done()) {
        sleep(1);
        attempt();
    }
}
```

Figure 4-18. A screenshot of the out-of-place debugger in VS Code.

to be synchronized. In the case of our example, only the callback map is updated by the subscribe action. For the transfer we can use the *sync_callback* primitive which will add to the transfer, the minimal necessary data to update the callback environment on the server.

4.8.4 Prototype: Testing and Debugger Frontend

Our prototype implementation on top of the WARDuino runtime allows developers to use the existing VS Code extension for the warduino debugger to debug AssemblyScript programs using stateful out-of-place debugging. Figure 4-18 shows a screenshot of the debugger frontend in VS Code. The frontend supports the standard debug operations, pause, play, step forward, step into, step over, and breakpoints. Additionally, the extension features a view of the current events in the event queue on the server. These events can be triggered at any point by the developer, similar to performing a step.

We tested the prototype implementation, by checking the invoke non-interference as described in Section 4.4. To test this empirically, we randomly generating a thousand simple WebAssembly programs, which included a number of actions that changed the memory in the WebAssembly module. We ran the programs both with and with the out-of-place debugger, and

```
1  import * as wd from warduino;
2
3  const LED: u32 = 25;
4  const BUTTON: u32 = 26;
5
6  function buttonPressed(): void {
7    wd.digitalWrite(LED, !wd.digitalRead(LED));
8  }
9
10 export function main() : void {
11   wd.interruptOn(BUTTON, wd.FALLING, buttonPressed);
12   while(true);
13 }
```

Listing 4-14. A simple AssemblyScript program that toggles an LED when a button is pressed.

verified that the memory at the end of the program was indeed identical for each program.

4.9 Evaluation

4.9.1 Debugging Common Bug Issues

As mentioned in the introduction, a 2021 study on `5,565 bugs in 91 IoT projects` showed that the most frequent types of bugs are related to software development and device issues (Makhshari and Mesbah, 2021). In this section, we show an example program illustrating how out-of-place debugging better accommodates finding and solving device issues than regular remote debugging. Cref{subsec:concurrency} provides a similar comparison but for a software development issue due to concurrency.

The Bug. Many device issues are related to handling interrupts (Makhshari and Mesbah, 2021). Listing 4-14 shows a simple AssemblyScript application that toggles an LED when a button is pressed. The code listens for a hardware interrupts triggered on the falling edge of the button pin (line 11). Upon receiving an interrupt, the `buttonPressed` function is called, which toggles the LED (line 7). While the code may not contain errors, the hardware can cause bugs in it. Consider the following bug scenario: when testing the application with a real button, the LED sometimes does not change despite the button being pressed.

Bugfixing with a Remote Debugger. With a regular remote debugger, developers could start their diagnosis by adding a breakpoint in the `buttonPressed` callback function triggered when pressing the button. Note that in this simple example, there is only one single callback function, but in more complex IoT applications developers may need to place breakpoints in many callback functions as it is difficult to rule out which ones are not causing to the faulty behavior.

Stepping through code with asynchronous callbacks is generally not easy with current state of the art remote debuggers. Keeping track of all the asynchronous callbacks increases the number of times a developer needs to manually step through the application before discovering the error, complicating debugging. Moreover, stepping through the code is relatively slow, as the network latency between the developer's machine and the remote device slows down the debug session. Finally, most applications will not feature a busy loop as in our example, but the main thread runs concurrently with the asynchronous invocations, making it harder to notice errors.

Once the developer has stepped through all the asynchronous code letting the callbacks execute, the de developer might notice that the `buttonPressed` callback is strangely invoked multiple times. The reason is that a single button press can trigger multiple hardware interrupts due to a common problem of physical buttons called *contact bouncing* (McBride, 1989). Contact bouncing happens when the voltage of a mechanical switch pulses rapidly, instead of performing a clean transition from high to low. In that case, the pin can register a falling edge multiple times in a row. Subsequently, the `buttonPressed` function is triggered multiple times for a single press. If contact bouncing causes the function to be triggered an even number of times, the state of the LED seems to remain the same, making the developer believe the code does nothing. It is not trivial to deduce the underlying contact bouncing problem by only stepping through the program.

Bugfixing with EDWARD. Let us now revisit the scenario using out-of-place debugging. With EDWARD, developers can pull an out-of-place debug session from the remote device, and begin debugging locally at their machine. EDWARD provides the developer with a dedicated view on the event queue with all asynchronous events that happen at the remote device, and the ability to choose when the next event happens. When the developer pushes the physical button once during debugging, they will immediately notice that EDWARD's events view suddenly contains multiple identical events for a single button press, as shown in cref{fig:manyevents}. This information enables the developer to more easily detect the contact bouncing issue.

If the developer has not yet deduced the root cause of the bug, they could use stepping through the code in a similar way than when using the remote debugger. However, this time, stepping through the code is fast as debugging happens locally without incurring in network communication. Moreover, EDWARD allows debugging the program backwards. This means that during debugging when the LED does not turn on, the developer can step back to the previous step to diagnose what exactly went wrong during the execution. There is no need to restart the program and try to guess what the right conditions for the bug were.

begin{figure} includegraphics[width=0.25textwidth]{figures/screenshots/manyevents} caption{The debugger frontend shows a list of identical interrupts after a single button press.} label{fig:manyevents} end{figure}

Conclusion. This example illustrates that using out-of-place debugging makes a difference when debugging device issues compared to a remote debugger. Since EDWARD captures all non-transferable resources and provides a view on the event queue with all asynchronous events that happened at the remote device, developers can more easily diagnose device issues. For those cases where stepping is still needed, this happens with low latency. EDWARD also allows developers to step backwards, potentially reducing the debugging time as applications may not need to be restarted to reproduce the conditions for the bug to appear.

4.9.2 Quantitative Evaluation

We now present some preliminary quantitative evaluation of EDWARD, to underscore the potential of our approach to reduce performance impact while debugging IoT devices.

begin{figure} begin{tabular}{ lrr } & #Instructions & \ Location independent & 2092 & 99.15% \ Nt & 18 & 0.85% \ hline Total & 2110 & 100% \ end{tabular} caption{Labeling of Wasm instructions for a smart lamp application (cref{app:concurrency} in cref{subsec:concurrency}).} label{tab:colouring} end{figure}

paragraph{Code Analysis.} To analyze the potential communication needed between debugger and remote device, consider the smart lamp application written in AssemblyScript allowing users to control the brightness of an LED (cf. cref{app:concurrency} in cref{subsec:concurrency}).

While remote debugging requires network communication between debugger and the remote device for emph{all} debugging operations and all types of instructions, oop debugging only requires network communication for those instructions that access nt resources. In order to get an estimate of the amount of instructions which are location dependent compared to the non-transferable instruction we labeled each of the Wasm instructions of the smart lamp application's code as a location-independent instruction, or an instruction that accesses a nt resource. The results shown in cref{tab:colouring} confirm our suspicion that location-independent instructions outweigh instructions accessing nt resources.

%Our benchmarks were conducted using an ESP32-DevKitC V4 boardfootnote{url{https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html}} connected with a dell xps 9310 laptop through a local network. This board features an ESP32 WROVER IE chip that operates at 240 MHz, with 520 KiB SRAM, 4 MB SPI flash and 8 MB PSRAM.

paragraph{Network Overhead.} In order to get an estimate of the difference in network overhead between remote debugging and oop debugging we benchmarked the (debugging) network overhead of the smart lamp application. Our benchmarks were performed on a M5StickC cite{m5stickc} connected to a MacBook Pro with an Apple M1 Pro chip operating at 3.2 GHz CPU and 32GiB of RAM, through a local network.

Cref{fig:remote_debugging_overhead} plots the network overhead of stepping through the application with a remote debugging session. As we can see, there are small step-wise fluctuations caused by the changing amount of local variables in the program. The network overhead for each debugging step is approximately 2.2 kB.

For oop debugging, we benchmarked the network overhead of taking a full snapshot at each remote stepping operation, i.e. the network overhead involved with starting an out-of-place debugging session. Note that in practice the developer needs to perform this operation only once. Cref{fig:remote_snapshot_overhead} shows the results of taking a snapshot at each stepping operation of the smart-lamp application. As expected, the network overhead involved with taking a full snapshot is much higher than a single debugging step, each full snapshot takes approximately 130 kB.

The significant difference in network overhead between a remote debugging step and a full snapshot is expected and is mostly because the snapshot captures the stack and a full memory dump of the running application.

Luckily, once a snapshot has been taken the debugging session can be executed locally and the subsequent debugging session will be much faster. Avoiding access to the remote device reduces network overhead and lowers debugging latency. The network overhead for proxied calls is much smaller than a normal debugging step and takes at most 10 bytes per remote call with an additional 4 bytes per argument. More importantly the network overhead for stepping through each of the location independent instructions is zero.

paragraph{Latency.} Finally, we also benchmarked the difference in latency between local debugging steps and remote debugging steps. When stepping through the smart-lamp application with oop debugging, we find that local steps take on average approximately 5ms while a remote emph{proxy call} takes approximately 500ms.

In practice this means that the developer using oop debugging will perceive almost instantaneously local debugging steps interleaved with remote calls which are perceived slower. As these non-transferable instructions make up less than 1% of the code, most debugging steps will be able to be executed fast.

begin{figure}[t] begin{center} includegraphics[width=1columnwidth]{figures/remote_step_overhead} caption{Network communication overhead of 30 step operations using remote debugging. Note that the overhead axis starts at 2kB.} label{fig:remote_debugging_overhead} end{center} end{figure}

4.10 Discussion

Before giving a detailed overview of the related work, we now consider the wider context of our work and outline the advantages and disadvantages of our design decisions.

On the design choices

This chapter presents the first formal foundation for out-of-place debugging and address within it the important challenge of state desynchronization, which has been neglected by previous works. This resulted in the first stateful out-of-place debugger implementation. We choose to focus on the context of microcontroller programming, as it is a domain where out-of-place debugging can provide significant benefits, and where the support for stateful operations is crucial. The resource constraints imposed by microcontrollers are the main motivation for out-of-place debugging in this context, which inevitably impacted the design choices we made in a significant manner.

The impact of the resource constraints especially impacted the first of our key assumptions, that underlie our formalization. To minimize the impact of transfering state, we require that state dependencies of the actions can be know statically. This allows both client and server to quickly determine which parts of the state need to be transferred, without the need for expensive analysis procedures.

Additionally, we have chosen to keep the modeling of the asynchronous events simple, by only considering the order of events and assuming that any interleaving of events in the program is possible. The main motivations for this choice, were to reduce the complexity of our formalization, and keep the focus on how out-of-place debugging can be extended with stateful operations, and how this can be formalized. Certainly more complex and accurate models for the events exist, which do take into account the interleavings, and the exact timing of events. Using existing models for event interleaving, it should be possible to weaken our assumptions, and thereby relax the requirements our system puts on stateful actions.

Opportunities for advanced environment modeling

The current abstract of asynchronous events in our debugger semantics is a very simple model of the environment. The model does not take into account the exact timing of events, and assumes all interleaving of events are possible. However, such timings and interleavings are crucial for many applications (Lamport, 1978). Luckily, this problem has been extensively studied in the field of distributed systems, and a wide range of models exist to capture the asynchronous behavior. Likewise, the literature on automated testing has a broad range of techniques for modeling the environment. We give a brief overview of the literature on synchronization and consistency in distributed systems, and environment modeling in testing.

4.11 Related Work

We have discussed the different implementations of out-of-place debugging extensively in cref{oop:oop}. In this section, we will discuss other related works.

Remote Debugging

In remote debugging (Rosenberg, 1996), a debugger frontend connects to a remote backend that executes the target program. However, remote debugging may worsen the probe effect (Gait, 1986) and can experience significant delays due to the overhead of running the debugger on the microcontroller coupled with continuous communication requirements. Regardless, due to the ability to debug remote processes, remote debuggers are ubiquitous in software development, with popular debuggers such as GDB (Free Software Foundation, n.d.) and LLDB (“Remote Debugging - LLDB,” n.d.), as well as default support for remote debugging by many development environments

Remote Debugging Embedded Systems

Remote debugging is widely used in embedded systems (Pötsch et al., 2017, Skvařc Bŏzĭc et al., 2024, Söderby and De Feo, 2024) and typically follows one of two approaches: stub or on-chip debugging (Li et al., 2009). A stub is a lightweight software module running on the microcontroller that instruments the program, whereas on-chip debugging employs dedicated hardware —such as JTAG-based debuggers ("IEEE Standard for Test Access Port and Boundary-Scan Architecture," 2013)—to facilitate the process. These hardware debuggers can integrate with various software tools (Söderby and De Feo, 2024), including the popular OpenOCD debugger (Högl and Rath, 2006). Using hardware debuggers is not easy, and is mostly used to debug programs written in low-level programming languages such as C and C++. Additionally, there are several security concerns with JTAG interfaces (Lee et al., 2016, Vishwakarma and Lee, 2018) by allowing attackers to reverse engineering the microcontroller's software, or

To provide a more modern programming experience, several virtual machines (VMs) tailored for microcontrollers have been developed to abstract away the complexities of low-level programming, and provide stubs for remote debugging capabilities—to replace the hardware debuggers. Notable examples include Espruino for JavaScript (Williams, 2014), MicroPython for Python (George, 2021), and WARDuino and Wasm3 for languages compiled to WebAssembly (Lauwaerts et al., 2024, Massey and Shymanskyy, 2021). The debugging support varies widely between these VMs. For example, Espruino (Williams, 2014) enables remote debugging of JavaScript programs and allows developers to modify source code to log runtime information (e.g., stack traces), which is either forwarded to a debugger client or stored on the microcontroller if the client is disconnected. In contrast, MicroPython (George, 2021) does not offer a remote debugger, requiring programmers to rely on printf statements for debugging. Wasm3 introduces a source-level remote debugger integrated with GDB (Shymanskyy, 2023), though this feature is in its early stages and has not been actively maintained for the past two years. In comparison, our work not only delivers a remote debugging experience but also provides online debugging with minimal latency on demand. Moreover, to the best of our knowledge, no other approach offers developers the ability to access and control the processing of events generated by the remote device.

Out-of-place Debugging

We briefly discussed the previous works on out-of-place debugging in Section 4.1.1. Here, we provide a more indepth discussion of both the IDRA, and the Edward debugger, and the difference between the motivation behind the two debuggers.

The IDRA debugger (Marra et al., 2018) was developed to debug big data applications written in Pharo Smalltalk (Black et al., 2010). Given this context, it had two objectives. First, to enable the online debugging of non-stoppable applications. In a live production setting, it is not possible to stop the running of a big data applications because of a fault in one of the cluster's nodes. By moving the debugging session locally, out-of-place debugging can still provide a online experience similar to a remote debugger without the need to stop the application on the remote cluster. Second, the IDRA debugger aims to reduce debugging interference caused by the communication delays in remote debuggers, by reducing the amount of communication needed with the remote process.

The Edward debugger on the other hand was developed for the microcontroller environment, where the motivations for out-of-place debugging are very different. The main motivation of the Edward debugger is to overcome the resource constraints of the microcontroller, allowing for more complex and resource intensive debugging techniques. This in turn provides a more modern programming experience to developers. The work further focuses on asynchronous non-transferable resources. While access to non-transferable resources in IDRA is always in one direction, from local debugger to remote process, the Edward debugger allows for two-way communication, based on the callback system from the WARDuino runtime. This is the same system we based our semantics of asynchronous events on (Section 4.5).

Synchronization and Consistency in Distributed Systems

In our work, the out-of-place debugger comprises two remote processes, the server and the client. The semantics of the debugger clearly describe the synchronization between these two devices, however, if we were to extend the debugger to multiple devices, the synchronization would become much more complex. Synchronization in distributed systems has of course been widely studied. Clock synchronization goes back to the earliest distributed systems in the seventies and eighties (Kopetz and Ochsenreiter, 1987, Lamport, 1978, Schmuck and Cristian, 1990), and has been a crucial part of distributed systems ever since (Auguston et al., 2005). In fact, with the rise of internet of things applications, the problem has received renewed attention (Mani et al., 2018, Yĭgitler et al., 2020).

More generally, the problem of replicating data and consistency within a distributed network is an enormous field of research on its own. Much effort has been put into developing solutions for (strong) eventual consistency, where the requirement for synchronization is weakened to allow for higher availability. A common approach is to use conflict-free replicated data types (CRDTs) (Shapiro et al., 2011), which allow for concurrent updates to data without the need for any coordination (Almeida, 2024). It is an open question whether eventual consistency is enough for out-of-place debugging, or whether stronger consistency guarantees are needed. However, many other forms of consistency exist, such as sequential consistency (Lamport, 1979), causal consistency (Perrin et al., 2016, Terry et al., 1994), and linearizability (Herlihy and Wing, 1990). It is our believe that the type of consistency used in out-of-place debugging is tied strongly to its application context. Yet given the vast amount of work in this field, we believe that the existing techniques for consistency can be used to generalize our formalization to multiple devices, and to further strengthen the formal guarantees.

Program Slicing

Given the microcontroller setting, our approach determines the state needed to be transferred statically as part of the definition of the actions on non-transferable resources. However, for more complex actions, it would be advisable to use static analysis to determine the slice of the state. This is very similar to program slicing (Weiser, 1981, Xu et al., 2005) which decomposes a program into segments based on a *slicing criterion*. This criterion can slice in two directions, either *backward* when identifying segments that might affect the criterion, or *forward* when the segments is affected by the criterion. In our semantic, the transfer at the start of an invocation is similar to backward slicing, and the difference returned at the end is similar to forward slicing. Many different techniques for slicing exist (Xu et al., 2005), both dynamic and static, or hybrid. Only a few works have looked into static (Stiévenart et al., 2022) and dynamic (Stiévenart et al., 2023) slicing of WebAssembly programs, however, many of the existing techniques can be expected to work with WebAssembly as well, without great difficulty.

Environment Modeling

Environment modeling is a technique used in testing to model the behavior of the environment in which a program runs (Blackburn, 1998). Such models are often used for automatic test generation (Auguston et al., 2005, Dalal et al., 1999) for a certain specification, and has also been applied to real-time embedded software (Iqbal et al., 2015). Our work models the asynchronicity of the environment through simple partial order reduction of instantaneous events. This model enables exploring different behavior based on the order of events, and to a certain extent the timings of asynchronous events. More advanced models of the environment could help take into account additional dependencies between events, and real-time effects.

4.12 Conclusion

While existing out-of-place debuggers can already support a wide range of programs and application domains with purely stateless operations on non-transferable resource, they lack support for stateful operations or provide only some very minimal ad hoc support. In this work, we address this limitation by presenting the first formal semantic for out-of-place debugging, in which we incorporate our novel stateful out-of-place debugging technique. Our approach allows for the debugging of programs with stateful operations on non-transferable resources, with a lazy synchronization strategy where state is only send to the client device when it is required. Our formalization allows us to also define correctness for stateful out-of-place debugging, which we divide into soundness and completeness. The proof for these theorems show that stateful out-of-place debugging is able to debug programs without introducing impossible execution paths, or missing concrete paths. We have implemented our approach in a prototype debugger, called *Edward*, which is built on top of the WARDuino runtime, and VS Code extension. Initial empirical testing shows that our implementation indeed satisfies the correctness criteria defined in our formalization.

Chapter 5

Multiverse debugging on microcontrollers

Our knowledge can only be finite, while our ignorance must necessarily be infinite.
— Karl Popper, *Knowledge without Authority*

The second, and final, new debugging technique we investigated for microcontrollers, is multiverse debugging. As part of our investigation, we extended multiverse debugging to handle input/output operations, and created a prototype debugger that enables reversible actions on microcontrollers.

5.1 Introduction

Debugging non-deterministic programs is a challenging task, since bugs may only appear in very specific execution paths (Gurdeep Singh, 2022, McDowell and Helmbold, 1989). This is especially true for microcontroller programs, which typically interact heavily with the environment. This makes reproducing bugs unreliable and time-consuming, a problem that traditional debuggers do not account for. By contrast, multiverse debugging (Torres Lopez et al., 2019) is a novel technique that solves this problem by allowing programmers to explore all possible execution paths. A multiverse debugger allows users to move from one execution path to another, even jumping to arbitrary program states in parallel execution paths. This entails traveling both forwards and backwards in time, i.e. a multiverse debugger is also a time travel debugger. So far, existing implementations work on abstract execution models to explore all execution paths of a program (Pasquier et al., 2023a, 2023b, 2022; Torres Lopez et al., 2019). Within these semantics, only the internal state of the program is controlled.

Unfortunately, debugging programs that involve input/output (I/O) operations using existing multiverse debuggers can reveal inaccessible program states that are not encountered during regular execution. This is known as the probe effect (Gait, 1986), and can occur in multiverse debuggers when they do not account for the effect of I/O operations on the external environment when changing the program state. Encountering such states during the debugging session can significantly hinder the debugging process, as the programmer may mistakenly assume a bug is present in the code, when in fact, the issue is caused by the debugger. In this chapter, we investigate how we can scale multiverse debugging to programs running on a microcontroller which interacts with the environment through I/O operations. This introduces three new challenges.

First, the effect of output operations on the environment can influence later states in the execution path, for example when a robot drives forward. Therefore, when stepping backwards, the changes made to the environment by the program must be reverted to stay consistent with normal execution. Otherwise, the debugger may enter inaccessible execution paths. This far-reaching probe effect, is especially difficult to control when making arbitrary jumps in the execution tree.

Second, input operations from the external environment make it difficult to maintain reproducibility (Frattini et al., 2016). There are often too many possible execution paths to explore due to an infinite range of inputs. Furthermore, it is impractical for developers to enumerate all possible inputs, or to perfectly configure the environment to achieve a specific execution path every time. Without a way to handle the large ranges of possible inputs, the reproducibility of non-deterministic bugs in the multiverse debugger becomes challenging.

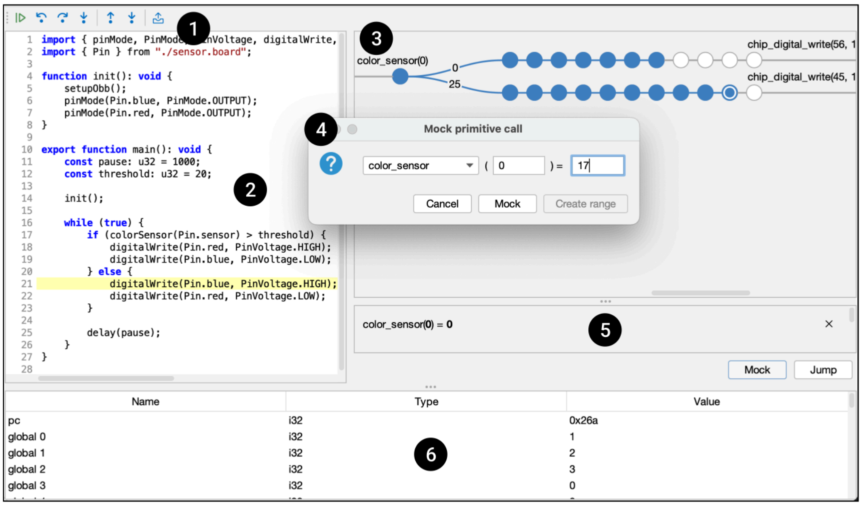
Third, due to the hardware limitations of the microcontrollers that we target it is unfeasible to run the multiverse debugger entirely on the microcontroller. We thus need to expand multiverse debugging so that it can be used even in such a restricted environment.

5.2 MIO: Multiverse Input/Output Debugger in Practice

Before we discuss the details of our contributions and the implementation, we give an overview of how our multiverse debugger, MIO, works in practice. We will use a simple example to illustrate the different concepts of our multiverse debugger.

5.2.1 Example: A Light Sensor Program

Consider a simple program that reads a value from a color sensor, to measure the light intensity, and turns on an LED with a color corresponding to the value read. The left side of Figure 5-19 (2) shows the example program written in AssemblyScript (Battagline, 2021). Reading the value from the color sensor happens through the *colorSensor* function, and changing the color of the LED


```
import { pinMode, PinMode, PinVoltage, digitalWrite,
import { Pin } from "./sensor.board";

function init(): void {
    setupObb();
    pinMode(Pin.blue, PinMode.OUTPUT);
    pinMode(Pin.red, PinMode.OUTPUT);
}

export function main(): void {
    const pause: u32 = 1000;
    const threshold: u32 = 20;

    init();

    while (true) {
        if (colorSensor(Pin.sensor) > threshold) {
            digitalWrite(Pin.red, PinVoltage.HIGH);
            digitalWrite(Pin.blue, PinVoltage.LOW);
        } else {
            digitalWrite(Pin.blue, PinVoltage.HIGH);
            digitalWrite(Pin.red, PinVoltage.LOW);
        }

        delay(pause);
    }
}
```

Mock primitive call

color_sensor (0) = 17

color_sensor(0) = 0

Name	Type	Value
pc	i32	0x26a
global 0	i32	1
global 1	i32	2
global 2	i32	3
global 3	i32	0

Figure 5-19. Screenshot of the MIO debugger debugging a small light sensor program. Top (1): the debug operations; pause or continue, step back, step over, step into, step to the previous line, step to the next line, and update software. Left pane (2): source code. Top-right pane (3): the multiverse tree. Popup window (4): window to add mocked input. Bottom-right pane (5): the current mocked input. Bottom pane (6): general debugging information, such as the global and local variables, and the current program counter.

happens through the *digitalWrite* function. In an infinite loop, the program reads a value from the color sensor. If the value is below a threshold, the red LED is turned on, and otherwise the blue LED is turned on. At the end of the loop the program waits for 1000 milliseconds before starting the next iteration.

5.2.2 The Frontend of the Multiverse Debugger

Figure 5-19 shows the MIO multiverse debugger in action. While debugging, the program and the debugger backend run on a microcontroller connected to a computer running the debugger frontend, shown in the screenshot. The left pane (2) shows the program being debugged, in this case the light sensor program. In the top left corner (1), the user can see the debug operations available, which are in order; pause or continue, step back, step over, step into, step to the previous line, step to the next line, and finally, update the code on the microcontroller. The top-right pane (3) shows the multiverse tree, where each edge represents a WebAssembly instruction. In MIO all nodes are considered unique states, which means that loops are unrolled and no states will ever collapse into each other, making this graph always a rooted tree. MIO allows input values to be mocked using the *Mock* button, which triggers a popup window (4) where users can specify the mocked return value for a given input primitive. The *Create range* button can be used to add a range of new branches to the multiverse tree. The mocked primitives show up in the bottom-right pane (5), where they can also be removed again. The bottom pane (6) shows the global and local WebAssembly variables, and other program state information.

While debugging the light sensor program, the user can pause the program at any moment. At this point, the right pane will show the already explored paths of the multiverse tree. It is important to note that the multiverse tree does not correspond to a control flow graph, but shows every succeeding program state. Every edge in the tree represents the concrete execution of a single WebAssembly instruction, and every node represents the program state after that instruction. The multiverse tree in Figure 5-19 labels every node before the execution of a primitive with the primitive name, and labels the outgoing edges with the associated return value.

5.2.3 Debugging with the Multiverse Debugger

A developer can use the multiverse debugger to explore the execution of the light sensor program, using the debug operations in the top left corner, following normal debugging conventions. While the right panel continually updates to show where in the program's execution the user is. However, the multiverse debugger allows for more than just stepping forwards and backwards through the program. The user can also jump to any node in the multiverse tree by simply clicking on that node, and explore the program from that point onwards with the available debugging instructions.

5.2.4 Exploring the multiverse tree

The screenshot in Figure 5-19 shows the program paused at line 64, after reading a value from the color sensor. This value was 25, and if the user steps forward, the program will turn on the red LED (pin 45) as shown by the multiverse tree. In an earlier stage, the user actually read a value of 14. Using the debugger, it is possible to explore that execution path again, without having to recreate the situation where the sensor reads 14. This is done by clicking on any node on the execution path where the sensor reads 14, which is shown as a blue node with a white circle in it. If the user presses the *Jump* button, the debugger will traverse the blue-indicated path in the multiverse tree, from the current node to the selected node. When reading the sensor value, the debugger will use mock input actions, to override the normal behavior of the sensor and return the value 14.

The mocking mechanism is a crucial part of the multiverse debugger, since it allows the user to explore different execution paths without having to recreate the exact conditions that led to that path. It is therefore not only a part of the automatic traversal of the multiverse tree, but also a part of the manual exploration of the tree. While a user is stepping forwards through a program, and a non-deterministic input primitive is encountered, they can press the *Mock* button below the right pane instead of the *step* button. When doing so, window (4) shown in Figure 5-19 is used. This window allows the user to specify that the virtual machine should return a specific value when a primitive is called with certain arguments. This allows users to easily reproduce bugs that might not occur with the sensor values read from the current environment.

Now that we have established how our multiverse debugger works in practice, and its terminology, we will discuss the challenges in creating a multiverse debugger for microcontrollers.

5.2.5 Challenge C1: Inconsistent External State during Backwards Exploration

When moving backwards in time, the external state must also be restored. A multiverse debugger that allows for backwards exploration, but does not handle the external state, will be able to create impossible situations. For instance, if the light sensor program is paused when the red LED has just been turned on, the debugger can move backwards in time to when the light intensity is measured. Reading a different value as you step forwards again, this time, the blue LED could be turned on, even though the red LED is still on. This is an impossible situation, since the LEDs can never be on at the same time in a normal execution of the program. Stepping back over line 64, therefore, requires a compensating action, which turns off the red LED. Luckily, such actions can be defined for many different output operations (Laursen et al., 2018, Schultz, 2020).

5.2.6 Challenge C2: Exploring Non-deterministic Input in Multiverse Debuggers

When a program's execution path is determined by input from the environment, the multiverse debugger needs to explore different execution paths by traveling back in time and changing the input to the program. However, it is impractical for developers to cover all possible inputs, or to configure the environment exactly right for a specific execution path every time. To solve this problem, we propose a new approach to multiverse debugging that uses time travel debugging in combination with virtual machine instrumentation to mock input values. It is important that no impossible values are used for mocking, since they can lead to inaccessible program states.

5.2.7 Challenge C3: Keeping Track of the Program State Efficiently

In order for multiverse debugging to work on a microcontroller it has to keep track of the program state and its output effects on the external environment. However, tracking this information on microcontrollers is not feasible due to the limited memory capacity of these devices. Therefore, the MIO debugger is a remote debugger, which enables minimal interference with the microcontroller, and allows information to be stored on a more powerful computer. Even so, it is not feasible to take snapshots for every executed instruction as it would slow down the program significantly, and the size of the snapshot history would quickly become unmanageable.

In time travel debugging, this problem is usually solved by taking snapshots at regular intervals, called checkpoints. When moving backwards in time, the debugger can then jump to the nearest checkpoint, and replay the program's execution from that point.

To make a checkpointing system work correctly for multiverse debugging, we had to adopt a slightly different approach. Instead of only taking snapshots at regular intervals, we also take snapshots after each input or output action. By carefully choosing when snapshots are taken we can ensure the correctness of the replay mechanism. Using this approach, we can significantly reduce the run-time overhead, making multiverse debugging practically usable on microcontrollers.

5.3 A Multiverse Debugger for WebAssembly

In this section, we discuss the operation of our multiverse debugger through a small-step semantic defined over a stack-based language. We use WebAssembly as an example language, since it has a full and rigorous language semantics (Haas et al., 2017; Rossberg, 2023, 2019). However, our small-step rules include very few details specific to WebAssembly. Therefore, we believe that the principles of our multiverse debugger can be applied to any stack-based language, with minimal effort.

5.3.1 Requirements for I/O operations

While our multiverse debugger can deal with a large set of I/O operations, there are some limitations that make it impossible to support certain I/O operations. Intuitively, our multiverse debugger supports non-deterministic input primitives as long as the range of possible input values is known. Output primitives are supported as long as they are atomic, and are deterministically reversible, i.e. after reversing an output operation the environment will be in the same state as before applying the operation.

Input primitives Input primitives are allowed to be non-deterministic as long as the *range* of the input primitives is known, i.e. a temperature sensor might have a range between -20 degrees till 160 degrees. Knowing the range of our input primitives is important so that the debugger can be instrumented to only sample values that can actually be observed during normal execution.

Output primitives First, we require output primitives to be synchronously and atomically, i.e. all side effects from the operation have to be fully completed during the call of the operation in the virtual machine. Second, for a given execution of the I/O operation, there must be a *deterministic compensating action*. This compensating action undoes the effects of the forward execution, bringing the environment back to a state before executing the actions.

Predictable dependencies We assume *predictable* dependencies of the I/O operations are known, for example consider a setup where an LED is directly pointed towards a light sensor. During regular execution, turning the LED on will directly influence the possible values which can be read, i.e. there is a dependency between the output pin and the possible sensor values which can be read. Our MIO debugger, has initial support for expressing such simple dependencies, in the semantics however, we take abstraction and assume that sensor values are independent.

5.3.2 WebAssembly Language Semantics

WebAssembly is a low-level, portable binary code format designed for safe and efficient execution on various platforms. Its formal semantics are grounded in a stack-based virtual machine, where instructions and values operate on a single stack with strict typing to guarantee fast static validation. We base our formalisation on the semantics from the original WebAssembly paper by Haas et al. (2017), where the core semantics include structured control flow (blocks, loops, and conditionals) and memory management via linear memory. WebAssembly intentionally excludes external interface definitions, including I/O operations, to minimize its dependence on platform-specific details. This design choice enables us to deliberately sculpt the I/O operations so that they are deterministically reversible, a necessary condition for multiverse debugging.

The execution of a WebAssembly program is defined by a small-step reduction relation, denoted as $\hookrightarrow_i$ where, i refers to the index of the currently executing module. The relation $\hookrightarrow_i$ is defined over a configuration $\{s; v^*; e^*\}$, with global store s, local values v^*, and the current stack of instructions

$$
\begin{array}{ll}
\text{(WebAssembly Program state)} & K ::= \{s, v^*, e^*\} \\
\text{(Global store)} & s ::= \{\mathsf{inst}\ \mathsf{inst}^*, \mathsf{tab}\ \mathsf{tabinst}^*, \mathsf{mem}\ \mathsf{meminst}^*, \ \mathsf{prim}\ \mathrm{P}\} \\
\text{(Primitive table)} & \mathrm{P} ::= p^* \\
\text{(Primitive)} & \mathrm{p} = f : v^* \rightarrow \{\mathsf{ret}\ v, \mathsf{cps}\ r\} \\
\text{(Compensating action)} & \mathrm{r} = f : \varepsilon \rightarrow \varepsilon
\end{array}
$$

Figure 5-20. The configuration for the reversible primitives embedded in the WebAssembly semantics from the original paper by Haas et al. (2017), the differences are highlighted in gray. *Top:* The WebAssembly semantics extended with a primitive table. *Bottom:* The signatures of primitive and their compensating actions.

e^*. The reduction rules take the form $\{s; v^*; e^*\} \hookrightarrow_i \{s'; v'^*; e'^*\}$. Important for our semantics, the global store s contains instances of modules, tables, and memories. The global store allows access to any function within a module instance, denoted as $s_{\text{func}}(i, j)$, where i represents the module index and j corresponds to the function index.

5.3.3 Extending WebAssembly with Primitive I/O Operations

Since multiverse debuggers explore all possible execution paths, they have to be able to reproduce the program's execution, even when non-determinism is involved. It is therefore important to consider where non-determinism is introduced in the program, and how it can be handled. Since the WebAssembly semantics on their own are fully deterministic, we can choose precisely where non-determinism is introduced in our system. In the context of microcontrollers, non-deterministic input is unavoidable. Our system therefore limits non-determinism exclusively to the input. This means that each branch in the execution tree can therefore be traced to a different input value. The output primitives in MIO, on the other hand, are deterministic in terms of the program state. Section 5.3.7 discuss how the debugger can reproduce non-deterministic input reliably through input mocking.

We do not consider parallelism as a source of non-determinism, since this has been examined thoroughly by the original paper on multiverse debugging by Torres Lopez et al. (2019).

5.3.3.1 Primitive I/O operations

We extend WebAssembly with a set of primitives P that fulfil the prerequisites outlined above. Figure 5-20 shows the definition of the primitives in WebAssembly. Each primitive in P can be identified by a unique index j, similar to the function indices in WebAssembly, and looking up primitives is done through the global $P(j)$. When calling a primitive, it returns both the return value v and the compensating action r. The function r compensates, or reverses, the effects of the primitive, but takes no arguments and returns

Non-Deterministic Input Primitives

$$\frac{P(j) = p \qquad p \in P^{\text{In}} \qquad v \in \lfloor p(v_0^*)_{\text{ret}} \rfloor}{\{s; v^*, v_0^*(\text{call } j)\} \hookrightarrow_i \{s; v^*, v\}} \text{ input-prim}$$

$$\frac{P(j) = p \qquad p \in P^{\text{Out}} \qquad \lfloor p(v_0^*)_{\text{ret}} \rfloor = v}{\{s; v^*, v_0^*(\text{call } j)\} \hookrightarrow_i \{s; v^*, v\}} \text{ output-prim}$$

Figure 5-14. Extension of the WebAssembly language with non-deterministic input primitives.

nothing as indicated by its type $\epsilon \rightarrow \epsilon$. There is no need for any arguments, since the compensating action is generated uniquely for each execution of the primitive.

5.3.3.2 Forwards Execution of Primitives

Given the definition of the primitives in P, we can define the forwards execution of the primitives in WebAssembly, as shown in Figure 5-14. Non-determinism is introduced exclusively through the *input-prim* rule, which is used to evaluate input primitives. The evaluation of the primitive p non-deterministically returns a value v from the codomain of the primitive function, $\lfloor p(v_0^*)_{\text{ret}} \rfloor$. Here, the rule simply discards the compensating action, and places the return value of the primitive on the stack. The *output-prim* rule works analogously, except the evaluation produces its return value deterministically. Note that compensating actions $p(v_0^*)_{\text{cps}}$ are not used for the regular forward execution, but are crucial when moving backwards in time. In the next sections, we show how the compensating actions are used during multiverse debugging.

5.3.4 Configuration of the Multiverse Debugger

Using the recipe for defining debugger semantics from Torres Lopez et al. (2019), we can define our multiverse debugger on top of the extended WebAssembly semantics presented in Section 5.3.3. Figure 5-21 shows the configuration of the multiverse debugger for WebAssembly with input and output primitives. The program state in the underlying language semantics is labeled with an iteration index n, which corresponds to the number of steps in the underlying semantic since the start of the execution, or the depth in the multiverse tree. The debugger state dbg contains the execution state of the program, incoming debug message msg, mocked input mocks, the program state K_n, and the snapshot list S^*. The snapshots S^* are a cons list of snapshots S_n, containing the program state K_n and the compensating action p_{cps}. The

(Debugger state)	$dbg ::= \langle es, msg, mocks, K_n \mid S^* \rangle$
(Execution state)	$es ::= play \mid pause$
(Incoming messages)	$msg ::= \varnothing \mid step \mid stepback \mid pause \mid play \mid mock \mid unmock$
(Program state)	$K ::= \{s, v^*, e^*\}$
(Overrides)	$mocks ::= \varnothing \mid mocks, (j, v^*) \mapsto v$
(A snapshot)	$S_n ::= \{K_m, p_{\{cps\}}\}$
(Snapshots list)	$S^* ::= S_0 \cdot \ldots \cdot S_{\{n-1\}} \cdot S_n$
(Starting state)	$dbg_{\text{start}} ::= \langle pause, \varnothing, \varnothing, K_0 \mid \{K_0, E\} \rangle$
(Empty action)	$r_{nop} ::= \lambda().\ nop$

Figure 5-21. The multiverse debugger state for WebAssembly with input and output primitives.

Figure 5-22. The small-step rules describing forwards exploration in the multiverse debugger for WebAssembly instructions without primitives.

rules of the debugger semantics, presented in the following sections, will show how the snapshot list is extended—and how the snapshots are used to travel back in time.

Mocked inputs are stored as a key value pairs, where the index identifying the input primitive j, and the list of argument values v^* are mapped to the overriding return value v. The key value map is represented here as a partial function, which compares lists of values v^* element-wise. For any key that is not defined in the map, we write mocks$(j, v^*) = \epsilon$.

The starting state of the debugger dbg$_{\text{start}}$ is defined as the paused state with no incoming or outgoing messages, an empty mocks environment, the initial program state K_0, and a snapshot list containing only the initial snapshot $S_0 = \{K_0, r_{\text{nop}}\}$. Here r_{nop} is the empty action, which takes no arguments and returns nothing. This function indicates that no compensating action is needed.

5.3.5 Forwards Exploration in Multiverse Debuggers

Figure 5-22 shows the basic small-step rules for stepping forwards in the multiverse debugger, without input and output primitives. These rules allow the debugger to explore traditional WebAssembly programs, without any non-deterministic input or output. For clarity, we use several shorthand notations in the rules. We use the notation $(K_n \hookrightarrow_i K_{n+1})$ to say that the program state K takes a step to the program state K' in the underlying language semantics, where $K' = K_n + 1$. The notation (non-prim K) is used to indicate that the program state K is not a primitive call, or more fully, it is not the case that $K = \{s; v^*; v_0^*(\text{call}; j)\} \wedge P(j) = p$. We describe the rules in detail below.

Figure 5-23. The small-step rules describing forwards exploration for input and output primitives in the multiverse debugger for WebAssembly, without input mocking.

run The rule for running the program forwards in the underlying language semantics. The debugger takes a step in the underlying language semantics ($K_n \hookrightarrow_i K_{n+1}$) as long as the execution state is *play*, there are no incoming or outgoing messages, and the program state is not a primitive call. While running in this way, the snapshot list S^* remains unchanged.

step-forwards When the debugger receives the *step* message, it takes one more step ($K_n \hookrightarrow_i K_{n+1}$), and transitions to the *pause* state if it was not already paused.

pause When the debugger receives a *pause* message in the *play* state, it transitions to the *pause* state. Note that afterwards, all the run rules are no longer applicable.

play The rule for continuing the execution. When the debugger receives the *play* message in the *pause* state, the execution state transitions to the *play* state.

The rules in Figure 5-23 are the minimal set of rules for stepping forwards when the next instruction in the program state K_n is a primitive call. The rules also describe the snapshotting behavior of the multiverse debugger, which is needed to travel back in time. We describe the run rules in detail below. The step rules are identical to the run rules, but they transition to the paused state after taking a step, and are triggered by the *step* message. These rules can be found in Appendix G.3.

The rules described here form the backbone of the multiverse debugger, and already allow for a multiverse debugger that can explore the execution of a program forwards in time. It is important to note that the debugger always takes snapshots when a primitive call is made. Only when this primitive is an output primitive, can the state of the environment change. On the other hand, only input primitives can introduce new branches to the multiverse tree. In the next section, we discuss how the multiverse debugger can explore the execution of a program backwards in time.

5.3.6 Backwards Exploration with Checkpointing

The multiverse debugger is also a time travel debugger, which means it can move backwards in time. It does this by restoring the program state from a previous snapshot, and then replaying the program's execution from that point. Figure 5-24 shows schematically the way the debugger steps back in time using the snapshots. In the situation depicted by the figure, the

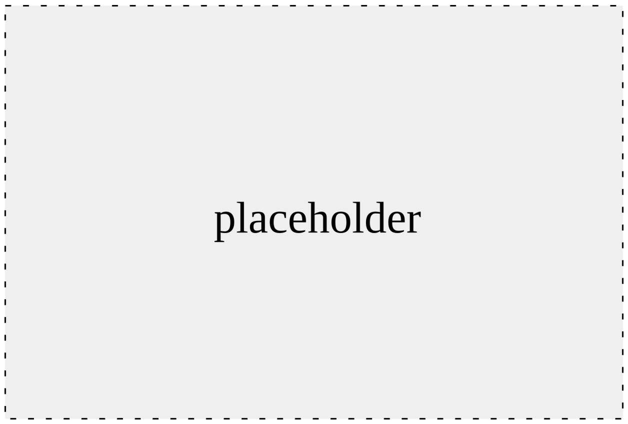


Figure 5-24. Schematic of how the *step-back-compensate* rule works. Starting from state K_m, the dotted arrow shows how the debugger jumps to the previous state K_n, and compensates the output primitive with $r'()$, while the full arrows show the normal execution. Top right: the snapshots before. Top left: the snapshots after.

debugger has just stepped over an output primitive and added a snapshot to the snapshot list. Since the debugger will now step back over the execution of this output primitive, its effects must be reversed with the compensating action r'' in the last snapshot. This is shown with the dashed arrow. As part of this jump back in time, the snapshot containing the compensating action r'' is removed from the snapshot list and the program state K_n from the next snapshot is restored.

Since snapshots are only added when the program performs a primitive call, the second to last snapshot in the list was taken after the previous primitive call resulting in state K_n. This means, that after restoring the internal virtual machine state, the program is now at the point right after the previous primitive call. Starting from this point, the debugger can replay the program's execution forwards to K_{m-1}, which will not include any primitive calls. This means the steps will be deterministic, and will not change the external environment. This corresponds with the full arrow at the bottom of the figure. Next to it is shown the snapshot list after the step back, which now only contains the snapshot of the state K_n.

In the outlined scenario the last transition in the chain, from K_{m-1} to K_m, performs an output action. However, if this transition is a standard WebAssembly instruction instead, no compensating action will be performed. Instead, the debugger will immediately restore the virtual machine state to K_n and replay the program's execution forwards to K_{m-1}. In this case, none of the snapshots will be removed. This enables the debugger to continue stepping back in time.

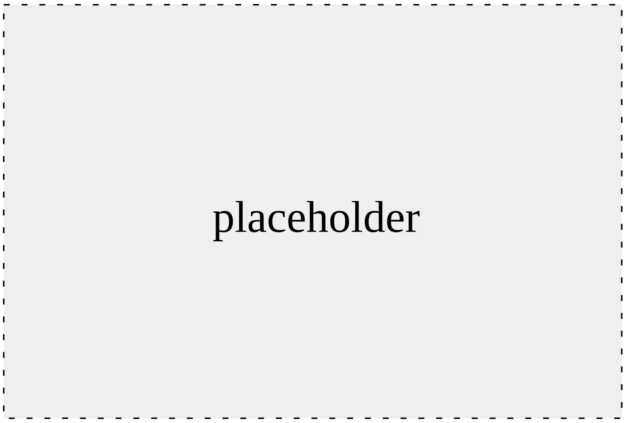


Figure 5-25. The small-step reduction rule for stepping backwards in the multiverse debugger.

Each of the two outlined scenarios correspond with a rule in the multiverse debugger semantics, shown in Figure 5-25. The first scenario where the effects of an output primitive is reversed, is described by the *step-back-compensate* rule. The second scenario where the last transition is a standard WebAssembly instruction, is described by the *step-back* rule. We describe the rules in detail below.

step-back The rule for stepping back in time. When the debugger receives a *step back* message, the debugger restores the external state from the last snapshot in the snapshot list, which is not the current state. The debugger then replays the program's execution from that point to exactly one step ($K_n \hookrightarrow_i^{m-n-1} K_{m-1}$) before the starting state. Since the restored snapshot remains in the past, it is kept in the snapshot list, to allow for further backwards exploration.

step-back-compensate The rule for stepping back in time when the last transition was a primitive call. This is always the case when the current state is K_m is part of the last snapshot. When the debugger receives a *step back* message, the debugger performs the compensating action r' from the last snapshot in the snapshot list, which reversed the effects of the last primitive call. Then, the debugger restores the external state K_n from the second to last snapshot in the snapshot list. The debugger then replays the program's execution from that point to exactly one step before the starting state. The last snapshot is removed from the snapshot list, since it now lies in the future.

In the case, where no primitive call has yet been made, the snapshot list contains exactly $\{K_0, r_{\text{nop}}\}$, as defined by $\text{dbg}_{\text{start}}$, which the *step-back* rule can jump to. If the current state is K_0, stepping back is not possible. Specifically,

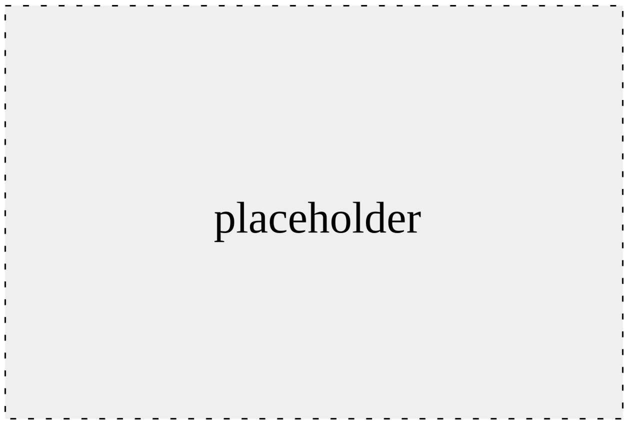


Figure 5-26. The small-step rule for mocking input in the MIO debugger, only including the step rule. The analogous rule for when the debugger is not paused (*run-mock*) is shown in Appendix G.3.

the *step-back* rule is not be applicable, since m and n are both zero, and the *step-back-compensate* rule requires the snapshot list to contain at least two snapshots.

5.3.7 Instrumenting Non-deterministic Input in Multiverse Debuggers

In order to replay execution paths in the multiverse tree accurately, the multiverse debugger needs to be able to override the input to the program. Mocking of input happens through the key value map mocks shown in Figure 5-21. New values can be added to the map using the *register-mock* rule, and existing values can be removed using the *unregister-mock* rule. Whenever the debugger encounters an input primitive call, it will always check the mocks map for an overriding value. If a value is found, the debugger will replace the call to the primitive with the mock value v. This is done by the *step-mock* rule.

register-mock The rule for registering a new mock value in the multiverse debugger. When the debugger receives a message mock(j, v^*, v), the debugger will update the entry for (j, v^*) in the mocks environment to v. If an entry already exists in the environment, the rule will override the existing value.

unregister-mock The rule for unregistering a mock value in the multiverse debugger. When the debugger receives a message unmock(j, v^*), the debugger will remove the mock value from the mocks map. If no value is found in the environment, the rule will have no effect.

step-mock The *step-mock* rule for stepping forwards in the multiverse debugger when an input primitive call is encountered. If the input primitive call is found in the mocks map, the debugger will replace the call with the mock value v.

The program state is then updated to the new program state K_{n+1}, and a new snapshot is added to the snapshot list. The snapshot includes the new program state and the empty compensating action r_{nop}, since no compensating action is needed for input primitives.

5.3.8 Arbitrary Exploration of the multiverse tree

With the semantics of input mocking in place, we now have the entire multiverse debugger semantics for WebAssembly with input and output primitives. In this section, we discuss how the multiverse debugger can be used to explore different universes. This can be done by Algorithm 5-3. When the debugger jumps from a state K_m to a state K_n, the debugger will find the smallest common ancestor of K_m and K_n, or the join. The debugger will then step backwards from K_m to the join. We use the notation $\hookrightarrow_r$ to indicate that the debugger is reversing the execution, it is equivalent to a debugging step $\hookrightarrow_{d,i}$ that only uses the *step-back* and *step-back-compensate* rules. In the final step of the algorithm, execution is replayed from the join to K_n using the *step-mock* rule whenever it encounters a non-deterministic primitive call.

```
1 Require the current program state $K_m$ and the target program state $K_n$ and the snapshot
  list $S^*$.
2 $K_{\text{join}} \leftarrow \text{find\_join}(K_m, K_n)$
3 while $\text{dbg}_{\text{current}}[K] \neq K_{\text{join}}$ do
4     $\text{dbg}_{\text{current}} \hookrightarrow_r \text{dbg}_{\text{next}}$
5     $\text{dbg}_{\text{current}} \leftarrow \text{dbgnext}$
6 while $\text{dbg}_{\text{current}}[K] \neq K_n$ do
7     $\text{dbg}_{\text{current}} \hookrightarrow_{d,i} \text{dbg}_{\text{next}}$
8     $\text{dbg}_{\text{current}} \leftarrow \text{dbg}_{\text{next}}$
```

Algorithm 5-3. The algorithm for traveling to any position in the multiverse tree.

Figure 5-27 illustrates the algorithm for jumping to an arbitrary state, when the user clicks on a node on another branch in the multiverse tree. The figure shows a possible multiverse tree for a program where the second and third instruction are input primitives. The program has executed two input primitives in a row, and the debugger has explored some of the possible inputs. Each node in the figure is labeled with the program state and possible compensating action, where $r_{\{\text{nop}\}}$ indicates that no compensating action is needed For clarity, the external state are also numbered. The figure shows clearly that the external state only changes after a primitive call. The current state is K_5, and the debugger wants to jump to $K_{4'}$. Per the algorithm, the

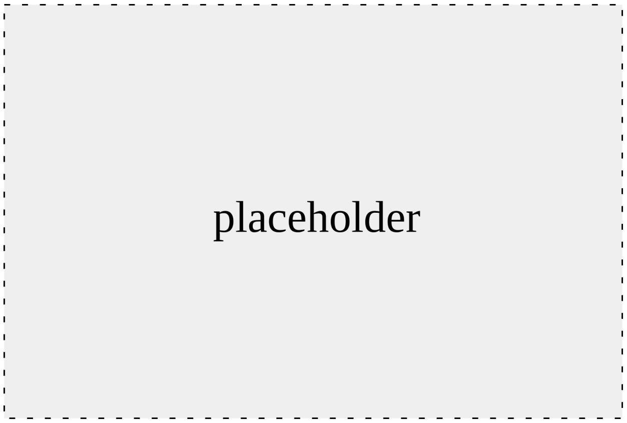


Figure 5-27. Schematic of how the multiverse debugger can jump to any arbitrary state in the past, using the *step-back* and *step-mock* rules. For the arbitrary jump from state K_5 to $K_{4'}$, the join K_1 is underlined and shown in blue. Top right: the list of snapshots before the arbitrary jump. Bottom: the execution path from K_5 to $K_{4'}$. Steps with the *step-back* and *step-back-compensate* rules are shown as $\hookrightarrow_r$.

debugger finds the join of the two states, which is K_1. The debugger then replays the execution from K_5 to K_1 in reverse order, using the *step-back* and *step-back-compensate* rules. It is important that the debugger steps back one instruction at a time, to ensure that the external state is correctly restored. From the join K_1, the debugger replays the execution to $K_{4'}$ in the forward order, using the *step-mock* rule whenever it encounters a non-deterministic primitive call. This ensures that the jump deterministically follows the exact execution path, thereby ensuring that the external state is correctly restored.

5.3.9 Correctness of the Multiverse Debugger Semantics

Given the small-step semantics, we can prove the correctness of the MIO debugger in terms of soundness and completeness. The soundness theorem states that for any debugging session ending in a certain state, there also exists a forwards execution path in the underlying language semantics to that state. A debugging session is seen as any number of debugging steps starting from the initial debugging state $\text{dbg}_{\text{start}}$. The completeness of the debugger means that the debugger can always find a path in the multiverse tree that corresponds to a path in the underlying language semantics. Together, these properties ensure that the debugger is correct in terms of its observation of the underlying language, and will never observe any inaccessible states. For brevity, we only provide a sketch of the proofs here, but the full proofs can be found in Appendix G.4.

The proof for debugger soundness proceeds by induction over the number of steps in the debugging session. In the base case, where the debugging session consists of a single step, the proof is trivial since the step starts from the initial state. In the inductive case, the proof proceeds very similarly, the only non-trivial cases are those for stepping backwards and mocking.

The proof for completeness follows almost directly from the fact that for every transition in the underlying language semantics, the debugger can take a corresponding step. For non-deterministic input primitives, we can step to the same state with the *register-mock* and *step-mock* rules.

Together the debugger soundness and completeness theorems ensure that the multiverse debugger is correct in terms of its observation of the underlying language semantics. However, it gives us no guarantees about the correctness of the compensating actions, and the consistency of external effects during a debugging session. Due to the way effects on the external environment are presented in the MIO debugger semantics, we can define the entire effect of a debugging session of regular execution, both as ordered lists of steps that have external effects. There are only two options, the output primitive rules, and the rule that applies the compensating action.

The proof of this theorem is based on the fact that our multiverse debugger is a rooted acyclic graph, and a debugging session is a walk in this tree starting from the root, which can include the same edge several times. Any such walk in a tree can be constructed by adding any number of random closed walks to the path from the root to the final node. Such closed walks are null operations in terms of their effect on the external state. This leaves only the forward steps of the minimal path to be considered, meaning the external effects of a debugging session are always the same as those of the regular execution of the program.

5.4 The MIO debugger

We have implemented the multiverse debugger described above in a prototype debugger, called the MIO debugger. The MIO debugger is built on top of a WebAssembly runtime for microcontrollers, called WARDuino (Lauwaerts et al., 2024). WARDuino is written in C++, includes primitives for controlling hardware peripherals, and has support for traditional remote debugging. Our prototype implementation builds further on its virtual machine and the remote debugging facilities but needed to be extended significantly in order to support all the basic operations for multiverse debugging: smart snapshotting, mocking of primitives and reversible actions. Additionally, we created a high-level interface which implements the message passing interface described in Section 5.3 as messages in the remote debugger of WARDuino. On top of this interface we built a Kotlin application for debugging AssemblyScript programs on microcontrollers running WARDuino. This application keeps track of the program states, and shows them as part of the multiverse tree, as shown in Figure 5-19 from Section 5.2. Algorithm 5-3 for arbitrary jumping, is implemented at the level of the Kotlin application using the message passing interface of the remote debugger. Finally, the MIO prototype also has support for expressing simple dependencies, so that the mocking of the various sensor values can be limited depending on state of the output pins.

5.4.1 Output: Reversible Primitives

Primitives in WARDuino are implemented in the virtual machine using C macros. In order to implement reversible primitives, we have extended the existing macros with two new macros; one defines how the external state effected by the primitive can be captured, and the other defines the compensating action given this captured state. When stepping back over a primitive, the compensating action looks at the state captured after the previous primitive call, and restores this external state. This is the same as undoing the effects of the last primitive call.

definecolor{mRed}{rgb}{1,0.4,0.4} definecolor{mBlue}{rgb}{0.4,0.4,1} definecolor{mGreenGraph}{RGB}{102,255,110} definecolor{mGreen}{rgb}{0,0.6,0} definecolor{mGray}{rgb}{0.5,0.5,0.5} definecolor{mDarkGray}{rgb}{0.2,0.2,0.4} definecolor{mPurple}{rgb}{0.58,0,0.82} definecolor{backgroundColour}{rgb}{1,1,1}

lstdefinestyle{CStyle}% { basicstyle=footnotesizettfamilylinespread{0.7}% , captionpos=b% , identifierstyle=% , backgroundcolor=color{backgroundColour} , commentstyle=color{mGray} , keywordstyle=bfseriescolor{black} , numberstyle=tinyttfamilycolor{mGray} , stringstyle=color{mRed} , keywordstyle=color{mBlue}bfseries% reserved keywords , keywordstyle=[2]color{mRed}% traits , keywordstyle=[3]color{mBlue}% primitive types , keywordstyle=[4]color{mRed}% type and value constructors , keywordstyle=[5]color{mBlue}% macros , columns=spaceflexible% , keepspaces=true% , showspaces=false% , showtabs=false% , showstringspaces=false% , numbers=left% , numbersep=5pt% }

begin{figure} begin{minipage}[t]{.44textwidth} begin{lstlisting}[language=C++, style=CStyle,escapechar=‘] def_prim(rotate, threeToNoneU32) { int32_t speed = arg0.int32; int32_t degrees = arg1.int32; int32_t motor = arg2.int32; pop_args(3); auto encoder = encoders[motor]; encoder->set_angle(“ encoder->get_angle() + de-

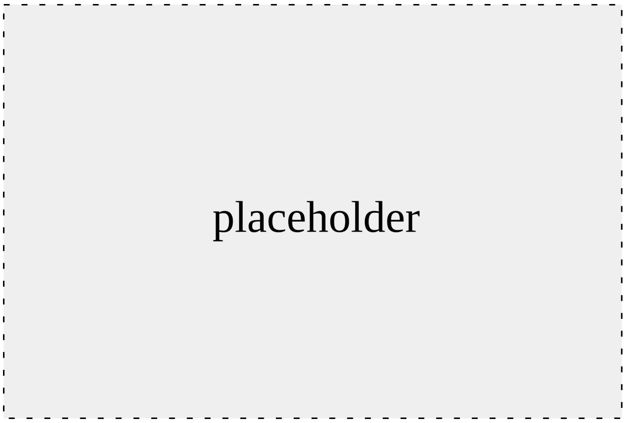


Figure 5-28. *Left:* The implementation of the *rotate* primitive. *Right:* The implementation of the compensating action for the *rotate* primitive, in the MIO debugger.

grees’‘); return drive(motor, encoder, speed);’‘ }end{lstlisting} end{minipage} hfill begin{minipage}[t]{.50textwidth} begin{lstlisting}[language=C++, style=CStyle,escapechar=’,firstnumber=10] def_prim_serialize(rotate) { for (int m = 0; m < MOTORS; i++) { external_states.push_back(new MotorState(m, encoders[m]->angle()));‘’ }}

def_prim_reverse(rotate) { for (IOState s : external_states) { if (isMotorState(s)) { int motor = stoi(s.key); auto encoder = encoders[motor]; encoder->set_angle(s.degrees);‘’ drive(motor, encoder, STD_SPEED); }}} end{lstlisting} end{minipage} caption{} label[listing]{fig:motor-impl} end{figure}

To illustrate the implementation of reversible primitives, we will use the example of the *rotate* primitive, which rotates a servo motor for a given number of degrees. The forwards implementation is shown on the left side of Figure 5-28. To move the motor a given number of degrees the primitive first sets the target angle of the motor encoder, this happens on line ref{line:encode}. The motor encoder is used to track the current motor angle, as well as the absolute target angle, which can be set with the *set_angle* method. To rotate the motor a number of degrees relative to its current position, the primitive adds the degrees to the current motor angle (line ref{line:relative}). Once the target angle is set, the primitive drives the motor to that angle using the *drive* method, as shown on line ref{line:drive}.

The implementation of the compensating action for the *rotate* primitive is shown on the right side of Figure 5-28. First, the *def_prim_serialize* macro captures the external state. For each motor, the current angle of the motor is stored along with its index, as shown on line ref{line:serialize}. Second, the *def_prim_reverse* macro compensates the primitive by moving all motors back to the angles captured in the previous snapshot. The angles captured by the *def_prim_serialize* macro are absolute target angles. The compensating action moves the motors back to these angles by first setting the target angle, as shown on line ref{line:set-angle}. It then uses the same *drive* function to move the motor.

5.4.2 Input: mocking of primitives

The input mocking is implemented analogous to the debugger semantics, by adding a map to the in the virtual machine state. This map is used to store the mocked values for the input primitives, which are received by a new debug message in the remote debugger. In line with the semantics, there is also a new debug message to remove a mocked value from the map. Currently, the map only supports registering primitive calls with their first argument. %but this can easily be extended to support multiple arguments. This is sufficient for the current input primitives to be mocked, without any changes to their implementation.

The virtual machine will check the map of mocked values for every primitive call. The prototype includes two input primitives that can be mocked in this way, the *digitalRead* primitive which reads the value of a digital pin, and the *colorSensor* primitive which reads a value from a uart color sensor. The digitalRead primitive enables the user to mock the value of a digital pin, and thereby the behavior of a wide range of possible peripherals. However, the range of possible input values is not always known statically, as it may be influenced by the output effects of the program. To handle this, the MIO debugger includes initial support for predictable dependencies that can be defined as simple conditions, for example, *"when the value of a digital pin n is x, then input primitive p with arguments m will return the value c"*.

5.4.3 Performance: Checkpointing

To reduce memory usage, the MIO debugger only stores the snapshots at certain checkpoints. The semantics of MIO only takes snapshots after a call to a primitive, the prototype implementation follows this checkpointing policy precisely. As shown by the debugger semantics and the proof, this is the minimum number of snapshots needed to enable backwards and forwards exploration of the multiverse tree. To further reduce the performance impact on the microcontroller, snapshots are received and tracked by the desktop frontend of the MIO debugger. To have minimal traffic between the debugger backend and frontend, snapshots after primitive calls are sent automatically

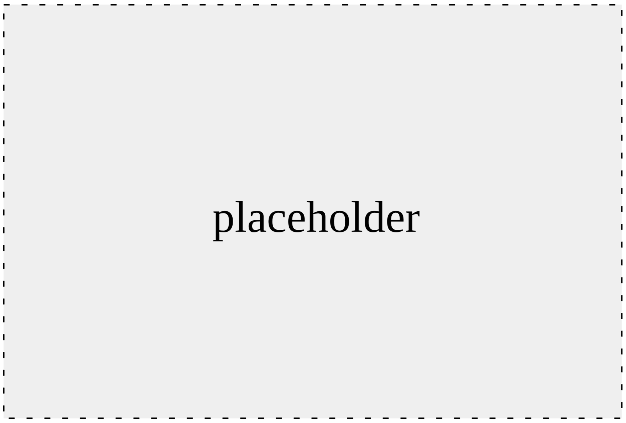


Figure 5-29. Comparison of execution time of *no snapshotting* with *snapshotting for every instructions*, and different checkpointing intervals; *every 5, 10, 50, and 100 instructions*. The performance overhead is shown as execution time relative to the execution time when taking no snapshots. Left: Comparison of all checkpointing policies, snapshotting, and no snapshotting. Right: Comparison of all checkpointing policies with no snapshotting. The averages are taken over 10 runs of the same program.

to the frontend. Alternatively, the debugger frontend can request snapshots at will through the remote debugger interface.

5.5 Evaluation

To validate that our checkpointing strategy is performant enough for apply multiverse debugging on microcontrollers we performed a number of experiments. All experiments were performed on an STM32L496ZG microcontroller running at 80 MHz. This microcontroller was connected to a laptop running the MIO debugger frontend that communicates with the microcontroller.

5.5.1 Forward execution with checkpointing

The first experiment evaluates the performance impact of checkpointing on the execution speed. We measured the execution time of a fixed number of instructions, when taking no snapshots, taking a snapshot every instruction, and for snapshotting after different intervals (5, 10, 50, or 100 instructions), as shown in Figure 5-29. To reduce the impact of variable unknown factors, the program executed by the virtual machine includes no primitive calls. Specifically, this program checks for each integer from 1 to 13, 374, 242 if they are prime. Because this program has no primitive calls, the VM will only take snapshots at fixed intervals which are determined by the frontend.

The left plot shown in Figure 5-29, gives the time it took to execute up to 1250 instructions for each snapshot policy relative to taking no snapshots. Since snapshotting every instruction is so much slower, we added the right plot showing the same results, but without snapshotting at every instruction. For such small numbers of instructions, the execution time without any debugger intervention, remains roughly the same, taking on average 222.7ms. These results are shown in red. In contrast, when taking snapshots after every executed instruction, the execution time increases dramatically. For 1250 instructions it takes on average 19 seconds, which is around 85 times slower. For only 250 instructions the execution time increases seventeen-fold, to 3.9 seconds.

Once checkpointing is used the overhead reduces significantly. When taking snapshots every five instructions, the virtual machine only needs 4 seconds to execute 1250 instructions. Per instruction this results in an execution that is only 17.9 times slower. Taking a snapshot every 10 instructions results in a total execution time of 2.1 seconds. This results in a slowdown of factor 9.5. When taking snapshots every 50 instructions, the slowdown lowers to a factor of 2.7. Going up to a hundred instructions every snapshot, this becomes only a factor 1.9.

This initial benchmark of the checkpointing strategy shows that the performance overhead can be greatly reduced by reducing the number of snapshots taken. Yet, execution times are still significantly slower than without any snapshotting. This is due to the fact that the current prototype has not yet been optimized for performance. The prototype only uses a simple run-length encoding of the WebAssembly memory to reduce the size of the snapshots. In future improvements, the snapshot sizes could be reduced greatly by only communicating the changes compared to the previous snapshot. However, in practice the performance is already sufficient to provide users with a responsive debugger interface as we illustrate in the online demo videos, which can be found here[6]. The example we highlight later in Section 5.5.3, requires on average snapshot every 37 instructions. This reduces overhead sufficiently to have a responsive debugging experience for users. Additionally, the I/O operations by comparison typically take much longer to execute, a single action easily taking several seconds.

5.5.2 Backwards execution

The graphs in Figure 5-29 only show part of the picture, where the less snapshots are taken, the better the performance. Unfortunately, there is no

[6]Full link: https://youtube.com/playlist?list=PLaz61XuoBNYVcQqHMAAXQNf8fz5IAMahe&si=HNrKY9YzqDFadATN

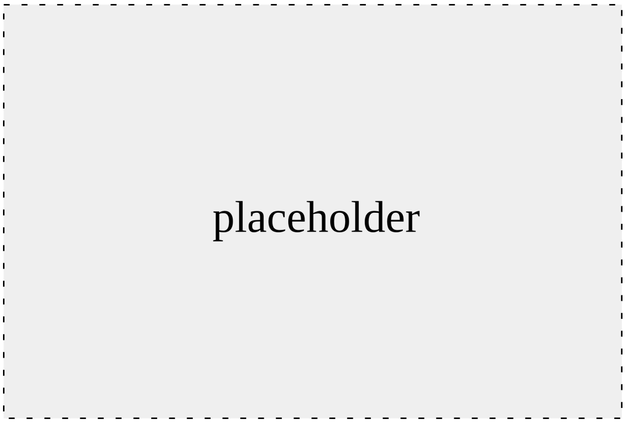


Figure 5-30. Plot showing the average time to step back as the number of instructions requiring re-execution increases in increments of one thousand. Averages are calculated over 10 runs of the same program.

such thing as a free lunch, and while only taking one snapshot at the start of the program and never again, would result in the lowest possible overhead for forwards execution, this is not the case for backwards execution. In that case stepping back would always have to re-execute the entire program. Clearly, the further apart the snapshots, the longer it will take to step back. To illustrate this trade-off, we examined the impact of the number of re-executed instructions on stepping back speed.

Figure 5-30 shows the average time it takes to step back as the number of instructions requiring re-execution increases in increments of one thousand. The averages are calculated over 10 runs of the same program used in the previous section. When executing only a handful of instructions, the time to step back is dominated by the communication latency between the microcontroller and the debugger frontend. On average, this results in a minimal time of 468ms to step back. Between one thousand and 30 thousand re-executed instructions, the time to step back increases linearly by roughly 11ms per a thousand instructions.

Our analysis of the checkpoint strategy's impact on stepping back shows that the overhead is minimal. The prototype is able to re-execute 30 thousand non-I/O instructions in around one second. Compared to the overhead of checkpointing on forwards execution (see Figure 5-29), we can safely conclude that in practice the overhead on backwards execution is negligible. This is further evidenced in our demo videos, where developers mostly have to wait for physical I/O actions to complete, and stepping back is otherwise instantaneous.

5.5.3 Use case: Lego Mindstorms color dial

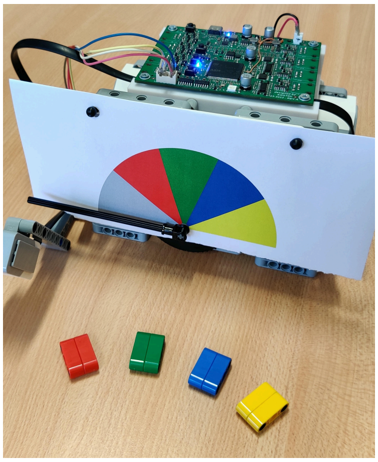
To illustrate the practical potential of MIO and its new debugger approach, we present a simple reversible robot application using Lego Mindstorms components components. However, not just microcontroller applications may benefit from our novel approach, there are many application domains where output is entirely in the form of digital graphics, which are more easily reversible—such as video games, simulations, etc. Nevertheless, to highlight the potential of the approach we demonstrate the MIO debugger using small physical robots and other microcontroller applications, as this is a more challenging environment for multiverse debugging. Using the digital input and motor primitives described in Section 5.4, we developed a color dial, as a simplified application. We developed this example alongside a few others to further demonstrate the usability of the MIO debugger[7], and have created demo videos for a few of the examples, which can be found online[8].

The color dial application works as follows, the robot has a color sensor that can detect the color of objects. Depending on the color seen by the sensor, a single motor will move the needle on the dial to the location indicating the color seen by the sensor. We built the dial using LEGO Mindstorms components (Ferreira Ruiz et al., 2024) as shown on the left of Figure 5-31. The right-hand side of Figure 5-31 shows the infinite loop that controls the robot, written in AssemblyScript. In this loop, the robot will continually read sensor values from the color sensor. While doing so it will move the needle of the dial to the correct position indicating the current color seen by the sensor. The needle is only moved if the color sensor sees a value different from what the dial is currently indicating. The relative amount that the needle needs to move is calculated by taking the difference between the current color the needle is pointing at and the new color.

The program for the color dial uses the reversible primitive *rotate*, used as an example in Section 5.4, to rotate the needle of the dial. By using only reversible output primitives, the program written for this robot automatically becomes reversible. This means that while debugging the application, the color the needle is pointing towards will always correspond to variable *current* in the program. Concretely, if the debugger steps back through the program from the end of a loop iteration to line ref{line:target}, it will move the needle back to the previous color without having to read a new sensor value. This makes it easy to test certain state transitions where the needle is pointing at one particular color and now has to move to a different color.

[7]Code for all examples can be found *[link to repository removed for double-blind review]*

[8]Full link: https://youtube.com/playlist?list=PLaz61XuoBNYVcQqHMAAXQNf8fz5IAMahe&si=HNrKY9YzqDFadATN



```
enum Color { none = 0, red = 1, green = 2,
             blue = 3, yellow = 4 }

const sensor = colorSensor(Pin.IO2);
let current: Color = Color.none;
while (true) {
  let next: Color = sensor.read();
  if (next != current) {'<line:target>'
    // turn the needle if the color changed
    rotate(Pin.IO1,
      (next - current) * angle, speed);
  }
  current = next;
}
```

Figure 5-31. Left: Lego color dial that recognizes the color of objects. Right: The main loop controlling the behavior of the color dial. The dial is controlled by a single motor connected to pin IO1, and the color sensor is connected to pin IO2.

Aside from using time travel debugging which keeps external state in mind, users of our debugger are also able to leverage multiverse debugging capabilities to deal with the non-deterministic nature of this color sensor. This allows them to easily simulate various sensor values, and explore the different paths the robot can take without needing to use any real, correctly-colored objects. This example touches on a few common aspects of robotics applications, such as processing non-deterministic input, controlling motors and making decisions based on sensor values. Using the I/O primitives supported by MIO, various other applications could be build; such as a binary LED counter, a smart curtain, an analogue clock, a maze solving robot, and so on.

5.6 Related work

Our work builds directly on WebAssembly (Haas et al., 2017; Rossberg, 2023, 2019) and WARDuino (Lauwaerts et al., 2024), as we have discussed in Section 5.3.3 and Section 5.4. In this section, we present an overview of further related work.

5.6.1 Multiverse debuggers

Multiverse debugging has emerged as a powerful technique to debug non-deterministic program behavior, by allowing programmers to explore multiple execution paths simultaneously. It was proposed by Torres Lopez et al. (2019) to debug parallel actor-based programs, with a prototype called

Voyager (Gurdeep Singh et al., 2019), that worked directly on the operational semantics of the language defined in PLT Redex (Felleisen et al., 2009). Several works have expanded on multiverse debugging; Pasquier et al. (2022) introduced user-defined reduction rules to shrink the state space that must be explored during multiverse-wide breakpoint lookup, and Pasquier et al. (2023b) introduced temporal breakpoints that allow users to reason about the future execution of a program using linear temporal logic. In contrast to MIO, existing multiverse debuggers only work on a model of the program execution, and do not consider I/O operations, or their effects on the external environment.

paragraph{Multiverse Analysis} The idea of exploring the multiverse of possibilities, is more widely known as multiverse analysis. Within statistical analysis, it is a method that considers all possible combinations of datasets and analysis simultaneously (Steegen et al., 2016). Within software development, there are several frameworks for exploratory programming (Kery and Myers, 2017), which allow developers to interact with the multiverse of source code versions (Steinert et al., 2012). In exploratory programming, programmers actively explore the behavior of a program by experimenting with different code. This approach has led to *programming notebooks* (Kery et al., 2018, Perez and Granger, 2007), and dedicated *explore-first IDEs* with advanced version control (Kery and Myers, 2017, Steinert et al., 2012). Explore-first editors, such as the original by Steinert et al. (2012), allows programmers to explore different versions of their code in parallel. While explore-first editors consider the variations in the program code itself, multiverse debuggers focus on variations of program execution caused by non-deterministic behavior for a single code base. Combining these two techniques could lead to a powerful development environment, and represents interesting future work.

5.6.2 Exploring execution trees

Many automatic verification and other analysis tools also explore the execution tree of a program, such as software *model checkers* (Godefroid, 1997, Jhala and Majumdar, 2009), *symbolic execution* (Baldoni et al., 2018, Cadar et al., 2011, King, 1976), and *concolic execution* (Godefroid et al., 2005, Marques et al., 2022, Sen and Agha, 2006). These techniques are great at automatically detecting program faults, however, they rely on a precise description of the problem or program specification, often in the form of a formal model. This is in stark contrast with debuggers, which are tools to help developers find mistakes for which no precise formula exists, and for which the causes are often unknown. Despite the major differences, static analysis techniques could greatly help improve debuggers by providing the developers with more information. For multiverse debugging the techniques could help guide developers through large and complicated execution trees. Additionally, the techniques for handling the state explosion problem (Kahlon et al., 2009, Kurshan et al., 1998, Valmari, 1998) developed for these analysis tools, can help reduce the number of redundant execution paths in multiverse debugging.

5.6.3 Reversible debuggers

Reversible debugging, also called back-in-time debugging, has existed for more than fifty years (Balzer, 1969), and has been implemented with various strategies (Engblom, 2012). *Record-replay debuggers* (Agrawal et al., 1991, Boothe, 2000, Burg et al., 2013, Feldman and Brown, 1988, O'Callahan et al., 2017, Ronsse and De Bosschere, 1999) allow offline debugging with a checkpoint-based trace. In spite of all the different implementation strategies, few reversible debuggers also reverse output effects, with a few notable exceptions. The more recent RR framework (O'Callahan et al., 2017) is a culmination of many years of research, and is one of the most advanced record-replay debugger to date. While replaying it does not reverse I/O operations, in fact, the operations are not performed at all. For example, file descriptors are not opened during replay, instead the external effects are recorded and replayed within the debugger. One of the earliest works, the Igor debugger (Feldman and Brown, 1988), featured so-called *prestart routines*, which could perform certain actions after stepping back, such as updating the screen with the current frame buffer. This is one of the first attempts at dealing with external state, however, the solution was purely ad-hoc, and required significant user intervention; for instance, supplying the name, mode, and file pointer for each file currently opened during execution. Additionally, dealing with I/O in a structured way through the prestart routines was still too costly at the time. There is also no proof of soundness, or any characterization of which prestart routines lead to correct debugging behavior. *Omniscient debuggers* (Lewis, 2003, Pothier et al., 2007), on the other hand record the entire execution of a program, allowing free offline exploration of the entire history, and enabling advanced queries on causal relationships in the execution (Pothier et al., 2007). A third approach is based on *reversible programming languages* (Giachino et al., 2014, Lanese, 2018, Lanese et al., 2018). While not applicable in all scenarios, since it requires a fully reversible language, this approach can enable more advanced features, such as reversing only parts of a concurrent process, while still remaining consistent with the forwards execution (Lanese, 2018). The reversible LISP debugger by Lieberman (1997) not only redraws the graphical output, but also links graphics with their responsible source code. Reversible debuggers for the *graphical programming language* Scratch (Maloney et al., 2010), namely Blink (Strijbol et al., 2024) and NuzzleBug (Deiner and Fraser, 2024), also redraw the graphical output when stepping back. However, in all these debuggers, the output effects are internal to the system. For the Scratch debuggers, the visual output is actually part of the execution model (Maloney et al., 2010).

5.6.4 Reversible programming languages

The concept of reversible computation has a longstanding history in computer science (Bennett, 1988, Mezzina et al., 2020, Zelkowitz, 1973), with the most notable models for reversibility being reversible Turing machines (Axelsen and Glück, 2016), and reversible circuits (Saeedi and Markov, 2013). Furthermore, the design of reversible languages has evolved into its own field of study (Glück and Yokoyama, 2023), with languages for most programming paradigms, such as the imperative, and first reversible language, Janus (Lami et al., 2024, Lutz and Derby, 1986, Yokoyama et al., 2008), several functional languages (Matsuda and Wang, 2020, Yokoyama et al., 2012), object-oriented languages (Haulund et al., 2017, Hay-Schmidt et al., 2021, Schultz and Axelsen, 2016), monadic computation (Heunen and Karvonen, 2015), and languages for concurrent systems (Danos and Krivine, 2004, Hoey et al., 2018, Schordan et al., 2016). Several works have investigated how reversible languages can help reversible debuggers (Chen et al., 2001, Engblom, 2012, Lanese et al., 2018), however, full computational reversibility is not necessary for back-in-time debugging (Engblom, 2012). Moreover, these reversible languages do not consider output effects on the external world, with a few notable exceptions in the space of proprietary languages for industrial robots.

5.6.5 Reverse execution of industrial robotics

While numerous examples can be imagined where actions affecting the environment cannot be easily reversed, there are sufficient scenario's where this is possible, for reverse execution to be widely used in industry. The reversible language by Schultz (2020) is particularly interesting. The work proposes a system for error handling in robotics applications through reverse execution, and identifies two types of reversibility; direct and indirect. Through our compensating actions, MIO is able to handle both directly and indirectly reversible actions. Laursen et al. (2018) propose a reversible domain-specific language for robotic assembly programs, SCP-RASQ. While we do not focus on a single specific application domain, this work does show how reversible output primitives are possible for advanced robotics applications. SCP-RASQ uses a similar system of user-defined compensating actions, to reverse indirectly reversible operations. Using these kinds of languages, we believe that the MIO debugger could be extended to support more complex output primitives, which could control industrial robots.

5.6.6 Reversibility

The concept of reversibility is well understood on a theoretical level, for both sequential context (Leeman, 1986), and concurrent systems. The latter is much more complex, and has lead to two major definitions; causal-consistent reversibility (Danos and Krivine, 2004, Lanese et al., 2014), and time reversibility (Kelly, 1981, Weiss, 1975). Causal-consistent reversibility is the idea that an action can only be reversed after all subsequent dependent actions have been reversed (Lanese et al., 2014). This ensures that all consequences of an action have been undone before reversing, and the system always returns to a past consistent state. On the other hand, time reversibility only considers the stochastic behavior when time is reversed (Bernardo et al., 2023, Kelly, 1981, Weiss, 1975). However, it has recently been shown that causal-consistency implies time reversibility (Bernardo et al., 2023). Our debugger works on a single-threaded language, where the non-determinism is introduced by the input operations. In our work, the undo actions are causally consistent in the single-threaded world. We believe that we can extend MIO to support concurrent languages, and that the existing literature (Giachino et al., 2014, Lanese et al., 2018) can help to ensure it stays causally consistent.

5.6.7 Remote debugging on microcontrollers

In remote debugging (Rosenberg, 1996), a debugger frontend is connected to a remote debugger backend running the program being debugged. The MIO debugger uses remote debugging to mitigate some limitations of microcontrollers, an approach that has been adopted for many embedded systems (Pötsch et al., 2017, Skvařc Bǒzǐc et al., 2024, Söderby and De Feo, 2024). These debuggers fall in two categories; *stub* and *on-chip* (Li et al., 2009). A stub is a small piece of software that runs on the microcontroller, instrumenting the software being debugged, this is the approach taken by the MIO debugger. On-chip debugging uses additional hardware to debug the embedded device, a common example are JTAG ("IEEE Standard for Test Access Port and Boundary-Scan Architecture," 2013) hardware debuggers. These debuggers can interface with different software, such as the popular OpenOCD debugger (Högl and Rath, 2006). However, remote debugging can exacerbate the probe effect (Gait, 1986), and can be very slow since the debugger runs on the microcontroller, combined with constant communication overhead. To address these limitations, a new form of remote debugging, called out-of-place debugging, has been proposed (Lauwaerts et al., 2022, Marra et al., 2018). This technique moves part of the debugging process to a more powerful machine, which can reduce debugging interference and speedup performance. The MIO debugger is already sufficiently fast, but a speed-up can likely be achieved by adopting out-of-place debugging.

5.6.8 Environment modeling

There are many environment interactions that can influence the possible input values and thereby the possible execution paths of a program. We have elided these interactions from the formal model and assume that I/O operations are independent, while our prototype does support defining simple *predictable dependencies* between I/O operations. Modeling the interactions between I/O operations is also hugely important for testing, and *environment modeling* has therefore been widely studied in this area (Blackburn, 1998). Environment models are often used for automatic test generation (Auguston et al., 2005, Dalal et al., 1999) for a certain specification, and have also been applied to real-time embedded software (Iqbal et al., 2015).

5.6.9 Formalizing debuggers

Previous efforts to define debuggers formally have been incredibly varied in their depth and approach, and have not yet reached a consensus on any standard method. An early attempt used PowerEpsilon (Zhu and Wang, 1992, 1991) to define a denotational semantic describing the source mapping needed to debug a toy language that can compile to a toy instruction set (Zhu, 2001b). In 2012, the work by Li and Li (2012) focussed on automatic debuggers, and defined operational semantics for tracing, and for backwards searching based on those traces. In 1995, Bernstein and Stark (1995) defined a debugger in terms of an underlying language semantic for the first time, an approach we adopted in this work as well. In fact, the approach is followed by a number of later works (Ferrari and Tuosto, 2001, Holter et al., 2024, Lauwaerts et al., 2024, Torres Lopez et al., 2017), including the work by Torres Lopez et al. (Torres Lopez et al., 2019) in 2019, which defines the correctness of their debugger in terms of the non-interference with the underlying semantic. The correctness criteria requires each execution observed by either semantic is observed by the other, which is similar to our soundness and completeness theorems rolled into one. A more recent work presented a novel abstract debugger, that uses static analysis to allow developers to explore abstract program states rather than concrete ones (Holter et al., 2024). The work defines operational semantics for their abstract debugger, and for a concrete debugger. The soundness of the abstract debugger is defined in terms this concrete debugger, where every debugging session in the concrete world is guaranteed to correspond to a session in the abstract world. The opposite direction cannot hold since the static analysis relies on an over-approximation, which means there can always be sessions in the abstract world which are impossible in the concrete world. This is in stark contrast with the soundness theorem in our work, which states that any path in the debugging semantics can be observed in the underlying language semantics.

5.7 Conclusion

While existing multiverse debuggers have shown promise in abstract settings, they struggled to adapt to concrete programming languages and I/O operations. In this article, we address these limitations by presenting a novel approach that seamlessly integrates multiverse debugging with a full-fledged WebAssembly virtual machine. This is the first implementation that enables multiverse debugging for microcontrollers. Our approach improves current multiverse debuggers by being able to provide multiverse debugging in the face of a set of well-defined I/O primitives. We have formalized our approach and give a soundness proof. We have implemented our approach and have given various examples showcasing how our approach can deal with a wide range of specialized I/O primitives, ranging from non-deterministic input sensors, to I/O pins and even steering motors. Our sparse snapshotting approach delivers reasonable performance even on a restricted microcontroller platform. Our initial implementation provides a substantial benefit over existing approaches, but we believe there are further opportunities to relax the constraints on I/O primitives further. For example, our current implementation only supports simple dependencies between I/O actions, but we believe this could be relaxed further by introducing an explicit rule language so that programmers can define more complex dependencies between the I/O actions.

Chapter 6

Managed Testing

If you know the way broadly, you will see it in everything.
— Miyamoto Musashi, *The Book of Five Rings*

6.1 Introduction

Software testing for constrained devices, still lags behind standard best practices in testing. Widespread techniques such as automated regression testing and continuous integration are much less commonly adopted in projects that involve constrained hardware. This is mainly due to the heavy reliance on physical testing by Internet of Things (IoT) developers. A 2021 survey on IoT development found that 95% of the developers rely on manual (physical) testing (Makhshari and Mesbah, 2021). Testing on the physical hardware poses three major challenges, which hinder automation and the adoption of modern testing techniques. First, the *memory constraints* imposed by the small memory capacity of these devices makes it difficult to run large test suites. Second, the *processing constraints* of the hardware causes tests to execute slowly, preventing developers from receiving timely feedback. Third, *timeouts and flaky tests* pose a final challenge. When executing tests on constrained hardware it is not possible to know when a test has failed or is simply taking too long.

To circumvent the limitations of constrained hardware, simulators are sometimes used for testing IoT systems (Bures et al., 2020). Their usage makes adopting automated testing and other common testing practices much easier. Unfortunately, simulators can never fully capture all aspects of real hardware (Espressif Systems, 2023a, Khan et al., 2011, Roska, 1990). Therefore, to fully test their applications, IoT developers have no other option than to test on the real devices. This is the primary reason why developers still prefer physical testing. Another reason is the lack of expressiveness when specifying tests in automated testing frameworks. Testing frameworks with simulators almost exclusively focus on unit testing, and hence provide no good alternative to end-to-end physical testing performed by developers manually (VanderVoord et al., 2015).

In this chapter, we argue that programmers should not be limited by either the constraints of the hardware, or a simulator imposed by the testing framework. Therefore, our goal is to design and implement a testing framework for automatically running large-scale versatile tests on constrained systems. This has lead to the development of the *Latch* testing framework (Large-scale Automated Testing on Constrained Hardware). *Latch* enables programmers to script and run tests on a workstation, which are executed on the constrained device. This is made possible by a novel testing approach, we call *managed testing*. In this unique testing approach, the test suite is split into small sequential steps, which are executed by a testee device under the directions of a controlling tester device. The workstation functions as the tester which maintains full control over the test suite. Only the program under test—not the entire test suite—will be sent to the constrained device, the testee. The tester will use instrumentation to manage the testee and instruct it to perform the tests step-by-step. This means the constrained testee is not required to have any knowledge of the test suite being executed. This is quite different from traditional remote testing, where the entire test suite is sent to the remote device. The instrumentation of the testee is powered by debugging-like operations, which allow for traditional whitebox unit testing, but also enables the developer to write debugging-like scripts to construct more elaborate testing scenarios that closely mimic manual testing on hardware.

The research question we seek to answer in this chapter, is whether the managed testing approach, i.e. splitting tests into sequential steps, is sufficient for executing large-scale tests on microcontrollers. To answer this question, we will show how managed testing allows *Latch* to overcome all three major challenges of testing on constrained devices. The approach can be summarized as follows. In *Latch* test suites are split up into smaller test instructions that are sent incrementally to the managed testee, thereby freeing the test suites from the *memory constraints* of the hardware. This is crucial in enabling large-scale test suites on microcontrollers, such as the large unit testing suite containing 10,213 tests we use to evaluate our approach. To overcome the *processing constraints*, *Latch* can skip tests that depend on previously failing tests resulting in a faster feedback loop. Finally, *Latch* handles *timeouts* automatically, and includes an analysis mode which reports on the *flakiness of tests*.

6.1.1 Contributions

- We define a test specification language for writing large tests suite for constrained devices.
- We develop the *Latch* framework, that implements the test specification language as an embedded domain-specific language (EDSL).

- We present a novel testing methodology based on debugging methods, that allows common manual testing of code on hardware to be automated.
- We illustrate how *Latch* can be used to address testing scenarios from all three layers of the testing pyramid (Cohn, 2009).
- We evaluate *Latch* by using it to run 10,213 unit tests on an ESP32 micro-controller.

The rest of the chapter starts with a discussion of the challenges of testing on constrained device in Section 6.2. In Section 6.3 we give a first introduction to the *Latch* test specification language through a basic example, and use the example to give an overview of the *Latch* framework. We discuss the details of the language in Section 6.4, and focus on how tests are written and executed by the framework. For each aspect of the test specification language we discuss how it helps *Latch* to address the challenges outlined previously. We conclude the section by briefly touching on the prototype implementation. Section 6.5 further illustrates how *Latch* can be used to handle different testing scenarios, and can help testers implement a range of testing methodologies. We discuss three scenarios, classic large-scale unit testing, integration testing, and automating physical end-to-end testing using the debug-like operations provided by *Latch*. In Section 6.6 we evaluate the runtime performance of *Latch* based on a variety of test suites, and present empirical evidence that managed testing enables large-scale automatic testing on constrained hardware. In Section 6.7 we discuss the related works, before concluding in Section 6.8.

6.2 Challenges of Testing on Constrained Devices

This section outlines the challenges preventing large-scale testing on constrained hardware.

6.2.1 Memory Constraints

In this article we focus on the ESP32 microcontroller family having about 400 KiB SRAM and 384 KiB ROM, typically operating at a clock frequency around 160-240 MHz. Due to these hardware limitations, programs cannot be arbitrarily large as the program memory is quite small and they execute slower than workstations. For companies producing IoT devices it is often desirable to make use of the cheapest and most minimal hardware possible that can handle the task at hand. This means that when executing on the hardware, there are often very few resources to spare, which limits the ability to test the applications on the device.

ESP32 devices can have different amounts of memory, but the order of magnitude is the same.

When test suites become large, executing these test suites on the hardware is often not possible because the compiled binary is too big to fit in the program memory of the microcontroller. The only option then is to split the test suite into smaller parts which can fit on the device. Current testing frameworks, however, do not provide automated support for splitting large test suites and executing them incrementally on the hardware. Programmers who want to execute large test suites thus have to manually partition the test suite, execute the test on the hardware, read out the results and process the dump of the individual parts.

Finally, even when the testing framework supports partitioning of the test suite reflashing the hardware for every partition is quite time-consuming. To change the program executing on the hardware the programmer needs to flash the microcontroller, i.e. write the program in the ROM partition of the microcontroller. Depending on the microcontroller, synchronization and flashing of a new program can take several seconds making it undesirable to flash the microcontroller often.

6.2.2 Processing Constraints

When relying on regression testing, the programmer wants a tight feedback loop. Ideally, the entire test suite is run after each change, but this requires feedback to be reported quickly. However, by testing on constrained devices, executing the test suite can take a lot of time, slowing down the software development cycle significantly. To provide feedback as early as possible, the framework should catch failures early. This can take many forms, but in essence a failure of any kind during a test should be visible to the developer as soon as possible. Additionally, to avoid spending time on tests that cannot succeed, the framework should run as few of these tests as possible.

Finally, when multiple hardware testbeds are available it should be easy for the developer to run tests in parallel to speed up testing. The same facilities for scheduling and parallelization options available for unconstrained devices, should be integrated into testing frameworks for constrained devices.

6.2.3 Timeouts and Flaky Tests

Due to the limited memory and processing power of constrained devices, large test suites need to be split up in smaller chunks. Moreover, the results of the test need to be communicated with a test machine and combined. Unfortunately, this approach implies that test engineers suddenly need to take into account many of the problems associated with distributed computing.

First, when the test machine is waiting for a response, it cannot reliably distinguish between a failure or a delayed response. Many other testing frameworks need to deal with this problem, especially JavaScript frameworks (Flanagan, 2020) where asynchronous code is prevalent (Fard and Mesbah, 2017). These frameworks time out tests that take too long, unfortunately, the fact that a test timed out does not provide much information for developers, especially when a test includes multiple asynchronous steps.

Second, the non-determinism of the asynchronous communication also contributes to an inherent problem of testing, flaky tests (Lam et al., 2019). These are tests that can pass or fail for the same version of the code. Unfortunately, on constrained hardware, many tests have the potential to become flaky due to the inherent non-determinism of these systems. For example, when testing communication with a remote server small changes in the communication timing with the server could lead to different behavior.

6.3 Managed Testing with *Latch* by Example

To overcome the outlined challenges, *Latch* uses a unique testing approach that consists of declarative test specification language to describe tests, and a novel test framework architecture to run tests. We refer to our new approach as *managed testing*. In managed testing, the testing framework runs on a local machine and delegates tests step-by-step to one or more external platforms, which are running the software under test. To facilitate this approach, tests must be easily divisible into sequential steps. That is why *managed testing* specifies tests in a declarative test specification language, where tests are described as scenarios of incremental steps. In this section we give a first overview of how managed testing in *Latch* works through an example, before going into further detail in Section 6.4. The example is chosen as a small primer on how programmers can write traditional unit tests with *Latch*'s test specification language.

6.3.1 The Example

We define a unit test that verifies the correctness of a function for 32-bit floating point multiplication, shown in Listing 6-16. All example programs are written in AssemblyScript (The AssemblyScript Project, 2023), one of the languages supported by *Latch*'s current microcontroller platform.

Listing 6-17 shows a simple test in *Latch* containing one unit test for the target program in Listing 6-16. *Latch*'s declarative test specification language is implemented as an embedded domain specific language (EDSL) in TypeScript (Microsoft, 2023). Test scenarios are presented in *Latch* as TypeScript

```
export function mul(x: f32, y: f32): f32 {
    return x * y;
}
```

Listing 6-16. A `mul` function that multiplies its two arguments, written in AssemblyScript.

```
const multiplicationTest: Test = {
    title: "example test",
    program: "multiplication.ts",
    steps: [{
        title: "mul(6,7) = 42",
        instruction: invoke("mul", [WASM.f32(6), WASM.f32(7)]),
        expect: returns(WASM.f32(42))
    }]
};
```

Listing 6-17. A *Latch* scenario defining a unit test for the `mul` function.

objects that have a title, the path to the program under test, and a list of steps. These steps make up the test scenario, and will be performed sequentially. Each step performs a single instruction, and can perform several checks over the result of that instruction.

The example performs only a single instruction, it requests that the *mul* function is invoked with the arguments 6 and 7 (see). These arguments are first passed to the *WASM.f32* function, to indicate the expected type in *AssemblyScript*. On , the example specifies that the function returns the number 42. Usually, the instruction and expectations for a step are described as objects, but *Latch* provides a handful of functions to construct these objects for common patterns—such as *invoke* and *returns*. This makes test scenarios less verbose, and quicker to write. We go into further detail on the structure of the *instruction* and *expectation* objects in Section 6.4.

Similar to other testing frameworks, *Latch* allows test scenarios to be grouped into test suites. Crucially, the test suites in *Latch* have their own set of testee devices, on which they will be executed. When writing a new test suite in *Latch*, programmers need to add at least one testee to the suite. Such testees can range over a wide variety of microcontrollers, as well as local simulator processes. Each platform may differ in how software is flashed, or communication initialized and performed. These platform specific concerns are captured by a single TypeScript class, `Testee`. Each connection with a constrained device is represented by an object of such a class. In Listing 6-18 for instance, we use the Arduino platform to connect to an ESP32 over a USB port, as shown on . Users can add their own platforms by defining new sub-

```
const suite = latch.suite("Example test suite");
suite.testee("wrover A", new ArduinoSpec("/dev/ttyUSB0", "esp32:esp32:esp32wrover"),
5000)'\label{line:suite:testee}'
     .testee("wrover B", new ArduinoSpec("/dev/ttyUSB1", "esp32:esp32:esp32wrover"))
     .test(multiplicationTest);'\label{line:suite:test}'
latch.run([suite]);'\label{line:suite:run}'
```

Listing 6-18. *Latch* setup code to run the `multiplicationTest` on two ESP32 devices.

classes of the `Testee` class, which can handle the specific communication requirements of the new platform.

Aside from testees, a test suite also requires test scenarios to execute. The example multiplication test is added to the test suite on , before the suite is given to *Latch* to be run on .

Listing 6-18 shows how a test suite is built in *Latch* through a fluent interface (Xie, 2017), meaning the methods for constructing a test suite can be chained together. Each test suite in latch is entirely separate from the rest, and therefore contains only its own tests, and platforms to run those tests on. In the example, two ESP32 devices are configured for the test suite. This means that when the test suite is started with the **run** function on , the framework will execute all scenarios in the suite on all configured platforms. Alternatively, the user can configure *Latch* to not execute duplicate runs, but instead to split the tests into chunks that are performed in parallel on different devices. In that case, each test is only run once and the execution time of the whole test suite should be dramatically improved due to the parallelization.

6.3.2 Running the Example on the *Latch* Architecture

To run the above testing scenario on a remote constrained device, the test is loaded into *Latch* on the local unconstrained device, the *tester*. During testing, the *tester* manages one ore more *testees* (constrained devices) to execute tests step-by-step. Figure 6-32 gives an overview of all steps and components involved during testing in the *Latch* framework. The left-hand side shows the tester, which runs the *Latch interpreter* and *test execution platform*. The interpreter component is responsible for interpreting the test suites, which are written in the *test specification language*, while the test execution platform sends each instruction in a test step-by-step to the testee device over the available communication medium. The test execution platform also parses the result, and handles all other aspects of communication with the testee device.

We will go over the steps shown in Figure 6-32 in the order they are executed by *Latch*. Running a test suite is initiated by the interpreter, which takes the

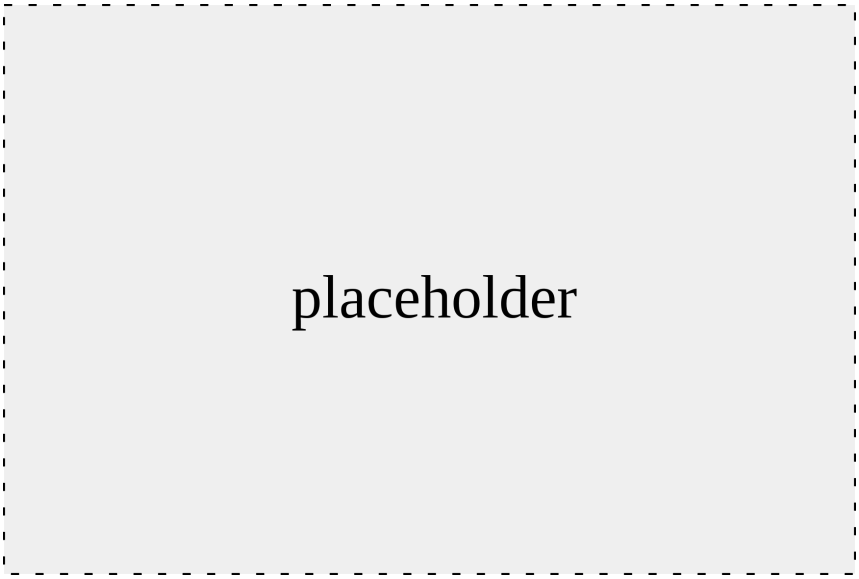


Figure 6-32. Schematic overview of the interaction between components in *Latch* during a test.

test suite specification ①, and schedules the *scenarios* ②. Since the example test suite in Listing 6-18 only contains a single test scenario, the multiplication test, with a single step—the scheduling is not relevant in this case. In real test suites, the order in which tests are run is important, it can help detect failing tests early, or minimize expensive setup steps. When the interpreter selects a test to be executed, it will instruct the test execution platform ③ to first upload the *software under test*, and subsequently sends the instructions of the scenario to the *test instrumentation platform* ④. In the case of our example, *Latch* compiles the *multiplication.ts* file and uploads it to the ESP32 device that is connected to the USB port. Once this step is completed, *Latch* sends the invoke instruction to the testee, which will execute the *mul* function with the supplied arguments.

Aside from forwarding instructions to the test instrumentation platform, the tester can also perform custom actions to control the *environment* ⑤. For instance, these actions can control hardware peripherals, such as sensors and buttons, that interact with the constrained testee ⑥ during the test.

Listing 6-19 shows how a step might send an MQTT message to a server as an example of an action that acts on the environment. Such a step, could be useful when testing an IoT application that relies on MQTT messages.

```
1 const sendMQTT: Step = {
2     title: "Send MQTT message",
3     instruction: simpleAction((): void => {
4         let client: mqtt.MqttClient = mqtt.connect("mqtt://test.mosquitto.org");
5         client.publish("parrot", "This is an ex-parrot!");
6     })
7 };
```

Listing 6-19. An example *Latch* step, which performs a custom action that sends an MQTT message to a server.

The microcontroller can connect to an actual testing server, and via custom actions *Latch* can test if the device responds correctly.

In contrast with Listing 6-17, this example constructs the instruction object explicitly, rather than calling a function such as *invoke*. There are two types of instructions, they can be either a *request* to the test instrumentation platform, such as the invoking of a function, or a custom *action*. In this example we construct a simple action that takes no arguments and returns nothing. Actions allow tests to execute TypeScript functions as steps in the test scenario, in this case the function simply publishes a test message to the MQTT server (Line 5). We go into further detail on the types of actions and requests in Section 6.4.

As tests are performed, the software under test is controlled by the test instrumentation platform in accordance with the *request* instructions send by the test execution platform ⑦. In other words, the test instrumentation platform will receive the command from the tester to execute the *mul* function, and make the software under test invoke it. The instrumentation of the software under tests, allows the test instrumentation platform to return any generated output to the test execution platform ⑧. Whenever the tester sends an instruction to the testee, *Latch* will wait until the testee returns a result for the instruction. When working with constrained devices, communication channels may be slow or fragment messages. *Latch* takes care of these aspects automatically.

As part of a step, the scenario description can specify a number of assertions over the returned results. In the example, we require that the *mul* function returns 42, as specified on of Listing 6-17. Once the expected output is received by the tester, *Latch* checks all assertions against it. These assertions are verified by the interpreter ⑨, before the result of the step is shown in the *user interface* as either passed, failed, or timed out ⑩. For example, after the test instrumentation platform returns the result of the *mul* function, *Latch* will check if it indeed equals 42 and report the result.

A step can have three kinds of results; either it timed out, or all its assertions passed, or one of more assertions failed. In other words, step is marked as failing when at least one assertion fails. If no assertions were included in the step, *Latch* will not wait for output, and immediately report the action as passing. When the testee fails to return a result after a preconfigured period, it is marked as timed out. Similarly, a scenario is marked as failing when at least one step fails. When a step fails, the test execution platform will—by default —continue the scenario without retrying the step. This is useful when the steps in the scenario are independent of each other to gather more complete feedback. Otherwise, developers can configure *Latch* to abort a scenario after the first failure.

The results of each step are reported while the test suite is executing. When the entire suite has run, *Latch* will give an overview of all the results for both the steps and the test scenarios. This overview includes, the number of passing/failing tests, the number of passing/failing steps, the number of steps that timed out, and the overall time it took to run the suite. In addition, the developer can configure *Latch* to report on the flakiness of the test by executing the tests multiple times. This way, *Latch* can compare the results of different runs to give developers more insight into the flakiness of their test suites. As Figure 6-32 shows under the user interface component, the results in this case will be reported for each run separately. Whenever the runs give different results, the scenario is marked as flaky and the failure rate is reported.

6.3.3 From Small Examples Towards Large-scale Test

The running example in this section illustrates *Latch*'s basic testing features. In particular, how *Latch* divides tests into small steps that are executed sequentially. This means that the size of the test suite is no longer constrained by the memory size of the embedded device. While the example here only includes a single step, one can easily imagine test cases that require many more steps. Let us suppose we stay within the realm of unit testing a mathematical framework. We can imagine a more complicated mathematical operation than multiplication that requires thorough testing, for instance a function *eig* for calculating the eigenvalues of a matrix. In this case the test scenario would include many steps, that each invokes the *eig* function with a different matrix. This is similar to the large-scale unit testing suite we will discuss in Section 6.5, and those run as part of the evaluation in Section 6.6.

Section 6.5 discusses realistic examples for each layer of the testing pyramid; unit testing, integration testing, and end-to-end testing. The examples will illustrate how using small steps powered by debugging-like operations, uniquely enables *Latch* to test remote debuggers and automate IoT scenarios and manual hardware tests. For example, it becomes much easier to test whether a microcontroller successfully receives asynchronous messages from a remote server, and handles these message correctly. The test can set breakpoints in the code that is expected to be executed when a message arrives. Before sending the message, the test can pause the execution at the exact place in the program, it wants the message to be received. The *Latch* instructions allows users to write these kinds of testing scenarios in a convenient way. Moreover, the increased control over the program, makes the test scenarios much easier to repeat reliably under the same conditions.

6.4 The *Latch* Test Specification Language

Latch tests are written in a declarative test specification language embedded in TypeScript. This EDSL allows developers to specify what tests should be performed, while hiding the complexity of communicating with the constrained testing device. Equally important are the debug-like commands provided by the language, which make it easier to automate hardware testing scenarios. Latch tests can be viewed as scripted scenarios of sequential operations. The programmer can specify what the result of executing an operation should look like, instead of manually testing whether the returned value is consistent with the expected result. For more complex tests the programmer can write test-specific evaluation functions to check whether the program behaves as expected.

The test specification language consists of four major abstractions: a test, a testing step, test instructions, and assertions. Each test includes a name, some start-up configuration and the testing steps which need to be executed during the actual test. Each testing step specifies an instruction that needs to be executed. There are two types of instructions, commands and actions. The commands are debug-like operations that are send directly to the test instrumentation platform of the testee, such as invoking a method, pausing the program, etc. Alternatively, there is support for user-specified instructions called actions. These actions allow programmers to implement their own logical and physical interactions with the hardware or the environment.

The interface of a test, shown in Listing 6-20, consists of a title, the path to the program to load on the testee device, a set of initial breakpoints to halt execution, a list of dependent test, and a set of steps to be executed during the test. Both the initial breakpoints and dependent tests are optional, as indicated by the question mark after their identifier.

Testing steps all adhere to the *Step* interface shown in Listing 6-21. Each step should minimally have a title and specify which instruction to perform when

```
interface Test {
  title: string;
  program: string;
  steps: Step[];
  dependencies?: Test[];
  initialBreakpoints?: Breakpoint[];
}
```

Listing 6-20. Interface for *Latch* tests. Each test has a title, indicates a program to be tested, and lists the steps to executed.

```
interface Step {
    readonly title: string;
    readonly instruction: Command<any> | Action<any>;
    readonly expect?: Assertion[];
}
```

Listing 6-21. A step has a name, a specific command or action it should perform, and a possibly list of assertions to check.

executed. A step only contains a single instruction, and all steps are executed synchronously. As part of a step, the result of executing an instruction can be verified by means of assertions.

An instruction in *Latch* is either a command, or an action. Both instruction types are annotated with their return type, this is the type of the object passed to each assertion of the step. The list of assertions is optional, a step without any assertions will always succeed and immediately go to the next step.

6.4.1 Default Commands in *Latch*

The set of commands *Latch* supports is shown in Table 6-15. We divide the set of commands in intercession, meta, and introspection commands. The intercession commands, allow *Latch* tests to intervene directly with the software under test. With **invoke** the programmer can call a function and wait for the result, as illustrated by the step in our multiplication example (Listing 6-17). This enables unit testing of specific functions, as is the popular approach adopted in most testing frameworks (The JUnit Team, n.d., Python Software Foundation, n.d.). With **set local** the programmer can change a local variable, this is especially useful to test a program with local boundary conditions without having to rerun the program completely.

The **reset** and **upload module** instructions are primarily for internal use in *Latch*, but are available in the test specification language. The upload module instruction loads a binary onto the testee, replacing any current program. The reset instruction restarts the current program.

Category	Commands
Intercession	invoke, set local, *upload module*
Meta	pause, set breakpoint, continue, delete breakpoint, step, step over, *reset*
Introspection	core dump, dump callback mapping, dump locals

Table 6-15. The *Latch* commands. Internal commands are in italic.

The meta instructions allow the programmer to install a debugging scenario by setting breakpoints and running the program to a particular point in the execution. These are especially useful for automating manual hardware tests, where different steps and events often need to happen in very specific orders. By controlling the execution of the program, these kinds of scenarios can be replicated accurately each time.

Finally, the introspection commands allow the programmer to inspect the current state of the program. Without these commands, *Latch* test would be limited to testing black boxes, since the software under test is executed on a different device. Thanks to the introspection commands, *Latch* supports black box as well as white box tests.

The proposed set of commands are inspired by standard debugging instructions, and focus on enable standard unit testing, as well as automation of manual hardware tests. Since the test specification language is embedded in TypeScript, the set of commands is easily extended by the user. Other debugging instructions could similarly inspire new *Latch* commands, such as run until, setting of conditional breakpoints, exception breakpoints, or inspecting memory addresses. Instructions tailored to asynchronous tests, such as awaiting an event, or waiting for a given time, would likewise be good additions. A new command has to implement the interface shown in Listing 6-22. A command is identified by the test instrumentation platform by its type, examples include pause, set breakpoint, and step. These commands can optionally take a payload, such as a breakpoint address for example, and each command has its own parser to interpret the response of the test instrumentation platform.

By taking inspiration from debugging instructions, managed testing permits for a wide range of automated tests to be implemented, which would otherwise require additional engineering efforts in existing unit testing frameworks. Additionally, we have found that it provides a very natural way of writing tests for constrained devices. We illustrate both these points by discussing in-depth examples for each layer of the testing pyramid in Section 6.5.

```
export interface Command<R> {
    type: Interrupt,                    // type of the debug message (pause, run,
    step, ...)
    payload?: (map: SourceMap.Mapping) => string,  // optional payload of the debug
    message
    parser: (input: string) => R                              // the parser for the
    response
}
```

Listing 6-22. Commands are distinguished by `type` and may have callback to access payload. Results are extracted by a parser.

```
type Assertable<T extends Object | void> = {[index: string]: any};

interface Action<T extends Object | void> {
    act: (testee: Testee) => Promise<Assertable<T>>;
}

declare function assertable<T extends Object>(obj: T): Assertable<T>;
```

Listing 6-23. *Latch* actions allow developers to execute arbitrary code in a test step. Output of such actions can be checked for correctness with the `Assertable<T>` interface.

6.4.2 Custom Actions in *Latch*

Aside from these commands, *Latch* allows steps to perform custom actions. These custom actions enables developers to execute arbitrary code as part of a step in the testing scenario. This is useful for interacting with the environment when testing the firmware of hardware components. Listing 6-23 shows the interface for a custom action. An action is an object with a single act field, containing a function that takes a Testee argument and returns a promise. The testee argument is provided at runtime by the *Latch* framework, to provide customs actions with access to the test instrumentation platform. This is useful to define actions that need to respond to changes on the testee device, for instance waiting for a breakpoint to be hit. Actions may be asynchronous and therefore return promises. A promise is the standard mechanism for managing asynchronicity in JavaScript and TypeScript (Madsen et al., 2017, Parker, 2015). If the action is expected to return a response, the promise should contain the output. For *Latch* to run checks over this output, it needs to be of the **Assertable** type. *Latch* provides a function that can turn any object into an **Assertable** object.

In Section 6.3 we briefly showed a simple action in Listing 6-19. However, this action returned no result, over which the test step could define assertions. Listing 6-24 gives an second example of an action that does return a result. The action will listen for the next MQTT message for a specific topic. On Line

```
1 function listen(topic: string): Action<Message> {
2     let client: mqtt.MqttClient = mqtt.connect("mqtt://test.mosquitto.org");
3
4     return {
5         act: () => new Promise<Assertable<Message>>((resolve) =>
6             client.on("message", (_topic: string, payload: Buffer) => {
7                 if (topic === _topic)
8                     resolve(assertable({topic: topic, payload:
                      payload.toString()}));
9 })) }; }
```

Listing 6-24. An example of a pure action that listens for the next MQTT message to a specific topic.

5, the `act` function returns a promise that resolves when the first message for the correct topic arrives. The promise contains the MQTT message of the application-specific **Message** type, including a topic and payload field. This object can be used to define checks over the payload of the message with *Latch* assertions. However, for *Latch* to run checks against the message, the returned object must conform to the *Assertable* interface. That is why on Line 8, the message object is wrapped in a *Assertable* by the assertable function, shown in Listing 6-23.

6.4.3 Assertions over instruction results

Aside from the instruction, each step contains a list of zero or more assertions. These assertions are used to perform checks on the result of the step's instruction. The result of an instruction is always of the *Assertable* type shown in Listing 6-25, which is an object that contains any number of properties that are indexed by strings.

For each string-indexed property of an *Assertable* result, a test step can contain one or more assertions. The interface of the assertions is shown in Listing 6-25. The Assertions represent a check over a single property of the assertable object, specified by their string index. The assertions over the object's properties follow the `Expect` interface, also shown in Listing 6-25. An `Expect` object represents an assertion over an object property of the result, and takes a type parameter `T` that should correspond with the type of that property. The `Expect` interface can be used to check for a value of type `T`, or a behavior encoded by the `Behavior` enum also shown in Listing 6-25. Behaviors can check for an unchanging, changing, increasing, or decreasing value. If these options do not suffice, developers can write their own custom checks. These are written as comparison functions that take the actual resulting value from the test, and return a boolean indicating whether the check passes.

```
interface Assertion { [index: string]: Expect<any>; }

type Expect<T> = T | Behaviour | (value: T) => boolean;

enum Behavior { unchanged, changed, increased, decreased }
```

Listing 6-25. Instructions return their results as Assertable objects. In Latch tests specify assertions over the arbitrary properties of these Assertable result.

```
const dump: Command<State>;

interface State {
    line: number;    // current line position
    column: number;  // current column position
    mode: Mode;      // execution mode
    func: string;    // current function
}
```

Listing 6-26. The *core dump* command returns a state object, which contains the source location, execution mode, and name of the currently executing function.

```
1  const step: Step = {
2      title: "CHECK: entered *echo* function",
3      instruction: Command.dump,
4      expect: [{mode: Mode.PAUSE}, {func: "echo"}]
5  }
```

Listing 6-27. Example step that uses the **core dump** command to check that execution paused in the `echo` function.

The interface for assertions is implemented in TypeScript using a discrimination union, which is a design pattern used to differentiate between union members based on a property that the members hold. For brevity, we have omitted this detail in Listing 6-25 and all examples that follow.

The introspection commands are particularly interesting for assertions, since they enable assertions over the internal state of the testee. Consider the core dump command which returns a state object, shown in Listing 6-26 shows the dump command, which returns a state object.

For example, the dump command allows a step to check whether the testee is paused in a particular function. Listing 6-27 shows how you might write this test step. Line 4 adds two assertions to the step. The first checks whether the mode field in the state is set to pause, and the second checks if the current function has the correct name.

```
class TestSuite {
    public testee(name: string, testee: Testee): TestSuite;
    public scheduler(scheduler: Scheduler): TestSuite;
    public test(test: Test): TestSuite;
}
```

Listing 6-28. The `TestSuite` allows developers to specify the testees, i.e., target devices, configure the scheduler, and the set of tests to be executed.

With the test specification language, developers can declaratively describe tests independently of the platform they should be executed on. By embedding the domain-specific language in TypeScript, we can use the type system of TypeScript to type all the constructs in the EDSL and catch mistakes in tests early.

6.4.4 Managed Testing

Given a test written in the *Latch* test specification language, the framework will execute it through a single tester which manages one or more constrained testees. That is, the software under test runs on a constrained device and the test suite is kept on the unconstrained tester device. The tester will instruct the constrained device to perform tests by sending instructions step-by-step. This design allows test to be run on constrained devices, while overcoming the memory constraints.

In the example of Section 6.3 we configured a test suite in *Latch* to run on two devices. The test specification language has two main components to specify this configuration. First, the language has an overarching concept of a test suite that groups a number of tests. Each test suite runs independently of the others, and maintains its own devices, and their communication. Listing 6-28 shows the public methods of the `TestSuite` class in *Latch* that can add new devices and tests to a test suite. Finally, when a test suite is created and fully configured, it can be executed on all devices with the `run` method.

The devices passed to a test suite, represent a single connection to a device. *Latch* supports different devices each with their own abstraction, which needs to be able to connect and disconnect, upload a program, and send instructions. The interface of these abstractions is captured by the abstract class in Listing 6-29.

6.4.5 Using Test Scheduling and Expressing Dependent Tests

Performing tests on remote hardware testbeds is often slow, which delays feedback. To make testing on constrained devices part of continuous integration in practice, we reduce the time it takes to get feedback on failing tests by

```
abstract class Testee {
    abstract connect(): Promise<void>;
    abstract upload(program: string): Promise<void>;
    abstract sendCommand<R>(command: Command<R>): Promise<R>;
    abstract disconnect(): Promise<void>;
}
```

Listing 6-29. The `Testee` implements support for different devices to enable upload of programs, and command execution.

```
function celsius(fahrenheit: f32): f32 {
    return (fahrenheit - 32) * 0.556;
}
```

Listing 6-30. AssemblyScript function to convert Fahrenheit to Celsius.

not running unnecessary ones. *Latch* allows dependencies between tests to be defined explicitly, as part of the test syntax as shown in Listing 6-20. Each test in *Latch* can specify a list of tests it depends on. The framework treats these dependencies between tests as transitive. This enables the framework to skip tests that cannot succeed, thereby mitigating the effects of the processing constraints.

6.4.5.1 Example

To illustrate test dependencies, we expand on our earlier multiplication test example. Suppose our constrained device is connected to a temperature sensor that uses Fahrenheit, but our software uses Celsius. For the conversion, we use the AssemblyScript function in Listing 6-30.

The conversion to Celsius depends on the multiplication of 32-bit floating point numbers, which we tested in our previous example. If the test for multiplication fails, we know that the `celsius` function will fail, too, and we can avoid running the temperature conversion test to safe time. Consequently, we list the `multiplicationTest` as a dependency on Line 4 in Listing 6-31. Dependencies are entirely defined by the user, the only restriction is the disallowing of cyclical dependencies. Currently, the framework throws a runtime error whenever it encounters a cyclical dependency between a group of tests.

For complex scenarios, we can list an arbitrary number of *dependencies*. If any of the dependencies should fail, *Latch* skips the test. For continuous integration these tests are considered failing, but they are marked with a distinct *skipped* label and counted separately from true failures by *Latch*.

```
const dependentTest: Test = {
  title: "Example Test with a dependency.",
  program: "celsius.ts",
  dependencies: [multiplicationTest],
  steps: [{
    title: "Fahrenheit to Celsius test",
    instruction: invoke("celsius", [WASM.f32(46.4)]),
    assert: returns(WASM.f32(-8.0))
  }]
};
```

Listing 6-31. *Latch* test suite for the `celsius` function, with a dependent scenario.

```
class Scheduler {
    public schedule(tests: Test[]): Test[];
}
```

Listing 6-32. `Schedulers` enable custom ordering of tests. The ordering can avoid unnecessary test execution, or allow for test prioritization.

6.4.5.2 User-defined Schedulers

The order in which tests are executed can also influence the execution time of the test suite, especially since failing dependent tests can prevent unnecessary computations. To further speed up the execution, the test specification language allows developers to configure the scheduling algorithm the framework uses when running a test suite. The best scheduling algorithm depends on the exact test suites. Therefore, scheduling is configured at the level of a test suite as shown earlier in Listing 6-28. Scheduling algorithms are implemented as subclasses of the `Scheduler` class from the test specification language shown in Listing 6-32. The class only has one public method that takes a list of tests, and returns a new list with the tests sorted according to the scheduler's prioritization. This class allows developers to embed their own schedules in the test specification language.

The current implementation of the *Latch* framework, provides two predefined schedulers, the default and the optimistic scheduler. We give the pseudocode for both scheduling algorithms in Algorithm 6-4 and Algorithm 6-5 respectively. Since dependencies amongst tests are transitive and cyclical dependencies are disallowed, we can extract trees from a test suite, where linked nodes depend on each other. Both algorithms will use this fact.

The default scheduler prioritizes the dependencies between tests and works best with test suites where a large number of tests dependent on a much smaller set of scenarios. The algorithm of the default scheduler, first finds all the dependency trees. The *findDependencyGraphs* function constructs a

Algorithm 6-4. The default scheduling algorithm in *Latch*.

Algorithm 6-5. The optimistic scheduling algorithm to minimizing program uploads.

Figure 6-33. The two scheduling algorithms provided by *Latch*.

forest of directed dependence trees. In these graphs the nodes are tests that directly depend on their parents. The function will throw a runtime error if any cyclical dependencies are encountered. After the trees are found, the algorithm will append their tests breadth-first to the schedule. Within the same depth the tests are sorted alphabetically based on the program's name, to minimize the number of times the tester needs to upload code. The resulting list of tests, is ordered in such a way that trees are executed one after the other, and no test is ever run before any test it depends on.

The optimistic scheduler is built on the assumption that dependent tests are more likely to use the same program. If this is the case for the test suite, it can result in far fewer code uploads during a run compared to the default scheduler. The algorithm starts by initializing an accumulator as a list of lists. Then, it constructs the dependence trees in the same way as the default scheduler. Next, for each tree the tests are aggregated into lists of tests with the same depth in the tree, the siblings in other words. Subsequently, the algorithm appends each group of siblings to the list in the accumulator that corresponds to its level. After all trees have been traversed, the accumulator is flattened to a one-dimensional list, and returned as the schedule.

The default scheduler iterates breadth-first over each dependence tree in succession. In contrast, the optimistic scheduler can be seen as traversing the entire dependence forest breadth-first. Again, at each depth the tests are sorted alphabetically according to their program. These two schedulers are provided as examples of scheduling algorithms, each test suite most likely has its own optimal algorithm.

Thanks to the scheduling based on the test dependencies, *Latch* can detect failures early and prevent unnecessary tests from running. However, the time needed for executing tests can be further minimized by executing test suites in parallel. Since microcontrollers are typically cheap and abundantly available, it makes sense to run different tests on separate devices at the same time. Currently, the schedulers still return a single ordering over the tests, but the dependency trees constructed as part of their algorithms offer an opportunity to parallelize. Different dependency trees can be safely run in parallel, since tests in different trees have no dependencies in common.

6.4.6 Handling and Reporting on Timeouts

Since all actions of a test are executed remotely, the tester cannot distinguish between an unresponsive test and a test that can still succeed after a long time. This is an unavoidable problem when testing in the presence of asynchronous actions. Many modern testing frameworks deal with this by adding timeouts to all asynchronous tests. In *Latch* we use timeouts for testing on constrained devices, too, but provide as much information as possible about where timeouts occur. Thus, *Latch* provides timeouts at the level of single instructions, following the example of frameworks dedicated to testing of asynchronous system (Haleby, n.d.), rather than merely at the level of a test, as is common practice in more general test frameworks (The JUnit Team, n.d., OpenJS Foundation, n.d., Python Software Foundation, n.d.). We found that debugging timeouts, is significantly easier with fine-grained information.

The actions of the tests are not the only source of asynchronicity in *Latch*. There are other asynchronous actions behind the scenes, from compiling test programs to connecting with hardware testbeds. In *Latch* every asynchronous action can time out, and each timeout has their own helpful message indented to make them easily identifiable by developers.

6.4.7 Detecting and Reporting Flaky Tests

The asynchronicity and non-determinism introduced by *Latch* and the hardware testbeds, can cause any test to become flaky. These tests can both succeed and fail for the same version of the software under test. In *Latch*, we follow the recommendation of (Harman and O'Hearn, 2018) to considers all tests as flaky. Indeed, flaky tests can hint at bugs. Therefore, we use an approach that improves the debuggability of flaky tests.

The framework can run in two modes. A normal mode which executes each action and each test at most once, and an analysis mode where tests are executed multiple times to analyze flakiness. In this mode we assume all tests are flaky. Therefore, tests are rerun even if they succeed. This can slow the test suite significantly, which is why it is provided as an optional mode. Indeed, the default mode still allows continuous integration to report initial results quickly, while the flakiness of the test suite can be reported at a later moment after the second mode has finished. The analysis mode trades performance for more information and certainty.

The analysis mode can be configured with a minimum and maximum number of runs. Since we consider all tests as flaky, we will execute each test at least the minimum number of times. If the test reports the same result for each of these runs, *Latch* assumes it is not flaky and stops for this scenario. In the other case, we already have proof that the test is flaky, and

Latch will continue executing up to the maximum number of times to get a more representative measure of the flakiness. The maximum number of runs is important to have statistically significant results, and can therefore be configured by the user for each run. The minimum and maximum number of runs can be configured by the user. When a test suite is executed on multiple platforms, flakiness is measured for each platform separately. At the end of the analysis run, *Latch* reports the global flakiness of the test suite for each platform as the number of flaky scenarios, and at the end of the analysis run, *Latch* reports the flakiness on each platform for each test and gives an overview with the overall flakiness of the test suite for each platform separately and all platforms together.

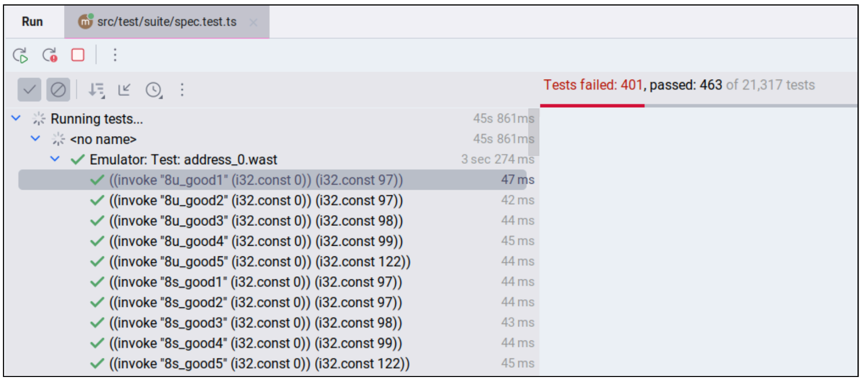
6.4.8 Prototype Implementation

The prototype implementation of *Latch* is a TypeScript library built on the WARDuino (Lauwaerts et al., 2024) virtual machine for constrained devices and the Mocha testing framework for JavaScript and TypeScript.

WARDuino is a WebAssembly (Haas et al., 2017) virtual machine targeting ESP32 microcontrollers. The virtual machine also has basic debugging support, which we used as the basis for implementation our test instrumentation platform in *Latch*. By using a WebAssembly virtual machine, *Latch* can test programs written in any language that can compile to WebAssembly. This includes most of the mainstream programming languages used today, such as C, C++, Java, Python, Ruby, Rust (Fermyon Technologies, Inc., 2023). In order to enable testing of a language fully, *Latch* needs to have support for compilation and sourcemapping. The current implementation has support for compiling and constructing sourcemaps for AssemblyScript.

We believe the general principles we use for implementing the *Latch* prototype on WARDuino, can be applied to any language or virtual machine which provides basic debugging support. This includes the C programming language that is supported by many microcontrollers, and which offers basic debugging by means of a JTAG interface.

The *Latch* prototype uses the Mocha testing framework for JavaScript and TypeScript to report the results of the tests in the *Latch* framework. Handling the output through an existing framework, immediately gives *Latch* integration into most of the existing IDEs used for programming in TypeScript and JavaScript.



```
Tests failed: 401, passed: 463 of 21,317 tests
Running tests...                                                  45s 861ms
  <no name>                                                       45s 861ms
    Emulator: Test: address_0.wast                                3 sec 274 ms
      ((invoke "8u_good1" (i32.const 0)) (i32.const 97))          47 ms
      ((invoke "8u_good2" (i32.const 0)) (i32.const 97))          42 ms
      ((invoke "8u_good3" (i32.const 0)) (i32.const 98))          44 ms
      ((invoke "8u_good4" (i32.const 0)) (i32.const 99))          45 ms
      ((invoke "8u_good5" (i32.const 0)) (i32.const 122))         44 ms
      ((invoke "8s_good1" (i32.const 0)) (i32.const 97))          44 ms
      ((invoke "8s_good2" (i32.const 0)) (i32.const 97))          44 ms
      ((invoke "8s_good3" (i32.const 0)) (i32.const 98))          43 ms
      ((invoke "8s_good4" (i32.const 0)) (i32.const 99))          44 ms
      ((invoke "8s_good5" (i32.const 0)) (i32.const 122))         45 ms
```

Figure 6-34. IDE integration in WebStorm (JetBrains s.r.o., 2023).

6.5 Testing with *Latch*

Latch offers a framework for writing unconstrained automated test scripts, that can address many different testing scenarios. To demonstrate the versatility of *Latch*, we will present a common testing scenario for microcontrollers for each stage of the testing pyramid (Cohn, 2009). Several versions of the pyramid exist, often tailored to specific software domains (Mukhin et al., 2021). Generally, testing pyramids split testing into three or more stages, which are often performed in order from bottom to top. Each successive layer in the pyramid tests larger parts of the software in one test. Therefore, each layer will typically have fewer tests than those before it. The testing pyramid is a common way of representing the full scope of testing for a software project, it is therefore suitable to showcase *Latch*'s ability to support the full range of testing scenarios.

In this section, we adhere to the classic testing pyramid, with unit testing at the bottom, followed by integration (or service) testing, and finally topped by end-to-end (user) testing. We first highlight how *Latch* can perform realistic, large-scale unit testing on constrained hardware. Then we show how *Latch* can test the instrumentation it uses as an illustration of integration testing. Finally, we show how manual testing on hardware can be automated to perform end-to-end testing. The example test suite illustrates how this can be used to test both the hardware itself, and the software libraries used for controlling that hardware.

6.5.1 Unit Testing: Large-scale Testing of a Virtual Machine

In the testing pyramid, the largest number of tests are the unit tests. The underlying virtual machine used by *Latch*, WARDuino, uses a subset of the official WebAssembly specification test suite, to test whether it conforms

```
(module (func (export "mul") (param $x f32) (param $y f32) (result f32) (f32.mul
(local.get $x) (local.get $y))))
(assert_return (invoke "mul" (f32.const -0x0p+0) (f32.const 0x0p+0)) (f32.const
-0x0p+0))
(assert_return (invoke "mul" (f32.const -0x1p-149) (f32.const -0x0p+0)) (f32.const
0x0p+0))
```

Listing 6-33. An *assert-return* test from the official WebAssembly Specification test suite, testing the `f32.mul` operation.

with the WebAssembly standard. WARDuino does not use the entire official test suite, since it does not yet support all the latest accepted proposals to the standard. The WARDuino project uses an extended version of the virtual machine to parse and run the unit tests from the test suite. Unfortunately, this means it cannot be executed on microcontrollers, since the entire suite needs to be included as well as the large parsing library needed to extract the unit tests. By using *Latch*, we are able to take the same test suite, and execute it on an ESP32 microcontroller. We discuss the results further in Section 6.6, in this section we focus on how the official specification test suite is written in *Latch*.

Test files in the WARDuino test suite contain a number of WebAssembly modules, each of which has a number of assertions. These assertions are so called *assert-return* tests, which invoke a WebAssembly function and specify the expected result. The assertions are written as S-expressions.footnote{This conforms with the official WebAssembly specification tests, which can be found on: url{https://github.com/WebAssembly/spec/tree/main/test/core}} Cref{lst.specsource} shows two such assertions.

With *Latch*, we can run the same tests on actual embedded hardware. The structure of the WebAssembly specification test suite is well suited for *Latch*'s test specification language. The asserts coincide perfectly with the steps in the test. Each assert contains a single action to perform and a single assertion to check. Therefore, all specification tests for WebAssembly can be encoded as a single test suite with a test for each distinct module. Listing 6-34 shows the example in Listing 6-33 translated into a *Latch* test.

To test the WARDuino virtual machine, we converted the official WebAssembly test specification into a large *Latch* test suite. Since *Latch* is a DSL embedded in TypeScript, this conversion can easily be done programmatically in TypeScript code. Converting the *assert-return* S-expressions to *Latch* syntax in this way is fairly, easy. The conversion enables us to test the WARDuino virtual machine incrementally. The test instrumentation framework will only load one WebAssembly module from the test suite at a time and each test is converted into steps, which are sent to the testee incrementally, i.e. the testing steps do not need to be stored in the memory of the testee.

```
const test: Test = { // Spec test
  title: "Test f32.mul operation",
  program: "module.wast",
  steps: [
    { title: "assert: -0 * +0 = -0",
      instruction: Command.invoke("mul", args: [-0, 0]),
      expect: returns(WASM.f32(-0)) },
    { title: "assert: -1e-149 * -0 = 0",
      instruction: Command.invoke("mul", [-1e-149, -0]),
      expect: returns(WASM.f32(0)) }
  ]
};
```

Listing 6-34. The `f32.mul` test has two steps, each checking the result of `mul` on different inputs.

```
1 export function main(): void {
2     blink();
3     print("started blinking");
4 }
```

Listing 6-35. The blink program used by the integration test for the WARDuino debugger API.

In Section 6.6, we give an overview of the performance of executing this test suite on an ESP32 device.

6.5.2 Integration Testing: Testing a Debugger API

Due to its design, *Latch* is well suited to test the debugging operations of the WARDuino virtual machine. Testing the debugger API exemplifies the second layer of the testing pyramid: integration testing.

As an example, consider the step over debug instruction, which steps over a single function call or a single instruction when the instruction does not call a function. A simple test starts at a function call and sends the debugging instruction, before checking if the program did step over it correctly.

The blink program in Listing 6-35 calls on Line 2 the `blink`() function and on Line 3 the `print`() function. With this program, we check that the program executes up to Line 3, rather than stopping at the start of the main function. Listing 6-36 shows the corresponding definition of a test in *Latch*. It loads the program, calls the main function, and sends a step over instruction. At the end of the test, it checks whether the current line has indeed moved to Line 3.

The debugging tests illustrate how integration tests can frequently dependent on each other. For instance, our small *step over* test uses the *invoke* and *dump*

```
const stepOverTest: Test = {
    title: "Test STEP OVER",
    program: "blink.wast",
    dependencies: [dumpTest, invokeTest]
    steps: [
      { title: "Start program",
        instruction: Command.invoke("main", []) },
      { title: "Send STEP OVER command",
        instruction: Command.stepOver },
      { title: "CHECK: execution stepped over direct call",
        instruction: Command.dump,
        expect: [{line: 3}] }
    ]
};
```

Listing 6-36. The description for *Latch* of the **step over** test.

instructions, which can also be tested with *Latch*. When tests for either these two instructions fail, we can no longer rely on the results of the *step over* test. Since the *invoke* or *dump* commands may be broken, they might cause false positives, or false negatives, in tests that use them. There is no reason to run tests that cannot be trusted. In *Latch*, we can encode the dependency of the *step over* test on the *invoke* and *dump* command, by adding their tests to the list of dependent tests. With this information, *Latch* can prevent unnecessary or unreliable tests from slowing down the test suite, and delaying actionable feedback.

6.5.3 End-to-End Testing: Automating Manual Testing on Hardware

Developers of embedded software rely heavily on manual testing of their programs on the targeted hardware. The goal of manual testing is to verify that both the hardware and software of the system work correctly. It is equally important to check that the effects on the environment and the interaction between the hardware and the environment, work as intended. This kind of comprehensive end-to-end testing of embedded systems requires extensive control over the environment and conditions the hardware operates under, such as simulating user interactions, or controlling the input for sensors. % These requirements account in large part for the ubiquity of manual testing, since they make automation of testing much more difficult.

Latch allows tests to control the behavior of the environment with local actions, and the behavior of the software under test through debugging instructions. This enables developers to script automated tests that correspond with manual testing scenarios.

```
function echo(topic: string, payload: string): void {'\label{line:mqtt:callback}'
    print(payload);
}

export function main(): void {
    // ...
    mqtt_init("broker.hivemq.com", 1883);'\label{line:mqtt:init}'
    mqtt_subscribe("echo", echo);'\label{line:mqtt:subscribe}'
    // ...
}
```

Listing 6-37. Tiny MQTT program used to regression test the callback handling system in WARDuino.

When performing end-to-end testing on the hardware, whether manual or automated, things outside the control of the system can go wrong and cause the test to fail even though no part of the software under test is at fault. Such failures are often rare and non-deterministic, leading to flaky tests. The built-in detection and reporting of flaky tests in *Latch* is therefore important for end-to-end testing scenarios with the hardware.

6.5.3.1 Example: Testing MQTT Primitives

The WARDuino virtual machine has a callback handling system that is used to implement different asynchronous IoT protocols (Lauwaerts et al., 2022), such as primitives for the MQTT protocol. Since the correct implementation of such protocols is crucial for applications, we need to test it extensively. Unfortunately, the public WARDuino project currently has no automated tests for these components, especially since they require interaction with the device to be tested. The following example illustrates how *Latch* can be used to write end-to-end tests for both the callback system and the MQTT primitives. The example wants to verify the following two requirements:

begin{enumerate}[ref=*requirement arabic* (Section 6.5.3.1)] item label{req:1} After the subscribe primitive is called, the callback function should be registered for the correct topic in the virtual machine's callback system. item label{req:2} When an MQTT message is received the correct callback function should be called. end{enumerate}

To test this functionality, we use a minimal program that subscribes on a single MQTT topic, and through a callback writes all messages it receives to the serial bus. An AssemblyScript implementation is shown in Listing 6-37.

The code in Listing 6-37, leaves out the code that connects to the Wi-Fi network, and checks the connection with the server whenever the program is idle. The example instead focuses on the three main things the program

```
const test: Test = { // MQTT test
  title: "Test MQTT primitives",
  program: "mqtt.ts",
  dependencies: [testWiFi],
  steps: [
    { title: "Start program",
      instruction: Command.invoke("main", []) },
    { title: "CHECK: callback function registered",
      instruction: Command.dumpCallbackMapping,
      expect: [{
        callbacks: (state, mapping) => mapping.some((map) => map["echo"].length > 0)}] },
    { title: "Set breakpoint at *echo* callback",
      instruction: Command.setBreakpoint(breakpointAtFunction("echo")) },
    { title: "Send MQTT message and await breakpoint hit",
      instruction: Actions.messageAndWait() },
    { title: "CHECK: entered callback function",
      instruction: Command.dump,
      expect: [{mode: Mode.PAUSE}, {func: "echo"}] }
  ]
};
```

Listing 6-38. Test for the callback handling system in WARDuino, showing multiple steps and a custom assertion.

needs to do for the end-to-end test. It configures the MQTT server on Line 7, and subscribes to the *echo* topic on Line 8 with the callback function defined on Line 1. The scenario in Listing 6-38 uses the program to test the callback system and MQTT primitives of the WARDuino virtual machine on real hardware.

Many hardware-specific tests require the environment to behave in a controlled way. *Latch* makes no assumptions about the hardware and environment used for testing. Instead, the test specification language offers the ability to define local actions, through which the tester in the framework can manipulate and control the environment, both real and simulated.

The first step of the scenario invokes the main function, and the second step checks whether the echo callback was correctly registered in WARDuino's internal callback mapping. In the third step, the scenario sets a breakpoint at the callback function, so in the next step it can check if the callback is indeed called whenever an MQTT message is sent. To this end, the fourth step tells the tester to perform a local action. In the example, the *messageAndWait* function will send a message to the MQTT broker and wait until the testee reports that a breakpoint is hit. Once its promise resolves, we know a breakpoint is hit, and the final step double-checks whether we are indeed in the right function. When the promise is rejected, however, the action is marked as failing before continuing the scenario. This fifth step retrieves a dump of the current virtual machine state, and checks that WARDuino is paused and the current function corresponds to the *echo* callback function.

6.6 Performance Evaluation

The goal of *Latch* is to allow large-scale testing of IoT software on microcontrollers, and to enable users to write a versatile range of tests. The testing scenarios in the previous section illustrate the versatility of *Latch* to implement many testing strategies. Sections Section 6.3 and Section 6.4 show how managed testing works, and what the *Latch* framework does to overcome the three challenges outlined in Section 6.2. In this section we provide empirical evidence to support our research question:

begin{description} item[Question] Is the managed testing approach, where tests are split into steps, sufficient for executing large-scale tests with *Latch*? end{description}

6.6.1 Test suites

To answer the question of performance, we execute a number of tests suites with *Latch* on an ESP32-WROVER IE and measure the runtime overhead compared to executing the same suites on a laptop. The test suites include the unit and debugging test suites presented in Section 6.5, and an additional test suite which is more computationally intensive. % We chose these three types of test suites in order to have a wide range of tests that are unique in different aspects. The specification test suites from Section 6.5.1 are structurally identical, but test very different aspects of computer programs, ranging from memory manipulation, to control flow. The suites also represent a very common test pattern, unit testing through single function invocation, which is ubiquitous in many modern testing practices. The debugging test suite on the other hand, does not limit itself to just the invoke command, but uses the entire range of *Latch* commands in its tests, which also contain multiple steps. The computing test suite is structurally similar to the specification test suites, but is computationally more intensive, with steps that generally take at least an order of magnitude longer to perform.

paragraph{Large Unit Test Suites} We use the WARDuino specification test suites as found in the public repository of the virtual machine, which we presented in Section 6.5.1. The collection contains 10,213 total tests across 25 test suites. The tests cover the operations on the numerical values, both integer and floating point, which are the only types of values in WebAssembly. The *copy*, *load*, *align*, and *address* categories test the WebAssembly memory, while the *local tee*, *local set*, *local get*, *nop*, *return*, *call indirect*, and *call* categories test stack manipulation. The remaining tests verify the structured control flow of WebAssembly. During the evaluation we used the default scheduling algorithm in *Latch*, and ran the test suites on a single remote testee.

The developers of the WARDuino virtual machine use simulation to test against the WebAssembly specification. However, the simulation ignores important hardware limitations. For instance, the memory of the simulated hardware is only limited by the amount of memory available to the host machine. Furthermore, to execute the specification tests, the WARDuino developers extended the simulator with a dedicated parsing library to parse the test suite written in S-expressions. This parsing library is too big to be run on the ESP32 and the S-expressions from the test suite alone, take up 713 KB of memory. This is already more than twice the size of the microcontroller's memory, without including the WARDuino virtual machine, the parsing library, and the infrastructure to run the test suite. This means, that the WARDuino developers cannot currently test on the microcontrollers they target. However, when comparing the outcome of this approach with the output of the *Latch* version, we found no differences, giving us confidence in the soundness of our framework. To assess the performance of *Latch*, we measure the overhead of executing the *Latch* test suites on a microcontroller compared with current practices, i.e. using a simulator. %

paragraph{WARDuino Debugger Test Suite} While the different specification test suites, test very different aspects, their structure are similar. We therefore include the debugger test suite outlined in Section 6.5, as an example that is not a traditional unit test suite. Rather than exclusively using the invoke command, this suite uses all commands available to *Latch*.

paragraph{Computing Test Suite} As a final example, we include a test suite that unit tests a few simple mathematical operations, calculate the factorial, get the nth number in the fibonacci sequence, find the greatest common divider, and check if a number is prime. Similar to the specification test suites, the computing tests each include a single invoke step. However, while the steps in the other test suites are very fast, taking just a few milliseconds—the steps in this test suite can take several centiseconds.

6.6.2 Performance

All test suites are run separately on a Dell XPS 13 laptop using an 11th Gen Intel Core i7-1185G7 and 32 GB RAM memory, and the ESP32-WROVER IE microcontroller operating at a clock frequency of 240 MHz, and with 520 KiB SRAM, 4 MB SPI flash and 8 MB PSRAM. Each run starts by initializing the WARDuino instance, in the case of the microcontroller this entails flashing the entire virtual machine to the device. Whenever the test suites use different programs, they are uploaded with the *upload module* command, which allows *Latch* to update the program under test during a test suite, without needing to flash.

A detailed comparison of the overhead of executing the test suite is shown in Figure 6-35. The overhead on microcontroller is shown as relative to the simulator, and is the sample mean taken over 10 runs. *Latch* is run in its default mode without the flakiness analysis, where tests are run at most once. Each bar in this graph shows the overhead for executing one test suite on the hardware with *Latch*. The test suites are ordered from most steps, to least. The number of steps are shown next to each name, and the specification tests suites are highlighted in a different color.

All test suites shown in Figure 6-35 can be executed with *Latch* on the simulated version of WARDuino in approximately 10 minutes. Executing the same test suites directly on the ESP32, takes around 20 minutes. While the test suites take on average twice as long on the embedded device, the largest of the specification test suites run faster on the microcontroller. This is counterintuitive, but can be explained by the nature of the test suites. The steps in the specification test suites, are very simple tasks that are performed too quickly for any difference to be observed between the two devices, The overhead therefore becomes dominated by the communication, not the actual instructions themselves. The way the TypeScript framework handles the interprocess communication is evidently slower than the serial communication with the microcontroller over USB-C. However, the flashing at the start remains much slower than starting a new process on the laptop, therefore the overhead of the specification test suites with the fewest steps, is dominated by the startup phase instead. This results in the highest overhead overall.

The specification test suites taken separately in Figure 6-35, shows that fewer test steps results in higher overhead, because the execution time becomes dominated by the flashing process. This shows how important it is to prevent unnecessary flashing by using the *upload module* command. Conversely, more steps result in lower overhead, because the communication dominates the execution time of the steps. However, the debugger and computing test suites are major outliers, suggesting this is not the full story. For instance, the computing test suite contains 3,470 steps, but has a much higher overhead than the memory copy suite of a similar size. This is due to the steps in the computing test suite being much more computationally intensive, and so much slower. Because the steps take longer to execute, the relative impact of the communication overhead is much lower. The debugger test suite on the other hand, has a much lower overhead than similarly sized specification

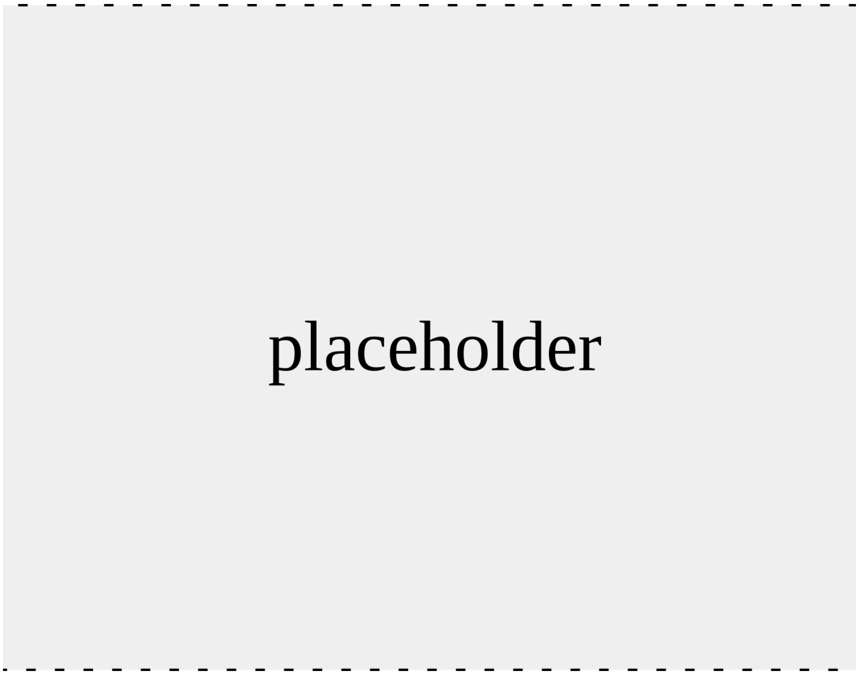


Figure 6-35. The relative runtime overhead of *Latch*'s WebAssembly specification test suite on hardware compared to a simulator for each test suite. Runtimes are calculated as sample means of 10 runs, and the exact relative overhead is shown next to each bar. The error bars show the confidence interval for the difference between the two means (normalized to the relative overhead) based on the Welch's t-test. The number of steps for each test suite is listed next to its name.

test suites. This is because, invoke instructions used in the specification test suites, are quite slow compared to most of the other *Latch* commands, used in the debugger test suite. An entire user-defined function is run, in contrast to the step and dump command, which run a single instruction, or only send data. While the differences in structure among the test suites, reveals how many factors impact the performance of *Latch*, the results for the suites are roughly inline with each other. The results show that *Latch* performs well for our use-case of very large test suites of many small unit tests, which are very common in regression testing and continuous integration.

6.6.3 Summary

It is important for the validity of these results, that the test suites used here, are representative for the typical workloads of microcontroller software test suites. We consider this to be the case, since the specification, computing and debugger test suites are very different structurally, yet present very similar runtime performance. Additionally, we believe that the specification test suites are representative for microcontroller software testing. The suites use standard unit tests that invoke a function and check its results. It is no coincidence that these kinds of unit tests are so widely spread in test-driven development. They are an excellent way of testing that can be applied to almost any piece of software, regardless of its structure or programming paradigm. This also holds for microcontroller software. Moreover, the *invoke* instruction is one of the more expensive operations in *Latch*, since user defined functions may take very long to execute. Finally, the specification test suites are quite heterogeneous, since the categories test wildly different aspects of the virtual machine—from among others, control flow, arithmetic, stack manipulation, and memory access. For these reasons, we believe that this test suite is able to give a representative evaluation of *Latch*'s performance.

The performance results themselves, are impacted by an innumerable number of factors. Especially since the benchmarks are run on two devices, the framework on the laptop and the tests on the microcontroller. The communication between the two is an important factor in the runtime performance, and may be influenced by any number of factors, such as the operating system of either device, the configuration of the microcontroller, or the hardware (serial connection) itself. However, we believe given the size and number of repetitions, the performance figures are illustrative for the overall performance of managed testing with *Latch*.

In conclusion, we believe that our initial evaluation shows that the *Latch* framework and its managed testing approach present a realistic answer to our research question. The framework is able to automatically execute large-scale test suites on constrained devices with good performance, considering the limited processing power of the constrained devices. *Latch* performs the best for our most important use-case, test suites with high numbers of small unit tests for the same software under test. On the other hand, the performance overhead is highest when *Latch* needs to upload new software frequently. The *Latch* prototype has initial support for parallel execution on multiple constrained devices which can help mitigate this overhead, especially when many test programs can be uploaded simultaneously to different devices.

6.7 Related Work

Common software development practices such as regression testing, continuous integration, and test driven development, are much harder to adopt when working with microcontrollers. This is in large part due to the need to test on the physical hardware, specifically microcontrollers. There are very few solutions for single-target testing of software on microcontrollers. Ztest

(Peress et al., 2024), Unity (PlatformIO, 2023a, VanderVoord et al., 2015), and ArduinoTest (Murdoch, 2023) are traditional unit testing frameworks for specific microcontroller architectures. Unfortunately, these frameworks do little to overcome the resource-constraints of microcontrollers themselves, and provide only the most standard unit testing functionality without any tailored solutions for testing on hardware. However, when testing on microcontrollers in this way, the test scenarios often rely on very specific hardware interactions as illustrated by our examples in Section 6.5. *Latch* addresses this lacuna with its novel testing methodology based on debugging methods. We are not aware of any testing framework that provides an alternative solution.

In this section, we will discuss the differences between *Latch* and the few exiting unit testing frameworks for microcontrollers further. In this chapter we have proposed a new way of testing on microcontrollers individually, but IoT systems are often tested as a whole in industry. While this kind of testing answers an entirely different set of demands than *Latch*, we do give a brief overview of these approaches here, for completeness. Similarly, testing plays a large role in general software development. As a result, a wide range of research topics are related to the *Latch* framework, of which not all have been previously applied to IoT. In the remainder of this section, we discuss *holistic IoT testing*, other *unit testing frameworks* broadly, *remote testing*, *scriptable debugging*, *test environments* for IoT programs, *device farms* for mobile applications, *conditional testing*, *test prioritization and selection*, and *flaky tests*. Wherever possible, we include examples from IoT or microcontroller settings.

Unit Testing Frameworks. Constrained devices are still programmed primarily in low-level language such as C and C++. Many traditional unit testing frameworks are available for these languages, such as Google Test (GoogleTest, 2023), Boost.Test (Boost.Test team, 2023), CUTE (IFS Institut für Software, 2023), and bandit (Beyer and Karlsson, 2023). There are a handful of frameworks targeting microcontrollers explicitly, such as Unity (PlatformIO, 2023a, VanderVoord et al., 2015) and ArduinoTest (Murdoch, 2023). These work analogous to other unit testing frameworks, but are small enough to run on some constrained devices. While preferable over manual testing, these frameworks require the tests suites to be very small, since they are compiled and run along with the framework in their entirety on the device. In contrast, *Latch* allows arbitrarily large test suites.

Remote Testing. Latch's managed testing is adjacent to remote testing, but with some important differences. Remote testing is not a novel idea, for instance Jard et al. (1999) argued in 1999 that local synchronous tests can be translated to remote asynchronous tests without losing any testing power.

Remote testing has mostly been used to test distributed systems (Yao and Wang, 2005).

The RobotFramework (Robot Framework Foundation, 2023) for instance, is a large testing framework that supports remote testing via an RPC interface offering a transparent distribution model. As argued in many papers "distribution transparency is a myth that is both misleading and dangerous" (Guerraoui, 1999, Lea, 1997, Waldo et al., 1997). The Latch framework takes into account these lessons and offers the test engineer a testing framework with inherit timeouts and support for flaky tests, going well beyond the RobotFramework.

Some examples of remote testing frameworks can also be found for constrained devices. The popular PlatformIO project (PlatformIO, 2023a), uses the Unity framework (VanderVoord et al., 2015) for remote testing. However, it works significantly different from how *Latch* executes large test suites. While *Latch* allows arbitrarily large test suites by executing tests step-by-step, Unity does not address the memory constraints of the target devices as it compiles and uploads test suites as one monolithic executable. The framework also does not provide the debugger-like scripts (with custom actions) supported by Latch that enable the automation of standard hardware tests.

Holistic IoT Testing. Existing tools for IoT testing focus largely on testing networked systems of many devices holistically (Kanstrén et al., 2018, Popereshnyak et al., 2018), rather than the more common approach where components are tested selectively. Holistic testing of networked systems are by and large incompatible with many of the common development practices; such as test driven development for instance, which relies on selective testing of single components. Moreover, wholesale testing of heterogeneous system is very difficult, so many testing tools instead focus on monitoring to try and detect errors ("AppOptics – APM and Infrastructure Tool | SolarWinds AppOptics," n.d., Datadog, 2024). The few real testing frameworks available, tend to provide testing as a service (Kim et al., 2018). While holistic testing makes sense for IoT applications in industry, the approach makes far less sense for more consumer-oriented applications, such as smart home devices. Besides, developers cannot trust that end-to-end testing on such a high level, is enough to test IoT systems thoroughly. Neither does it lend itself well to test-driven development, as testing can only take place with a fully operational system. Therefore, there is a real need for selective—rather than holistic—testing of IoT software on microcontrollers. This is much easier with the single target testing in the style provided by *Latch*.

Scriptable Debugging. Latch's scriptable debugger-like hardware tests are inspired by scriptable debugging, which has been used in many other domains (Marceau et al., 2007). Scriptable debugging refers to all debugging techniques that can be controlled by developers through a programming language or similar tools such as regular expressions. Programmable debugging goes back to the early eighties, with many of the early proposals, such as Dispel (Johnson, 1981) and Dalek (Olsson et al., 1990), exploring variations on the concept of breakpoints. Recent work on a scriptable debugger API for Pharo (Dupriez et al., 2019), exposes a wide variety of advanced debugging operations, and allows developers to solve many challenging debugging scenarios through automated scripts. We are not aware of any framework which also applies the idea of scriptable debugging to testing in the context of constrained hardware.

Test Environments. A popular research topic in the domain of IoT testing, are heterogeneous test environments (Bures et al., 2020), where software can be distributed to nodes which are connected via a controlled network. This solution focuses on the challenging heterogeneity of IoT systems, and does not take into account the constraints the limited memory puts on the test suite size. Most test environments are virtual, and emulate the entire IoT environment (Nikolaidis et al., 2021, Ramprasad et al., 2019, Symeonides et al., 2020).

While, simulators are widely used for testing Internet of Things systems (Bures et al., 2020), they can never capture all the aspects of real hardware (Khan et al., 2011, Roska, 1990). For example, bugs caused by mistakes in interrupt handling, incomplete or wrong configuration, and concurrency faults (Makhshari and Mesbah, 2021) are typically not simulated. Because accurate hardware emulation is difficult, modern simulators often incorporate parts of the hardware under test, as is the case for *hardware-in-the-loop simulations* (Mihalǐc et al., 2022). Similarly, some test environments do allow hardware to be integrated into their test environments, but still fundamentally rely on virtualization (Behnke et al., 2019, Keahey et al., 2020). There are far fewer works that look into full hardware test environments (Adjih et al., 2015, Burin Des Rosiers et al., 2012, Gluhak et al., 2011). Using these large test environments can give more control to the developer to change various aspects of the nodes and network, such as packet loss, latency, and so on. However, setting up such large and often complex systems is complicated and time-consuming, for that reason they are often provided as a service (Beilharz et al., 2022, Kim et al., 2018). Subsequently, the test environments confine users to the specific choices in hardware, virtualization, and network technologies made by the service. While these test environments reduce the overhead of setting up a testing lab, they do not fundamentally help developers overcome the hardware limitations faced when executing large test suites.

Device Farms. These test environments are sometimes called testing farms or device farms in case they use real hardware, and are a popular approach for testing mobile applications (Fazzini and Orso, 2020, Huang, 2014). Curiously, testing on devices seems much more prevalent in the field of testing mobile applications (Kong et al., 2019). We believe this might be because mobile devices have far more memory than the embedded devices targeted by *Latch*, and therefore have no problem running large test suites. This strengthens our view that testing on constrained hardware presents a worthwhile research direction. However, the existing device farms heavily target mobile devices, and again limit users to the chosen technologies and hardware.

Conditional testing. Dependencies in *Latch* can be viewed as conditional skips for tests, where a test is skipped if any of the scenarios it depends on fail. Conditional skips have been around for some time in unit testing frameworks, such as the pytest framework for the Python language (Krekel and team, 2023), and the JUnit framework for Java (Bechtold et al., 2023). Pytest includes a **skipif** annotation which takes a boolean expression as its argument. In JUnit developers can use the **Assume** class, which provides a set of methods for conditional execution of tests. Modern frameworks targeting constrained devices (Murdoch, 2023, PlatformIO, 2023b) do not support conditional tests.

Test prioritization and selection. Another purpose of the dependencies in the test description language, are to determine the order tests are run in. Research on software testing has recently increased its attention to test prioritization and test selection (Pan et al., 2021). These techniques can also be applied to testing IoT systems (Medhat et al., 2020), where they are particularly useful since they can reduce large test suites to the most important tests, and help prioritize tests in such a way that regression tests fail as early as possible. An interesting line of future research could focus on integrating these techniques in *Latch*.

Flaky Tests. Flaky tests represent an active domain of research (Parry et al., 2021), which focuses on three problems: detecting flaky tests, finding root causes, and fixing flaky tests (Zolfaghari et al., 2021). The first step is to detect which tests are flaky. A popular approach is to look at the code coverage of tests (Bell et al., 2018, Zolfaghari et al., 2021). Once a flaky test is found, the next step is to find the root cause of the flaky test. This is a considerably harder problem, which is still being actively worked on (Lam et al., 2019). Alternatively, some research looks into automatically fixing, or preventing, flaky tests (Shi et al., 2019). All these techniques, from detection to fixing, are developed with the ultimate goal of mitigating and preventing flaky tests. In contrast, *Latch* focuses on providing a simple way of detecting and measuring the number of flaky tests in a test suite run. When evaluating *Latch*, we encountered flaky tests only rarely, but we believe that further research is warranted to assess the degree in which testing on constrained devices can cause flaky tests, and how existing techniques can mitigate them.

6.8 Conclusion

Testing is an essential part of the software development cycle which is currently very challenging on constrained devices. The limited memory and processing power of these constrained devices restrict the size of the test suite and makes testing slow, impeding a fast feedback loop. Moreover, due to the non-deterministic and unpredictable environment, tests can become flaky.

In this chapter, we answered the question of how to design and implement a testing framework for automatically running large-scale versatile tests on constrained systems. We introduce our novel testing framework *Latch* (Large-scale Automated Testing on Constrained Hardware), which needed to overcome three challenges; the memory constraints, processing constraints, and the timeouts and flaky tests. In essence, *Latch* splits test suites into small test instructions which are sent by a managing tester to a managed testee (constrained device). Because the constrained device receives the test instructions incrementally from the tester, it does not need to maintain the whole test suite in memory. By using an unconstrained tester to manage the constrained devices and the test suites, *Latch* is able to overcome the memory constraints.

Our testing framework further allows programmers to indicate the dependencies between related tests. This dependency information is used by *Latch* to skip tests that depend on previously failing tests, thus resulting in a faster feedback loop and helping the framework overcome the processing constraints of microcontrollers. On top of that *Latch* addresses the issue of timeouts and flaky tests, by including an analysis mode that provides feedback on timeouts and the flakiness of tests. Finally, the framework uses a novel approach of debugging-like instructions to allows developers to automate manual testing on hardware.

To demonstrate the efficacy and versatility of *Latch*, we showcased three use-cases, each pertaining to one stratum of the testing pyramid. The first use-case exemplifies unit testing, and showcases how we implemented a large suite of unit tests in *Latch* for a WebAssembly virtual machine intended for constrained devices. This test suite consists of 10,213 unit tests for a virtual machine running on a small ESP32 microcontroller. The second use-case illustrates integration testing of the instrumentation API in *Latch*. The third use-case highlights how the *Latch* test specification language allows programmers to write debugging-like testing scripts to test more elaborate testing scenarios, that mimic common manual testing tasks. Benchmarks show that the overhead of the testing framework is within expectation, roughly matching the performance difference between the constrained hardware and using a simulator on a workstation. Our test-cases shows that the testing framework is expressive, reliable, and reasonably fast, making it suitable to run large test suites on constrained devices.

Bibliography

Adjih, C., Baccelli, E., Fleury, E., Harter, G., Mitton, N., Noel, T., Pissard-Gibollet, R., Saint-Marcel, F., Schreiner, G., Vandaele, J., Watteyne, T., 2015. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed, in: IEEE World Forum on Internet of Things, WF-IoT 2015 - Proceedings. pp. 459–464.. https://doi.org/10.1109/WF-IoT.2015.7389098

Agrawal, H., De Millo, R., Spafford, E., 1991. An Execution-Backtracking Approach to Debugging. IEEE Software 8, 21–26.. https://doi.org/10.1109/52.88940

Almeida, P.S., 2024. Approaches to Conflict-free Replicated Data Types. ACM Comput. Surv. 57, 1–36.. https://doi.org/10.1145/3695249

Alter the Program's Execution Flow, 2024.

Annamaa, A., 2015. Introducing Thonny, a Python IDE for Learning Programming, in: Proceedings of the 15th Koli Calling Conference on Computing Education Research, Koli Calling '15. Association for Computing Machinery, New York, NY, USA, pp. 117–121.. https://doi.org/10.1145/2828959.2828969

AppOptics – APM and Infrastructure Tool | SolarWinds AppOptics, n.d.

Aspencore, 2023. Embedded Survey 2023: More IP Reuse as Workloads Surge.

The AssemblyScript Project, 2023. AssemblyScript.

Auguston, M., Michael, J.B., Shing, M.-T., 2005. Environment Behavior Models for Scenario Generation and Testing Automation. SIGSOFT Softw. Eng. Notes 30, 1–6.. https://doi.org/10.1145/1082983.1083284

Axelsen, H.B., Glück, R., 2016. On Reversible Turing Machines and Their Function Universality. Acta Informatica 53, 509–543.. https://doi.org/10.1007/s00236-015-0253-y

Aycock, J., 2003. A Brief History of Just-in-Time. ACM Comput. Surv. 35, 97–113.. https://doi.org/10.1145/857076.857077

Baccelli, E., Doerr, J., Jallouli, O., Kikuchi, S., Morgenstern, A., Padilla, F.A., Schleiser, K., Thomas, I., 2018. Reprogramming Low-end IoT Devices from the Cloud, in: 2018 3rd Cloudification of the Internet of Things (CIoT). pp. 1–6.. https://doi.org/10.1109/CIOT.2018.8627129

Bahlke, R., Snelting, G., 1986. The PSG System: From Formal Language Definitions to Interactive Programming Environments. ACM Trans. Program. Lang. Syst. 8, 547–576.. https://doi.org/10.1145/6465.20890

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A Survey of Symbolic Execution Techniques. ACM Comput. Surv. 51, 1–39.. https://doi.org/10.1145/3182657

Balzer, R.M., 1969. EXDAMS: Extendable Debugging and Monitoring System, in: Proceedings of the May 14-16, 1969, Spring Joint Computer Conference, AFIPS '69 (Spring). Association for Computing Machinery, New York, NY, USA, pp. 567–580.. https://doi.org/10.1145/1476793.1476881

Banks, A., Gupta, R., 2014. MQTT Version 3.1. 1. OASIS standard 29.

Banzi, M., 2008. Getting Started with Arduino, 99th ed. Make Books - Imprint of: O'Reilly Media, Sebastopol, CA.

Battagline, R., 2021. The Art of WebAssembly: Build Secure, Portable, High-Performance Applications. No Starch Press.

Bechtold, S., Brannen, S., Link, J., Merdes, M., Philipp, M., de Rancourt, J., Stein, C., 2023. JUnit 5 User Guide.

Behnke, I., Thamsen, L., Kao, O., 2019. Héctor: A Framework for Testing IoT Applications Across Heterogeneous Edge and Cloud Testbeds, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC '19 Companion. Association for Computing Machinery, New York, NY, USA, pp. 15–20.. https://doi.org/10.1145/3368235.3368832

Beilharz, J., Wiesner, P., Boockmeyer, A., Pirl, L., Friedenberger, D., Brokhausen, F., Behnke, I., Polze, A., Thamsen, L., 2022. Continuously Testing Distributed IoT Systems: An Overview of~the~State of~the~Art, in: Hacid, H., Aldwairi, M., Bouadjenek, M.R., Petrocchi, M., Faci, N., Outay, F., Beheshti, A., Thamsen, L., Dong, H. (Eds.), Service-Oriented Computing – ICSOC 2021 Workshops. Springer International Publishing, Cham, pp. 336–350.. https://doi.org/10.1007/978-3-031-14135-5_30

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D., 2018. DeFlaker: Automatically Detecting Flaky Tests, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). pp. 433–444.. https://doi.org/10.1145/3180155.3180164

Bennett, C.H., 1988. Notes on the History of Reversible Computation. IBM Journal of Research and Development 32, 16–23.. https://doi.org/10.1147/rd.321.0016

Bernardo, M., Lanese, I., Marin, A., Mezzina, C.A., Rossi, S., Sacerdoti Coen, C., 2023. Causal Reversibility Implies Time Reversibility, in: Jansen, N., Tribastone, M. (Eds.), Quantitative Evaluation of Systems. Springer

Nature Switzerland, Cham, pp. 270–287.. https://doi.org/10.1007/978-3-031-43835-6_19

Bernstein, K.L., Stark, E.W., 1995. Operational Semantics of a Focusing Debugger. Electronic Notes in Theoretical Computer Science, MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference 1, 13–31.. https://doi.org/10.1016/S1571-0661(04)80002-1

Beyer, S., Karlsson, J., 2023. Bandit. Human-friendly Unit Testing for C++11.

Black, A.P., Nierstrasz, O., Ducasse, S., Pollet, D., 2010. Pharo by Example. Lulu. com.

Blackburn, M., 1998. Using Models for Test Generation and Analysis, in: 17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267). p. C45/1–C45/8 vol.1.. https://doi.org/10.1109/DASC.1998.741501

Bojanova, I., Eduardo Galhardo, C., 2021. Classifying Memory Bugs Using Bugs Framework Approach, in: 2021 IEEE 45th Annual Computers, Software, And Applications Conference (COMPSAC). pp. 1157–1164.. https://doi.org/10.1109/COMPSAC51774.2021.00159

Boost.Test team, 2023. What Is Boost.Test?.

Boothe, B., 2000. Efficient Algorithms for Bidirectional Debugging, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00. Association for Computing Machinery, New York, NY, USA, pp. 299–310.. https://doi.org/10.1145/349299.349339

Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J., 1987. Clean — A Language for Functional Graph Rewriting, in: Kahn, G. (Ed.), Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 364–384.. https://doi.org/10.1007/3-540-18317-5_20

Bures, M., Klima, M., Rechtberger, V., Bellekens, X., Tachtatzis, C., Atkinson, R., Ahmed, B.S., 2020. Interoperability and Integration Testing Methods for IoT Systems: A Systematic Mapping Study, in: de Boer, F., Cerone, A. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 93–112.

Burg, B., Bailey, R., Ko, A.J., Ernst, M.D., 2013. Interactive Record/Replay for Web Application Debugging, in: Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology, UIST '13.

Association for Computing Machinery, New York, NY, USA, pp. 473–484.. https://doi.org/10.1145/2501988.2502050

Burin Des Rosiers, C., Chelius, G., Fleury, E., Fraboulet, A., Gallais, A., Mitton, N., Noël, T., 2012. SensLAB: Very Large Scale Open Wireless Sensor Network Testbed. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering 239–254.. https://doi.org/10.1007/978-3-642-29273-6_19

Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M., 2017. Control-Flow Integrity: Precision, Security, and Performance. ACM Comput. Surv. 50, 1–33.. https://doi.org/10.1145/3054924

Cadar, C., Godefroid, P., Khurshid, S., Ṗašareanu, C.S., Sen, K., Tillmann, N., Visser, W., 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11. Association for Computing Machinery, New York, NY, USA, pp. 1066–1071.. https://doi.org/10.1145/1985793.1985995

Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F., 2019. Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types. Journal of Systems and Software 152, 165–181.. https://doi.org/10.1016/j.jss.2019.03.002

Chen, S.-K., Fuchs, W., Chung, J.-Y., 2001. Reversible Debugging Using Program Instrumentation. IEEE Transactions on Software Engineering 27, 715–727.. https://doi.org/10.1109/32.940726

Church, A., 1940. A Formulation of the Simple Theory of Types. Journal of Symbolic Logic 5, 56–68.. https://doi.org/10.2307/2266170

Clang contributors, 2021. Clang: A C Language Family Frontend for LLVM.

Cohen, I., 1994. The Use of "Bug" in Computing. IEEE Annals of the History of Computing 16, 54–55.. https://doi.org/10.1109/85.279235

Cohn, M., 2009. Succeeding with Agile: Software Development Using Scrum, 1th. ed. Addison-Wesley Professional, Boston, MA, USA.

D'Silva, V., Kroening, D., Weissenbacher, G., 2008. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27, 1165–1178.. https://doi.org/10.1109/TCAD.2008.923410

da Silva, F.Q.B., 1992. Correctness Proofs of Compilers and Debuggers : An Approach Based on Structural Operational Semantics.

Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M., 1999. Model-Based Testing in Practice, in: Proceedings of the 21st International Conference on Software Engineering, ICSE '99. Association for Computing Machinery, New York, NY, USA, pp. 285–294.. https://doi.org/10.1145/302405.302640

Danos, V., Krivine, J., 2004. Reversible Communicating Systems, in: Gardner, P., Yoshida, N. (Eds.), CONCUR 2004 - Concurrency Theory. Springer, Berlin, Heidelberg, pp. 292–307.. https://doi.org/10.1007/978-3-540-28644-8_19

Datadog, 2024. End to End Testing Automation.

De Sio, C., Azimi, S., Sterpone, L., 2023. Evaluating Reliability against SEE of Embedded Systems: A Comparison of RTOS and Bare-Metal Approaches. Microelectronics Reliability 115124.. https://doi.org/10.1016/j.microrel.2023.115124

de Troyer, C., Nicolay, J., de Meuter, W., 2018. Building IoT Systems Using Distributed First-Class Reactive Programming, in: 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). pp. 185–192.. https://doi.org/10.1109/CloudCom2018.2018.00045

Deiner, A., Fraser, G., 2024. NuzzleBug: Debugging Block-Based Programs in Scratch, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, Lisbon, Portugal, pp. 1–13.. https://doi.org/10.1145/3597503.3623331

Dessouky, G., Gens, D., Haney, P., Persyn, G., Kanuparthi, A., Khattri, H., Fung, J.M., Sadeghi, A.-R., Rajendran, J., 2018. When a Patch Is Not Enough - HardFails: Software-Exploitable Hardware Bugs.. https://doi.org/10.48550/arXiv.1812.00197

Dupriez, T., Polito, G., Costiou, S., Aranega, V., Ducasse, S., 2019. Sindarin: A Versatile Scripting API for the Pharo Debugger, in: Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019. Association for Computing Machinery, New York, NY, USA, pp. 67–79.. https://doi.org/10.1145/3359619.3359745

Engblom, J., 2012. A Review of Reverse Debugging, in: Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference. pp. 1–6.

English, K.V., Obaidat, I., Sridhar, M., 2019. Exploiting Memory Corruption Vulnerabilities in Connman for IoT Devices, in: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 247–255.. https://doi.org/10.1109/DSN.2019.00036

Espressif Systems, 2023a. ESP-IDF Programming Guide.

Espressif Systems, 2023b. ESPRESSIF. Espressif Offers Integrated, Reliable and Energy-Efficient Wireless SoCs.

Fard, A.M., Mesbah, A., 2017. JavaScript: The (Un)Covered Parts, in: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST. IEEE, New York, NY, USA, pp. 230–240.. https://doi.org/10.1109/ICST.2017.28

Fazzini, M., Orso, A., 2020. Managing App Testing Device Clouds: Issues and Opportunities, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1257–1259.

Feldman, S.I., Brown, C.B., 1988. IGOR: A System for Program Debugging via Reversible Execution. ACM SIGPLAN Notices 24, 112–123.. https://doi.org/10.1145/69215.69226

Felleisen, M., Findler, R.B., Flatt, M., 2009. Semantics Engineering with PLT Redex. Mit Press.

Fermyon Technologies, Inc., 2023. WebAssembly Language Support Matrix.

Ferrari, G., Tuosto, E., 2001. A Debugging Calculus for Mobile Ambients, in: Proceedings of the 2001 ACM Symposium on Applied Computing. ACM, Las Vegas Nevada USA.. https://doi.org/10.1145/372202.380701

Ferreira Ruiz, F., Collins, B., al, \.e., 2024. Open Bot Brain.

Flanagan, D., 2020. JavaScript: The Definitive Guide, 7th. ed. O'Reilly Media, Inc., Sebastopol, CA, USA.

Frattini, F., Pietrantuono, R., Russo, S., 2016. Reproducibility of Software Bugs. Principles of Performance and Reliability Modeling and Evaluation: Essays in Honor of Kishor Trivedi on His 70th Birthday.. https://doi.org/10.1007/978-3-319-30599-8_21

Free Software Foundation, n.d.. GDB: The GNU Project Debugger.

Gait, J., 1986. A Probe Effect in Concurrent Programs. Software: Practice and Experience 16, 225–233.. https://doi.org/10.1002/spe.4380160304

García-Ferreira, I., Laorden, C., Santos, I., Bringas, P.G., 2014. A Survey on Static Analysis and Model Checking, in: de la Puerta, J.G., Ferreira, I.G., Bringas, P.G., Klett, F., Abraham, A., de Carvalho, A.C., Herrero, Á., Baruque, B., Quintián, H., Corchado, E. (Eds.), International Joint Conference SOCO'14-CISIS'14-ICEUTE'14. Springer International Publishing, Cham, pp. 443–452.. https://doi.org/10.1007/978-3-319-07995-0_44

George, D., 2021. MicroPython.

Giachino, E., Lanese, I., Mezzina, C.A., 2014. Causal-Consistent Reversible Debugging, in: Gnesi, S., Rensink, A. (Eds.), Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 370–384.. https://doi.org/10.1007/978-3-642-54804-8_26

Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., Razafindralambo, T., 2011. A Survey on Facilities for Experimental Internet of Things Research. IEEE Communications Magazine 49, 58–67.. https://doi.org/10.1109/MCOM.2011.6069710

Glück, R., Yokoyama, T., 2023. Reversible Computing from a Programming Language Perspective. Theoretical Computer Science 953, 113429.. https://doi.org/10.1016/j.tcs.2022.06.010

Godefroid, P., 1997. Model Checking for Programming Languages Using VeriSoft, in: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97. Association for Computing Machinery, New York, NY, USA, pp. 174–186.. https://doi.org/10.1145/263699.263717

Godefroid, P., Klarlund, N., Sen, K., 2005. DART: Directed Automated Random Testing. SIGPLAN Not. 40, 213–223.. https://doi.org/10.1145/1064978.1065036

GoogleTest, 2023. GoogleTest User's Guide.

Gosling, J., Joy, B., Steele, G.L., 1996. The Java Language Specification, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Guerraoui, R., 1999. What Object-Oriented Distributed Programming Does Not Have to Be, and What It May Be. Informatik.

Gupta, D., Jalote, P., Barua, G., 1996. A Formal Framework for On-Line Software Version Change. IEEE Transactions on Software Engineering 22, 120–131.. https://doi.org/10.1109/32.485222

Gurdeep Singh, R., 2022. Taming Nondeterminism : Programming Language Abstractions and Tools for Dealing with Nondeterministic Programs.

Gurdeep Singh, R., Scholliers, C., 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. Association for Computing Machinery, New York, NY, USA, pp. 27–36.. https://doi.org/10.1145/3357390.3361029

Gurdeep Singh, R., Torres Lopez, C., Marr, S., Gonzalez Boix, E., Scholliers, C., 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Artifact). Dagstuhl Artifacts Series 5, 1–3.. https://doi.org/10.4230/DARTS.5.2.4

Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, \.J., 2017. Bringing the Web up to Speed with WebAssembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017. Association for Computing Machinery, New York, NY, USA, pp. 185–200.. https://doi.org/10.1145/3062341.3062363

Haleby, J., n.d.. Awaitility.

Hambarde, P., Varma, R., Jha, S., 2014. The Survey of Real Time Operating System: RTOS, in: 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies. pp. 34–39.. https://doi.org/10.1109/ICESC.2014.15

Harman, M., O'Hearn, P., 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis, in: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 1–23.. https://doi.org/10.1109/SCAM.2018.00009

Haulund, T., Mogensen, T.Æ., Glück, R., 2017. Implementing Reversible Object-Oriented Language Features on Reversible Machines, in: Phillips, I., Rahaman, H. (Eds.), Reversible Computation. Springer International Publishing, Cham, pp. 66–73.. https://doi.org/10.1007/978-3-319-59936-6_5

Hay-Schmidt, L., Glück, R., Cservenka, M.H., Haulund, T., 2021. Towards a Unified Language Architecture for Reversible Object-Oriented Programming, in: Yamashita, S., Yokoyama, T. (Eds.), Reversible Computation. Springer International Publishing, Cham, pp. 96–106.. https://doi.org/10.1007/978-3-030-79837-6_6

Herlihy, M.P., Wing, J.M., 1990. Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. 12, 463–492.. https://doi.org/10.1145/78969.78972

Heunen, C., Karvonen, M., 2015. Reversible Monadic Computing. Electronic Notes in Theoretical Computer Science, The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). 319, 217–237.. https://doi.org/10.1016/j.entcs.2015.12.014

Hickey, P., Konka, J., Gohman, D., Clegg, S., Brown, A., Crichton, A., Clark, L., Ihrig, C., Huene, P., Yuji, \.Y., Vasilik, D., Triplett, J., Rubanov, S., Akbary, S., Frysinger, M., Turner, A., Zakai, A., Mackenzie, A., Brittain, B., Beyer, C., McKay, D., Wang, L., Mielniczuk, M., Berger, M., PTrottier, Sikora, P., Schneidereit, T., martin, k., nasso, 2020. WebAssembly/WASI: Snapshot-01.. https://doi.org/10.5281/zenodo.4323447

Hoey, J., Ulidowski, I., Yuen, S., 2018. Reversing Parallel Programs with Blocks and Procedures.. https://doi.org/10.48550/arXiv.1808.08651

Holter, K., Hennoste, J.O., Lam, P., Saan, S., Vojdani, V., 2024. Abstract Debuggers: Exploring Program Behaviors Using Static Analysis Results, in: Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, And Reflections on Programming and Software, Onward! '24. Association for Computing Machinery, New York, NY, USA, pp. 130–146.. https://doi.org/10.1145/3689492.3690053

Huang, J.-f., 2014. AppACTS: Mobile App Automated Compatibility Testing Service, in: 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, And Engineering. pp. 85–90.. https://doi.org/10.1109/MobileCloud.2014.13

Huang, W., Wang Xin, 2021. WebAssembly Micro Runtime.

Högl, H., Rath, D., 2006. Open On-Chip Debugger–Openocd–. Fakultat fur Informatik, Tech. Rep.

IEEE Standard for Test Access Port and Boundary-Scan Architecture, 2013. . IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001) 1–444.. https://doi.org/10.1109/IEEESTD.2013.6515989

Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C., 1996. Lua—An Extensible Extension Language. Software: Practice and Experience 26, 635–652.. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

IFS Institut für Software, 2023. CUTE. C++ Unit Testing Easier.

Iqbal, M.Z., Arcuri, A., Briand, L., 2015. Environment Modeling and Simulation for Automated Testing of Soft Real-Time Embedded Software. Software & Systems Modeling 14, 483–524.. https://doi.org/10.1007/s10270-013-0328-6

ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary, 2017. . ISO/IEC/IEEE 24765:2017(E) 1–541.. https://doi.org/10.1109/IEEESTD.2017.8016712

Jangda, A., Powers, B., Berger, E.D., Guha, A., 2019. Not So Fast: Analyzing the Performance of {\vphantom }WebAssembly\vphantom {} vs. Native Code, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19). pp. 107–120.

Jard, C., Jéron, T., Tanguy, L., Viho, C., 1999. Remote Testing Can Be as Powerful as Local Testing. Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999, Beijing, China.. https://doi.org/10.1007/978-0-387-35578-8_2

JetBrains s.r.o., 2023. WebStorm. The Smartest JavaScript IDE.

Jhala, R., Majumdar, R., 2009. Software Model Checking. ACM Comput. Surv. 41, 1–54.. https://doi.org/10.1145/1592434.1592438

Johnson, M., 1981. Dispel: A Run-Time Debugging Language. Computer Languages 6, 79–94.. https://doi.org/10.1016/0096-0551(81)90068-0

The JUnit Team, n.d.. JUnit 5.

Kahlon, V., Wang, C., Gupta, A., 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique, in: Bouajjani, A., Maler, O. (Eds.), Computer Aided Verification. Springer, Berlin, Heidelberg, pp. 398–413.. https://doi.org/10.1007/978-3-642-02658-4_31

Kanstrén, T., Mäkelä, J., Karhula, P., 2018. Architectures and Experiences in Testing IoT Communications, in: Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018. pp. 98–103.. https://doi.org/10.1109/ICSTW.2018.00034

Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H.S., Hammock, C., Mambretti, J., Barnes, A., Halbah, F., Rocha, A., Stubbs, J., 2020. Lessons Learned from the Chameleon Testbed, in: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 219–233.

Kelly, F., 1981. Reversibility and Stochastic Networks / F.P. Kelly. SERBIULA (sistema Librum 2.0) 76.. https://doi.org/10.2307/2287860

Kemeny, J.G., Kurtz, T.E., Cochran, D.S., 1968. Basic: A Manual for BASIC, the Elementary Algebraic Language Designed for Use with the Dartmouth Time Sharing System. Dartmouth Publications.

Kennedy, H.C., 1974. Peano's Concept of Number. Historia Mathematica 1, 387–408.. https://doi.org/10.1016/0315-0860(74)90031-7

Kernighan, B.W., Ritchie, D.M., 1989. The C Programming Language. Prentice Hall Press, USA.

Kery, M.B., Myers, B.A., 2017. Exploring Exploratory Programming, in: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 25–29.. https://doi.org/10.1109/VLHCC.2017.8103446

Kery, M.B., Radensky, M., Arya, M., John, B.E., Myers, B.A., 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool, in: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18. Association for Computing Machinery, New York, NY, USA, pp. 1–11.. https://doi.org/10.1145/3173574.3173748

Khan, M.Z., Askwith, B., Bouhafs, F., Asim, M., 2011. Limitations of Simulation Tools for Large-Scale Wireless Sensor Networks, in: 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications. IEEE, New York, NY, USA, pp. 820–825.. https://doi.org/10.1109/WAINA.2011.59

Kim, H., Ahmad, A., Hwang, J., Baqa, H., Le Gall, F., Reina Ortega, M.A., Song, J., 2018. IoT-TaaS: Towards a Prospective IoT Testing Framework. IEEE Access 6, 15480–15493.. https://doi.org/10.1109/ACCESS.2018.2802489

King, J.C., 1976. Symbolic Execution and Program Testing. Communications of the ACM 19, 385–394.. https://doi.org/10.1145/360248.360252

Klimushenkova, M.A., Dovgalyuk, P.M., 2017. Improving the Performance of Reverse Debugging. Programming and Computer Software 43, 60–66.. https://doi.org/10.1134/S0361768817010042

Koji, Y., others, 2023. Mruby-Arduino. Mruby-Arduino Is Wrapper Mrbgem for Arduino API.

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T.F., Klein, J., 2019. Automated Testing of Android Apps: A Systematic Literature Review. IEEE Transactions on Reliability 68, 45–66.. https://doi.org/10.1109/TR.2018.2865733

Kopetz, H., Ochsenreiter, W., 1987. Clock Synchronization in Distributed Real-Time Systems. IEEE Transactions on Computers 933–940.. https://doi.org/10.1109/TC.1987.5009516

Krekel, H., team, p.-d., 2023. Pytest: Helps You Write Better Programs.

Kurshan, R., Levin, V., Minea, M., Peled, D., Yenigün, H., 1998. Static Partial Order Reduction, in: Steffen, B. (Ed.), Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, pp. 345–357.. https://doi.org/10.1007/BFb0054182

Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S., 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Issta 2019. Association for Computing Machinery, New York, NY, USA, pp. 101–111.. https://doi.org/10.1145/3293882.3330570

Lami, P., Lanese, I., Stefani, J.-B., 2024. A Small-Step Semantics for~Janus, in: Mogensen, T.Æ., Mikulski, Ł. (Eds.), Reversible Computation. Springer Nature Switzerland, Cham, pp. 105–123.. https://doi.org/10.1007/978-3-031-62076-8_8

Lamport, 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers 690–691.. https://doi.org/10.1109/TC.1979.1675439

Lamport, L., 1978. Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21, 558–565.. https://doi.org/10.1145/359545.359563

Lanese, I., 2018. From Reversible Semantics to Reversible Debugging, in: Kari, J., Ulidowski, I. (Eds.), Reversible Computation, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 34–46.. https://doi.org/10.1007/978-3-319-99498-7_2

Lanese, I., Mezzina, C., Tiezzi, F., 2014. Causal-Consistent Reversibility.

Lanese, I., Nishida, N., Palacios, A., Vidal, G., 2018. CauDEr: A Causal-Consistent Reversible Debugger for Erlang, in: Gallagher, J.P., Sulzmann, M. (Eds.), Functional and Logic Programming, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 247–263.. https://doi.org/10.1007/978-3-319-90686-7_16

Laursen, J.S., Ellekilde, L.-P., Schultz, U.P., 2018. Modelling Reversible Execution of Robotic Assembly. Robotica 36, 625–654.. https://doi.org/10.1017/S0263574717000613

Lauwaerts, T., Castillo, C.R., Singh, R.G., Marra, M., Scholliers, C., Gonzalez Boix, E., 2022. Event-Based Out-of-Place Debugging, in: Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, MPLR '22. Association for Computing Machinery, New York, NY, USA, pp. 85–97.. https://doi.org/10.1145/3546918.3546920

Lauwaerts, T., Singh, R.G., Scholliers, C., 2024. WARDuino: An Embedded WebAssembly Virtual Machine. Journal of Computer Languages 101268.. https://doi.org/10.1016/j.cola.2024.101268

Layman, L., Diep, M., Nagappan, M., Singer, J., Deline, R., Venolia, G., 2013. Debugging Revisited: Toward Understanding the Debugging Needs of Contemporary Software Developers, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 383–392.. https://doi.org/10.1109/ESEM.2013.43

Lea, D., 1997. Design for Open Systems in Java, in: Garlan, D., Le Métayer, D. (Eds.), Coordination Languages and Models. Springer, Berlin, Heidelberg, pp. 32–45.. https://doi.org/10.1007/3-540-63383-9_71

Lee, K., Lee, Y., Lee, H., Yim, K., 2016. A Brief Review on JTAG Security, in: 2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS). pp. 486–490.. https://doi.org/10.1109/IMIS.2016.102

Leeman, G.B., 1986. A Formal Approach to Undo Operations in Programming Languages. ACM Transactions on Programming Languages and Systems 8, 50–87.. https://doi.org/10.1145/5001.5005

Lehmann, D., Kinder, J., Pradel, M., 2020. Everything Old Is New Again: Binary Security of Webassembly, in: Proceedings of the 29th USENIX Conference on Security Symposium, SEC'20. USENIX Association, USA, pp. 217–234.

Levis, P., Culler, D., 2002. Maté: A Tiny Virtual Machine for Sensor Networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X. Association for Computing Machinery, New York, NY, USA, pp. 85–95.. https://doi.org/10.1145/605397.605407

Lewis, B., 2003. Debugging Backwards in Time.. https://doi.org/10.48550/arXiv.cs/0310016

Li, C., Chen, R., Wang, B., Wang, Z., Yu, T., Jiang, Y., Gu, B., Yang, M., 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, ISSTA 2023. Association for Computing Machinery, New York, NY, USA, pp. 1345–1356.. https://doi.org/10.1145/3597926.3598140

Li, H., Xu, Y., Wu, F., Yin, C., 2009. Research of ``Stub'' Remote Debugging Technique, in: 2009 4th International Conference on Computer Science & Education. pp. 990–994.. https://doi.org/10.1109/ICCSE.2009.5228140

Li, W., Li, N., 2012. A Formal Semantics for Program Debugging. Science China Information Sciences 55, 133–148.. https://doi.org/10.1007/s11432-011-4530-2

Lieberman, H., 1997. ZStep 95: A Reversible, Animated Source Code Stepper. Software Visualization: Programming as a Multimedia Experience.

Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII. Association for Computing Machinery, New York, NY, USA, pp. 329–339.. https://doi.org/10.1145/1346281.1346323

Lubbers, M., Koopman, P., Plasmeijer, R., 2021. Interpreting Task Oriented Programs on Tiny Computers, in: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL '19. Association for Computing Machinery, New York, NY, USA, pp. 1–12.. https://doi.org/10.1145/3412932.3412936

Lutz, C., Derby, H., 1986. Janus: A Time-Reversible Language. Letter to R. Landauer 2.

Madsen, M., Lhoták, O., Tip, F., 2017. A Model for Reasoning about JavaScript Promises. Proc. ACM Program. Lang. 1, 1–24.. https://doi.org/10.1145/3133910

Makhshari, A., Mesbah, A., 2021. IoT Bugs and Development Challenges, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 460–472.. https://doi.org/10.1109/ICSE43902.2021.00051

Maksimovic, M., Vujovic, V., Davidović, N., Milosevic, V., Perisic, B., 2014. Raspberry Pi as Internet of Things Hardware: Performances and Constraints, in: Proceedings of the 1st International Conference on Electrical, Electronic and Computing Engineering.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E., 2010. The Scratch Programming Language and Environment. ACM Transactions on Computing Education 10, 1–15.. https://doi.org/10.1145/1868358.1868363

Mani, S.K., Durairajan, R., Barford, P., Sommers, J., 2018. An Architecture for IoT Clock Synchronization, in: Proceedings of the 8th International Conference on the Internet of Things, IOT '18. Association for Computing Machinery, New York, NY, USA, pp. 1–8.. https://doi.org/10.1145/3277593.3277606

Marceau, G., Cooper, G., Spiro, J., Krishnamurthi, S., Reiss, S., 2007. The Design and Implementation of a Dataflow Language for Scriptable Debugging. Automated Software Engineering 14, 59–86.. https://doi.org/10.1007/s10515-006-0003-z

Marques, F., Fragoso Santos, J., Santos, N., Adão, P., 2022. Concolic Execution for WebAssembly, in: Ali, K., Vitek, J. (Eds.), 36th European Conference on Object-Oriented Programming (ECOOP 2022), Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 1–29.. https://doi.org/10.4230/LIPIcs.ECOOP.2022.11

Marra, M., Polito, G., Gonzalez Boix, E., 2018. Out-Of-Place Debugging: A Debugging Architecture to Reduce Debugging Interference. The Art, Science, and Engineering of Programming 3, 1–29.. https://doi.org/10.22152/programming-journal.org/2019/3/3

Massey, S., Shymanskyy, V., 2022. M3.

Massey, S., Shymanskyy, V., 2021. WASM3.

Matsuda, K., Wang, M., 2020. Sparcl: A Language for Partially-Invertible Computation. Proceedings of the ACM on Programming Languages 4, 1–31.. https://doi.org/10.1145/3409000

McBride, J., 1989. Electrical Contact Bounce in Medium-Duty Contacts. IEEE Transactions on Components, Hybrids, and Manufacturing Technology 12, 82–90.. https://doi.org/10.1109/33.19016

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., Zander, C., 2008. Debugging: A Review of the Literature from an Educational Perspective. Computer Science Education 18, 67–92.. https://doi.org/10.1080/08993400802114581

McDonald, C., others, 2023. Mruby-Esp32-System. System Library for Mruby-Esp32.

McDowell, C.E., Helmbold, D.P., 1989. Debugging Concurrent Programs. ACM Comput. Surv. 21, 593–622.. https://doi.org/10.1145/76894.76897

Medhat, N., Moussa, S.M., Badr, N.L., Tolba, M.F., 2020. A Framework for Continuous Regression and Integration Testing in IoT Systems Based on Deep Learning and Search-Based Techniques. IEEE Access 8, 215716–215726.. https://doi.org/10.1109/ACCESS.2020.3039931

Mezzina, C.A., Schlatte, R., Glück, R., Haulund, T., Hoey, J., Holm Cservenka, M., Lanese, I., Mogensen, T.Æ., Siljak, H., Schultz, U.P., Ulidowski, I., 2020. Software and Reversible Systems: A~Survey of Recent Activities.

Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405, Lecture Notes in Computer Science.. https://doi.org/10.1007/978-3-030-47361-7_2

Microsoft, 2023. The TypeScript Handbook.

Mihalǐc, F., Truntǐc, M., Hren, A., 2022. Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges. Electronics 11, 2462.. https://doi.org/10.3390/electronics11152462

Mishra, B., Kertesz, A., 2020. The Use of MQTT in M2M and IoT Systems: A Survey. IEEE Access 8, 201071–201086.. https://doi.org/10.1109/ACCESS.2020.3035849

Mogul, J.C., 2006. Emergent (Mis)Behavior vs. Complex Software Systems. SIGOPS Oper. Syst. Rev. 40, 293–304.. https://doi.org/10.1145/1218063.1217964

Mukhin, V., Kornaga, Y., Bazaka, Y., Krylov, I., Barabash, A., Yakovleva, A., Mukhin, O., 2021. The Testing Mechanism for Software and Services Based on Mike Cohn's Testing Pyramid Modification, in: Proceedings of the 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2021. pp. 589–595.. https://doi.org/10.1109/IDAACS53288.2021.9660999

Murdoch, M., 2023. ArduinoUnit.

nanoFramework Contributors, 2021.NET nanoFramework Documentation.

Nikolaidis, F., Marazakis, M., Bilas, A., 2021. IOTier: A Virtual Testbed to Evaluate Systems for IoT Environments, in: Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021. pp. 676–683.. https://doi.org/10.1109/CCGrid51090.2021.00081

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., Partush, N., 2017. Engineering Record and Replay for Deployability, in: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 377–389.

Olsson, R.A., Crawford, R.H., Ho, W.W., 1990. Dalek: A GNU, Improved Programmable Debugger., in: USENIX Technical Conference. The USENIX Association, Berkeley, CA, USA, pp. 221–231.

OpenJS Foundation, n.d.. Mocha - the Fun, Simple, Flexible JavaScript Test Framework.

Ozcan, M.O., Odaci, F., Ari, I., 2019. Remote Debugging for Containerized Applications in Edge Computing Environments, in: 2019 IEEE International Conference on Edge Computing (EDGE). pp. 30–32.. https://doi.org/10.1109/EDGE.2019.00021

Pan, R., Bagherzadeh, M., Ghaleb, T.A., Briand, L., 2021. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. Empirical Software Engineering 27, 29.. https://doi.org/10.1007/s10664-021-10066-6

Parker, D., 2015. JavaScript with Promises, 1st ed. O'Reilly Media, Inc.

Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P., 2021. A Survey of Flaky Tests. ACM Trans. Softw. Eng. Methodol. 31, 1–74.. https://doi.org/10.1145/3476105

Pasquier, M., Teodorov, C., Jouault, F., Brun, M., Lagadec, L., 2023a. Debugging Paxos in the UML Multiverse, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). pp. 811–820.. https://doi.org/10.1109/MODELS-C59198.2023.00130

Pasquier, M., Teodorov, C., Jouault, F., Brun, M., Le Roux, L., Lagadec, L., 2023b. Temporal Breakpoints for Multiverse Debugging, in: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023. Association for Computing Machinery, New York, NY, USA, pp. 125–137.. https://doi.org/10.1145/3623476.3623526

Pasquier, M., Teodorov, C., Jouault, F., Brun, M., Roux, L.L., Lagadec, L., 2022. Practical Multiverse Debugging through User-Defined Reductions: Application to UML Models, in: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems, MODELS '22. Association for Computing Machinery, New York, NY, USA, pp. 87–97.. https://doi.org/10.1145/3550355.3552447

Peano, G., 1891. Sul Concetto Di Numero. Rivista Matematica 1, 256–267.

Peress, Y., Nashif, A., Brunnen, M., Andersen, H.B., Emeltchenko, A., Massey, A.E., Bolivar, M., Olivares, I.H., 2024. Test Framework — Zephyr Project Documentation.

Perez, F., Granger, B.E., 2007. IPython: A System for Interactive Scientific Computing. Computing in Science & Engineering 9, 21–29.. https://doi.org/10.1109/MCSE.2007.53

Perrin, M., Mostefaoui, A., Jard, C., 2016. Causal Consistency: Beyond Memory, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16. Association for Computing Machinery, New York, NY, USA, pp. 1–12.. https://doi.org/10.1145/2851141.2851170

Phipps-Costin, L., Rossberg, A., Guha, A., Leijen, D., Hillerström, D., Sivaramakrishnan, \.K., Pretnar, M., Lindley, S., 2023. Continuing WebAssembly with Effect Handlers. Proceedings of the ACM on Programming Languages 7, 460–485.. https://doi.org/10.1145/3622814

Pierce, B.C., 2002. Types and Programming Languages, 1st ed. The MIT Press.

Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P., 2012. Task-Oriented Programming in a Pure Functional Language, in: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, PPDP '12. Association for Computing Machinery, New York, NY, USA, pp. 195–206.. https://doi.org/10.1145/2370776.2370801

PlatformIO, 2023a. Unity.

PlatformIO, 2023b. Unit Testing.

Plotkin, G., Pretnar, M., 2009. Handlers of Algebraic Effects, in: Castagna, G. (Ed.), Programming Languages and Systems, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 80–94.. https://doi.org/10.1007/978-3-642-00590-9_7

Popereshnyak, S., Suprun, O., Suprun, O., Wieckowski, T., 2018. IoT Application Testing Features Based on the Modelling Network, in: International Conference on Perspective Technologies and Methods in MEMS Design. pp. 127–131.. https://doi.org/10.1109/MEMSTECH.2018.8365717

Pothier, G., Tanter, É., Piquer, J., 2007. Scalable Omniscient Debugging. SIGPLAN Not. 42, 535–552.. https://doi.org/10.1145/1297105.1297067

Python Software Foundation, n.d.. Doctest — Test Interactive Python Examples.

Pötsch, A., Haslhofer, F., Springer, A., 2017. Advanced Remote Debugging of LoRa-enabled IoT Sensor Nodes, in: Proceedings of the Seventh International Conference on the Internet of Things, IoT '17. Association for Computing Machinery, New York, NY, USA, pp. 1–2.. https://doi.org/10.1145/3131542.3140259

Ramprasad, B., Fokaefs, M., Mukherjee, J., Litoiu, M., 2019. EMU-IoT-A Virtual Internet of Things Lab, in: Proceedings - 2019 IEEE International

Conference on Autonomic Computing, ICAC 2019. pp. 73–83.. https://doi.org/10.1109/ICAC.2019.00019

Rather, E.D., Moore, C.H., 1976. The FORTH Approach to Operating Systems, in: Proceedings of the 1976 Annual Conference, ACM '76. Association for Computing Machinery, New York, NY, USA, pp. 233–240.. https://doi.org/10.1145/800191.805586

Remote Debugging - LLDB, n.d.

Robot Framework Foundation, 2023. Robot Framework.

Rodriguez, M., Piattini, M., Ebert, C., 2019. Software Verification and Validation Technologies and Tools. IEEE Software 36, 13–24.. https://doi.org/10.1109/MS.2018.2883354

Rojas Castillo, C., Marra, M., Bauwens, J., Gonzalez Boix, E., 2021. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT Applications. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications.

Ronsse, M., De Bosschere, K., 1999. RecPlay: A Fully Integrated Practical Record/Replay System. ACM Transactions on Computer Systems 17, 133–152.. https://doi.org/10.1145/312203.312214

Rosenberg, J.B., 1996. How Debuggers Work: Algorithms, Data Structures, and Architecture. John Wiley & Sons, Inc., USA.

Roska, T., 1990. Limitations and Complexity of Digital Hardware Simulators Used for Large-Scale Analogue Circuit and System Dynamics. International Journal of Circuit Theory and Applications 18, 11–21.. https://doi.org/10.1002/cta.4490180104

Rospocher, M., Ghidini, C., Serafini, L., 2014. An Ontology for the Business Process Modelling Notation. Frontiers in Artificial Intelligence and Applications 267, 133–146.. https://doi.org/10.3233/978-1-61499-438-1-133

Rossberg, A., 2023. WebAssembly (Release 2.0).

Rossberg, A., 2019. WebAssembly (Release 1.0).

Rossum, G., 1995. Python Reference Manual. Amsterdam, The Netherlands, The Netherlands.

Saeedi, M., Markov, I.L., 2013. Synthesis and Optimization of Reversible Circuits—a Survey. ACM Comput. Surv. 45, 1–34.. https://doi.org/10.1145/2431211.2431220

Schmuck, F., Cristian, F., 1990. Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm, in: Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, PODC '90. Association for Computing Machinery, New York, NY, USA, pp. 133–143.. https://doi.org/10.1145/93385.93411

Schordan, M., Oppelstrup, T., Jefferson, D., Barnes, P.D., Quinlan, D., 2016. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application, in: Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS '16. Association for Computing Machinery, New York, NY, USA, pp. 111–122.. https://doi.org/10.1145/2901378.2901394

Schroeder, B., Pinheiro, E., Weber, W.-D., 2009. DRAM Errors in the Wild: A Large-Scale Field Study. SIGMETRICS Perform. Eval. Rev. 37, 193–204.. https://doi.org/10.1145/2492101.1555372

Schultz, U.P., 2020. Reversible Control of Robots. Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405.. https://doi.org/10.1007/978-3-030-47361-7_8

Schultz, U.P., Axelsen, H.B., 2016. Elements of a Reversible Object-Oriented Language, in: Devitt, S., Lanese, I. (Eds.), Reversible Computation. Springer International Publishing, Cham, pp. 153–159.. https://doi.org/10.1007/978-3-319-40578-0_10

Sen, K., Agha, G., 2006. Automated Systematic Testing of Open Distributed Programs, in: Baresi, L., Heckel, R. (Eds.), Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg, pp. 339–356.. https://doi.org/10.1007/11693017_25

Shapiro, M., Preguįca, N., Baquero, C., Zawirski, M., 2011. Conflict-Free Replicated Data Types, in: Défago, X., Petit, F., Villain, V. (Eds.), Stabilization, Safety, And Security of Distributed Systems. Springer, Berlin, Heidelberg, pp. 386–400.. https://doi.org/10.1007/978-3-642-24550-3_29

Shi, A., Lam, W., Oei, R., Xie, T., Marinov, D., 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019. Association for Computing Machinery, New York, NY, USA, pp. 545–555.. https://doi.org/10.1145/3338906.3338925

Shymanskyy, V., 2023. Wasm3/Wasm-Debug.

Skvařc Bŏzĭc, G., Irigoyen Ceberio, I., Mayer, A., 2024. In-Field Debugging of Automotive Microcontrollers for Highest System Availability, in: Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques, DEBT 2024. Association for Computing Machinery, New York, NY, USA, pp. 2–8.. https://doi.org/10.1145/3678720.3685314

Stallman, R., Pesch, R., Shebs, S., others, 1988. Debugging with GDB. Free Software Foundation 675.

Steegen, S., Tuerlinckx, F., Gelman, A., Vanpaemel, W., 2016. Increasing Transparency Through a Multiverse Analysis. Perspectives on Psychological Science 11, 702–712.. https://doi.org/10.1177/1745691616658637

Steinert, B., Cassou, D., Hirschfeld, R., 2012. CoExist: Overcoming Aversion to Change. SIGPLAN Not. 48, 107–118.. https://doi.org/10.1145/2480360.2384591

Stiévenart, Q., Binkley, D., De Roover, C., 2023. Dynamic Slicing of WebAssembly Binaries: 39th IEEE International Conference on Software Maintenance and Evolution. Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023) 84–96.. https://doi.org/10.1109/ICSME58846.2023.00020

Stiévenart, Q., Binkley, D.W., De Roover, C., 2022. Static Stack-Preserving Intra-Procedural Slicing of Webassembly Binaries, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22. Association for Computing Machinery, New York, NY, USA, pp. 2031–2042.. https://doi.org/10.1145/3510003.3510070

Strijbol, N., De Proft, R., Goethals, K., Mesuere, B., Dawyndt, P., Scholliers, C., 2024. Blink: An Educational Software Debugger for Scratch. SoftwareX 25, 101617.. https://doi.org/10.1016/j.softx.2023.101617

Symeonides, M., Georgiou, Z., Trihinas, D., Pallis, G., Dikaiakos, M., 2020. Fogify: A Fog Computing Emulation Framework, in: Proceedings - 2020 IEEE/ACM Symposium on Edge Computing, SEC 2020. pp. 42–54.. https://doi.org/10.1109/SEC50012.2020.00011

Söderby, K., De Feo, U., 2024. Debugging with the Arduino IDE 2.0.

Tan, S.-L., Anh, T.N.B., 2009. Real-Time Operating System (RTOS) for Small (16-Bit) Microcontroller, in: 2009 IEEE 13th International Symposium on Consumer Electronics. pp. 1007–1011.. https://doi.org/10.1109/ISCE.2009.5156833

Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., Welch, B., 1994. Session Guarantees for Weakly Consistent Replicated Data, in: Proceedings of 3rd International Conference on Parallel and Distributed

Information Systems. pp. 140–149.. https://doi.org/10.1109/PDIS.1994.331722

Tesone, P., Polito, G., Bouraqadi, N., Ducasse, S., Fabresse, L., 2018. Dynamic Software Update from Development to Production.. The Journal of Object Technology 17, 1.. https://doi.org/10.5381/jot.2018.17.1.a2

Tollervey, N.H., 2022. Code with Mu. a Simple Python Editor for Beginner Programmers.

Torres Lopez, C., Boix, E.G., Scholliers, C., Marr, S., Mössenböck, H., 2017. A Principled Approach towards Debugging Communicating Event-Loops, in: Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, And Decentralized Control, AGERE 2017. Association for Computing Machinery, New York, NY, USA, pp. 41–49.. https://doi.org/10.1145/3141834.3141839

Torres Lopez, C., Gurdeep Singh, R., Marr, S., Gonzalez Boix, E., Scholliers, C., 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper), in: DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2019.27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.. https://doi.org/10.4230/LIPIcs.ECOOP.2019.27

Ubayashi, N., Kamei, Y., Sato, R., 2019. When and Why Do Software Developers Face Uncertainty?, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). pp. 288–299.. https://doi.org/10.1109/QRS.2019.00045

Valmari, A., 1998. The State Explosion Problem. Lectures on Petri Nets I: Basic Models: Advances in Petri Nets, Lecture Notes in Computer Science.. https://doi.org/10.1007/3-540-65306-6_21

van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H., 2012. Memory Errors: The Past, the Present, and the Future, in: Balzarotti, D., Stolfo, S.J., Cova, M. (Eds.), Research in Attacks, Intrusions, And Defenses. Springer, Berlin, Heidelberg, pp. 86–106.. https://doi.org/10.1007/978-3-642-33338-5_5

VanderVoord, M., Karlesky, M., Williams, G., 2015. UNITY. Unit Testing for C (Especially Embedded Software).

Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T., 2007. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Transactions on Software Engineering 33, 856–868.. https://doi.org/10.1109/TSE.2007.70733

VanSickle, T., 2001. Programming Microcontrollers in C. Newnes.

Vishwakarma, G., Lee, W., 2018. Exploiting JTAG and Its Mitigation in IOT: A Survey. Future Internet 10, 121.. https://doi.org/10.3390/fi10120121

Waldo, J., Wyant, G., Wollrath, A., Kendall, S., 1997. A Note on Distributed Computing, in: Goos, G., Hartmanis, J., Leeuwen, J., Vitek, J., Tschudin, C. (Eds.), Mobile Object Systems Towards the Programmable Internet. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 49–64.. https://doi.org/10.1007/3-540-62852-5_6

Wasmer, Inc., 2022. Wasmer.

WebAssembly Community Group, 2022. WebAssembly Proposals.

Weiser, M., 1981. Program Slicing, in: Proceedings of the 5th International Conference on Software Engineering, ICSE '81. IEEE Press, San Diego, California, USA, pp. 439–449.

Weiss, A., Gautham, S., Jayakumar, A.V., Elks, C.R., Kuhn, D.R., Kacker, R.N., Preusser, T.B., 2021. Understanding and Fixing Complex Faults in Embedded Cyberphysical Systems. Computer 54, 49–60.. https://doi.org/10.1109/MC.2020.3029975

Weiss, G., 1975. Time-Reversibility of Linear Stochastic Processes. Journal of Applied Probability 12, 831–836.. https://doi.org/10.2307/3212735

Williams, G., 2014. Espruino.

Wills, M., 2022. The Bug in the Computer Bug Story.

Xie, H., 2017. Principles, Patterns, and Techniques for Designing and Implementing Practical Fluent Interfaces in Java, in: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, And Applications: Software for Humanity, SPLASH Companion 2017. Association for Computing Machinery, New York, NY, USA, pp. 45–47.. https://doi.org/10.1145/3135932.3135948

Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L., 2005. A Brief Survey of Program Slicing. SIGSOFT Softw. Eng. Notes 30, 1–36.. https://doi.org/10.1145/1050849.1050865

Yao, Y., Wang, Y., 2005. A Framework for Testing Distributed Software Components, in: Canadian Conference on Electrical and Computer Engineering. pp. 1566–1569.. https://doi.org/10.1109/CCECE.2005.1557280

Yĭgitler, H., Badihi, B., Jäntti, R., 2020. Overview of Time Synchronization for IoT Deployments: Clock Discipline Algorithms and Protocols. Sensors 20, 5928.. https://doi.org/10.3390/s20205928

Yokoyama, T., Axelsen, H.B., Glück, R., 2012. Towards a Reversible Functional Language, in: De Vos, A., Wille, R. (Eds.), Reversible Computation. Springer, Berlin, Heidelberg, pp. 14–29.. https://doi.org/10.1007/978-3-642-29517-1_2

Yokoyama, T., Axelsen, H.B., Glück, R., 2008. Principles of a Reversible Programming Language, in: Proceedings of the 5th Conference on Computing Frontiers, CF '08. Association for Computing Machinery, New York, NY, USA, pp. 43–54.. https://doi.org/10.1145/1366230.1366239

Yukihiro, M., others, 2023. Mruby. Mruby Is the Lightweight Implementation of the Ruby Language Complying with Part of the ISO Standard. Mruby Can Be Linked and Embedded within Your Application.

Yvon, S., Feeley, M., 2021. A Small Scheme VM, Compiler, and REPL in 4k, in: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2021. Association for Computing Machinery, New York, NY, USA, pp. 14–24.. https://doi.org/10.1145/3486606.3486783

Zelkowitz, M.V., 1973. Reversible Execution. Commun. ACM 16, 566.. https://doi.org/10.1145/362342.362360

Zerynth s.r.l., 2021. Zerynth Technical Reference v3.0.3.

Zhu, M.-Y., Wang, C.-W., 1992. Program Derivation in PowerEpsilon, in: 1992 Proceedings. The Sixteenth Annual International Computer Software and Applications Conference. IEEE Computer Society, pp. 206207208209210211–206207208209210211.. https://doi.org/10.1109/CMPSAC.1992.217567

Zhu, M.-Y., 2001a. Formal Specifications of Debuggers. SIGPLAN Not. 36, 54–63.. https://doi.org/10.1145/609769.609778

Zhu, M.-Y., 2001b. Denotational Semantics of Programming Languages and Compiler Generation in PowerEpsilon. SIGPLAN Not. 36, 39–53.. https://doi.org/10.1145/609769.609777

Zhu, M.-Y., Wang, \.C., 1991. A Higher-Order Lambda Calculus: PowerEpsilon.

Zolfaghari, B., Parizi, R.M., Srivastava, G., Hailemariam, Y., 2021. Root Causing, Detecting, and Fixing Flaky Tests: State of the Art and Future Roadmap. Software: Practice and Experience 51, 851–867.. https://doi.org/10.1002/spe.2929

B. Simply typed lambda calculus extensions

New syntactic forms

t ::=		*(terms)*
	...	
	true	*constant true*
	false	*constant false*
	if t then t else t	*conditional*
	0	*constant zero*
	succ t	*succ*
	iszero t	*iszero*
v ::=		*(values)*
	...	
	true	*true value*
	false	*false value*
	n	*numerical value*
n ::=		*(numeric values)*
	0	
	succ n	*constant true*
T ::=		*(types)*
	...	
	Bool	*booleans*
	Nat	*natural numbers*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{}{\text{if true then } t_1 \text{ else } t_2 \longrightarrow t_1} \quad \text{(E-IfTrue)}$$

$$\frac{}{\text{if false then } t_1 \text{ else } t_2 \longrightarrow t_2} \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\begin{array}{c}\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\ \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3\end{array}} \quad \text{(E-If)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \quad \text{(T-True)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \quad \text{(T-False)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T}} \quad \text{(T-If)}$$

$$\frac{}{\Gamma \vdash 0 : \text{Nat}} \quad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{succ } t_1 : \text{Nat}} \quad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{pred } t_1 : \text{Nat}} \quad \text{(T-Pred)}$$

$$\frac{\Gamma \vdash t_1 : \text{Nat}}{\Gamma \vdash \text{iszero } t_1 : \text{Bool}} \quad \text{(T-IsZero)}$$

Figure 2-16. Natural numbers and booleans for $\lambda^{\rightarrow}$. The syntax, evaluation, and typing rules for the natural numbers and booleans (Pierce, 2002).

New syntactic forms

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{}{\text{if true then } t_1 \text{ else } t_2 \longrightarrow t_1} \quad \textit{(E-IfTrue)}$$

$$\frac{}{\text{if false then } t_1 \text{ else } t_2 \longrightarrow t_2} \quad \textit{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\begin{array}{c}\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\ \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3\end{array}} \quad \textit{(E-If)}$$

Typing $\boxed{\Gamma \vdash t : T}$

Figure 2-17. Let bindings for $\lambda^{\rightarrow}$. The syntax, evaluation, and typing rules for let bindings (Pierce, 2002).

C. Full syntax and evaluation rules for the debugger

D. Progress and Preservation proofs

Here we include all relevant progress and preservation proofs for the debugger semantics from Chapter 2.

E. WebAssembly Specification Summary

In this appendix, we will discuss the elements of WebAssembly needed to understand the formalization of our extensions. A full and detailed account of all WebAssembly's formal semantics can be found in the excellent paper of Haas et al. (2017).

$$\begin{array}{rrcl} \emph{(value types)} & t & \Coloneqq & \textsf{i32} \; | \; \textsf{i64} \; | \; \textsf{f32} \; | \; \textsf{f64} \\ \emph{(packed types)} & tp & \Coloneqq & \textsf{i8} \; | \; \textsf{i16} \; | \; \textsf{i32} \\ \emph{(function types)} & \textit{tf} & \Coloneqq & t^* \rightarrow t^* \\ \emph{(global types)} & tg & \Coloneqq & mut^? \; t \\

\emph{(instructions) } & e & \Coloneqq & \key{unreachable} \; | \; \key{nop} \; | \; \key{drop} \; | \; \key{select} \; | \; \key{block} \; \textit{tf} \; e^* \key{end} \; | \\ & & & \key{loop} \; \textit{tf} \; e^* \key{end} \; | \; \key{if} \; \textit{tf} \; e^* \key{else} \; e^* \key{end} \; | \; \key{br} \; i \; | \; \key{br_if} \; i \; | \\ & & & \key{br_table} \; i^+ \; | \; \key{return} \; | \; \key{call} \; i \; | \; \key{call_indirect} \; \textit{tf} \; | \\ & & & \key{local.get} \; i \; | \; \key{local.set} \; i \; | \; \key{local.tee} \; i \; | \; \key{global.get} \; i \; | \\ & & & \key{global.set} \; i \; | \; t.\key{load} \; (\textit{tp_sx})^? a \, o \; | \; t.\key{store} \; \textit{tp}^? a \, o \; | \\ & & & \key{current_memory} \; | \; \key{grow_memory} \; | \; t.\key{const} \; c \; | \\ & & & t.unop_t \; | \; t.binop_t \; | \; t.testop_t \; | \; t.relop_t \; | \; t.cvtop \; t_sx^? \\ \emph{(functions)} & f & \Coloneqq & ex^* \; \key{func} \; \textit{tf} \; \key{local} \; t^* e^* \; | \; ex^* \; \key{func} \; \textit{tf} \, im \\ \emph{(globals)} & glob & \Coloneqq & ex^* \; \key{global} \; \textit{tg} \, e^* \; | \; ex^* \; \key{global} \; \textit{tg} \, im \\ \emph{(tables)} & tab & \Coloneqq & ex^* \; \key{table} \; n \, i^* \; | \; ex^* \; \key{table} \; n \, im \\ \emph{(memories)} & mem & \Coloneqq & ex^* \; \key{memory} \; n \; | \; ex^* \; \key{memory} \; n \, im \\ \emph{(imports)} & im & \Coloneqq & \key{import} \; \textit{"name" "name"} \\ \emph{(exports)} & ex & \Coloneqq & \key{export} \; \textit{"name"} \\ \emph{(modules)} & m & \Coloneqq & \key{module} \; \textit{tf}^* \; f^* \; glob^* \; tab^? \; mem^? \end{array}$$

E.1. Modules and Imports

A WebAssembly binary is organized as a *module*, which can only interact with its environment through typed imports and exports. WebAssembly embraces this strict encapsulation to better protect against possible security vulnerabilities associated with running WebAssembly bytecode in the browser. To further strengthen the security of WebAssembly, the language is defined with a strict type system that allows for fast static validation. As the last line of figure Figure 5-36 shows, a WebAssembly module contains *function types*, *functions*, *globals*, *tables* and at most one *memory*. All of these definitions, except *function types*, can be imported from other modules as well as exported under one or more names.

The life cycle of a WebAssembly module can be divided into three stages. In the first stage, the WebAssembly code is statically validated by the type system. This is usually done before compilation. After compilation the code can be executed by a WebAssembly runtime, such as WARDuino or a web browser. In the second stage, the runtime turns the module \$m = (\key{module}\,\textit{tf}^*\,f^\ast\,glob^\ast\,tab^?\,mem^?)\$ into a dynamic instance *inst*. As shown in figure Figure 5-38, all instances are kept in the WebAssembly global store *s*. During instantiation, the runtime must provide definitions for all imports, and it must allocate mutable memory. Finally, after this second stage, the WebAssembly code can be executed.

E.2. Control Flow

WebAssembly differs from traditional instructions sets in the way it prevents control flow hijacking (Burow et al., 2017, Lehmann et al., 2020). The instruction set does not allow arbitrary jumps, but only offers structured control flow. This means that unrestricted jumps do not exist. Instead, branches can only jump to the end of well-nested blocks inside the current function.

WebAssembly features three control constructs, , , and . These constructs all terminate with an instruction. The sequence of instructions e^* captured between these instructions form a so-called block. The can optionally hold two instruction sequences, separated by an extra opcode. When an instruction is executed, and the top of the stack is a non-zero number, the first block is executed, otherwise the second block is. Executing a executes the instructions captured in it unconditionally.

A branch instruction () specifies a target i. This target must be one of the blocks the instruction is executed in. If the target is a or , executing the branch will skip any instruction between the position of the and the of the targeted block. Branching to a jumps to the start of that . Counter-intuitively, the construct does not loop automatically. Instead, the instructions allow it to be repeated. Branches may also be conditional. A will only branch if the value on the top of the stack is non-zero. The , takes a list of targets and branches to the n-th target if the number n is on the top of the stack.[9]

[9]The instruction jumps to its last target if the index is out of bounds.

E.3. Functions

Modules contain a list of functions ($\key{func} \; \textit{tf} \; \key{local} \; t^* e^*$). All functions have a function type tf of the form $t_1^* \rightarrow t_2^*$. Because WebAssembly is a stack based language, this type describes the action of a function on the stack. The value types t_1^* before the arrow indicate the types of the elements the function expects to be on top of the stack when called. After the arrow, t_2^* indicates the return type.[10] The type $[\mathsf{i32}, \mathsf{i32}] \rightarrow [\mathsf{f32}]$ is used for a function that pops two 32-bit integers from the stack and pushes one 32-bit floating point number on the stack. Functions may also have local variables, these are declared as a list of value types after the function type as follows: $ \key{local} \; t^*$. These local variables are zero-initialized. The arguments and the local variables can be read or written via the $\key{local.get}$ and $ \key{local.set}$ instructions respectively. Both instructions take the index of the argument or local as argument. The body of a function e^* is a sequence of instructions that leaves the stack in a state matching the function's return type.

[10] Although a star is used, WebAssembly functions only return a single value.

E.4. Function Calls and Tables

Aside from arbitrary control flow, WebAssembly also lacks function pointers. It does provide an alternative with the instruction. This instruction can use a table to call functions based on an index operand calculated at runtime, similar to the instruction. Figure Figure 5-37 illustrates how this works. The instruction takes the value at the top of the stack and uses that to index a table of function references. Each table index corresponds to a function index, which in its turn points to a function. Tables can hold functions of different types, so the instruction takes a statically encoded argument specifying the type of the function it calls. At runtime, this encoded type is checked against the type of the function the index points to. If these do not correspond, the call is aborted, and a trap is thrown.

E.5. WebAssembly Linear Memory

The memory in WebAssembly is referred to as linear memory because it is a large array of bytes. Conceptually, the memory is divided into pages of 64 KiB. The size of memory is specified in terms of these pages, and linear memory can grow any number of pages at a time as long as the runtime can allocate the required space. Allocating additional pages can be done with the instruction. While the specification leaves the possibility of multiple memories open, WebAssembly still explicitly supports only one memory. However, a proposal for multiple memories is already in the implementation phase and so can be expected to be added to the standard in due course.

E.6. Execution

The execution of a WebAssembly program is described by a small-step reduction relation $\underset{i}{\hookrightarrow}$ over a configuration triple representing the state of the VM. A configuration contains one global store s, the local values v^* and the active instruction sequence e^* being executed. The rules are of the form $s; v^*; e^* \underset{i}{\hookrightarrow} s'; v'^*; e'^*$. In figure Figure 5-38, we present the most relevant small-step reduction rules for WARDuino.

At the top of the figure, we list all the relevant syntax for the rules. The store s consists of a set of module instances, table instances and memory instances. Tables and memories are only referenced by their index, since they can be shared between modules. A module instance consists of closures, global variables, tables and memories. A closure is the instantiated version of a function, and is represented by a tuple of the module instance and a code block. Values consist of constants. To elegantly represent the semantics a number of administrative operators are added to the list of instructions. The most important ones are **local** and **label**. The **local** operator indicates a call frame for function invocation (possibly over module boundaries), while the **label** operator marks the extent of a control construct.

In the lower part of figure Figure 5-38, we show some important small-step reduction rules for WebAssembly execution in WARDuino. Aside from the configuration $\{s, v^*, e^*\}$, the small step reduction rules operate on the currently executing instance. That is why the small-step reduction is indexed by the address i of that instance. The first two reduction rules govern the order of evaluation. The STEP-I rule splits a configuration into its context L^k and its focus and takes one step of the $\underset{i}{\hookrightarrow}$ relation. The second rule STEP-LOCAL explains how to evaluate a function that might reside in a different module. Note that this step changes the currently executing module, indicated by the two indices of the small-step relation $\underset{d,i}{\hookrightarrow}$. The last two rules are included because they are particularly relevant to our callback handling extension. The first rule, STEP-INDIRECT, transforms a instruction into a standard instruction. The STEP-INDIRECT rule takes a runtime index j, and an immediate function type tf. The index j must correspond to a function of the given type in the table of the current module $s_{tab}(i, j)$. If this is the case, the indirect call is replaced with a call to the function. On the other hand, when no correct function is found, the indirect call is replaced by a as shown by STEP-INDIRECT-TRAP. This means the program will stop executing. When all goes well, the resulting call can be reduced further. We omit any further reduction rules from the WebAssembly standard, because they are not changed or not relevant to the further discussion in this section. The interested reader can find all WebAssembly reduction rules in the original WebAssembly article(Haas et al., 2017).

Now we have all the formal tools required to describe the extensions to WebAssembly implemented in WARDuino. We will discuss each extension in turn in the following sections.

F. Built-in Modules

This appendix gives an overview of the WARDuino VM's built-in modules, which provide primitives for controlling peripheral hardware and other essential aspects of IoT applications. The examples are written in WebAssembly's textual format.

F.1. Input-Output Pins

A first module exposes the hardware pins of the microcontroller. In a microcontroller each pin is connected through a so-called port. A port controls the properties of a pin, such as its mode. The mode of a pin determines if it can be used for reading or writing, it is important to make sure the mode of the pin is always set correctly.

Arduino abstracts away the division between ports and pins through a simple API. This API allows us to set the pin mode, read the pin or write a value to it. In WARDuino we defined a native implementation of these functions in an IO module. The signatures of the functions in our IO module are listed on the right side of figure Listing 6-48. The first function, `pin_mode` returns no values, but takes two `i32`[11] arguments; the first argument identifies the pin and the second the mode, either input, output, or input_pullup. The second function `digital_write` has no return value and takes two arguments. Again, the first argument specifies the pin, and this time the second argument provides the value to be written to the digital pin, either high or low. Finally, the `digital_read` function takes a digital pin as argument, and returns the value read from the specified pin, either high or low, as an `i32` value.

[11]32 bit integer

```
 1 (module
 2 (; type declarations ;)
 3 (type $int->int->vd (func (param i32) (param i32) (result)))
 4 (type $int->vd      (func (param i32) (result)))
 5 (type $vd->vd       (func (param) (result)))
 6 (; imports ;)
 7 (import "env" "pin_mode"      (func $pin_mode  (type $int->int->vd)))
 8 (import "env" "digital_write" (func $dig_write (type $int->int->vd)))
 9 (import "env" "delay"         (func $delay (type $int->vd)))
10 ; export $blink as the main entry point of the program ;)
11 (export "main" (func $blink))
12 (; blink function ;)
13 (func $blink (type $vd->vd)
14   (call $pin_mode (i32.const 16) (i32.const 1)) (; write mode ;)
15   (loop $begin
16     (call $dig_write (i32.const 16) (i32.const 0)) (; off ;)
17     (call $delay (i32.const 5000)) (; sleep 5s ;)
18     (call $dig_write (i32.const 16) (i32.const 1)) (; on ;)
19     (call $delay (i32.const 5000)) (; sleep 5s ;)
20     (br $begin)))) (; jump back to start of $begin loop ;)
```

pinMode(pin,mode)	int × int → ()
digitalWrite(pin, value)	int × int → ()
digitalRead(pin)	int → int

Listing 6-48. API and example of the WARDuino digital input-output.

While WebAssembly is primarily a bytecode format, it can be represented in a human-readable text format. The leftside of figure Listing 6-48 shows a WARDuino program that blinks an LED light in the WebAssembly text format (WAT). The program is defined as a module with four major sections. The first part, lines Listing 6-483 to Listing 6-485, declares the types used throughout the program. WebAssembly byte code uses numerical indices to refer to other entities, such as function arguments, local variables, functions and so on. In the text format we can assign these entities a human-readable name prefixed with a dollar sign. For instance the first type gets the name \$int->int->vd on Listing 6-483. The name of the type is followed by the description of the type, which starts with the `func` keyword signifying it is a function type. Next, all the parameters are listed, each with a separate `param` keyword followed by the parameter type. A parameter can only have a basic numeric type. For the \$int->int->vd, there are two 32-bit integer parameters (`i32`). A function which takes no arguments (\$vd->vd), has only one `param` keyword without a type, as is the case on Listing 6-485. After the parameters, the base numeric type of the return value is specified in a similar way. The `result` keyword is followed by a basic numeric type, or by no type to indicate void (abbreviated `vd`), as is the case for all the types in the example.

The following section of the program imports the used WARDuino primitives. Specifically, on lines Listing 6-487 to Listing 6-489, the `pin_mode`, `digital_write`, and `delay` primitives are imported from the `env` module. Each import statement starts with the module name and the name of the entity within that module to be imported. This is followed by a declaration of the imported entity, which can be either a table, linear memory, a function, or a global variable. In this example, we only import functions. Thus, each description consists of the `func` keyword followed by an identifier referring to the type of the imported function.

Next, the program exports the function `\$blink` under the name “main” on Listing 6-4811. When executing a WebAssembly module with WARDuino, our VM will automatically look for an exported function with the name main”‘. It considers this function the entry point of the program, and will automatically start executing it.

The `$blink` function is defined on lines Listing 6-4813 to Listing 6-4820, it starts by setting the mode of the LED pin[12] to 1 (output). This is done by calling the `pin_mode` primitive imported under the name `$pin_mode`. WebAssembly allows two different function call syntaxes, folded”‘ and unfolded”‘. In this case we used the folded syntax”‘ to make the call. We placed the arguments to `$pin_mode` in the brackets of the `call` instead of placing these arguments on the stack first as one would usually do in a stack based language[13]. The two notations are equivalent and we will use them interchangeably. After this, the function continuously blinks the LED every 10 seconds in an infinite loop, starting at Listing 6-4815. In WebAssembly a `loop` construct has an identifier, in this case `$begin` and a body, everything after the identifier. Contrary to other languages loops do not automatically repeat, we must explicitly jump (branch) back to their start. This is done at Listing 6-4820 with (`br $begin`). Note that we can only branch to identifiers of blocks we are in. If we do not branch back to the start of a `loop`, its body is executed only once.

The first instruction in the loop writes zero to the LED pin, turning the LED off. Next, the `i32.const` instruction places the value 5000 on the stack. The call to our `$delay` primitive on the next line consumes this value and

[12]In the example, we assume the LED is attached to pin 16.

[13]The unfolded form is: (`i32.const 16`) (`i32.const 1`) (`call $pin_mode`)

waits that number of milliseconds before returning. After the microcontroller has waited for five seconds (5000 ms), it turns the LED back on with the `$dig_write` primitive. Then it waits for another five seconds before starting again at the top of the loop (Listing 6-4815).

For brevity, we will leave out the type declarations in future examples and indicate types by using a corresponding name, such as `$int->int->vd`. Additionally, we will also omit the export of the main function, instead we assign the entry point the identifier `$main`.

F.2. Pulse Width Modulation and Analog Reads

A pulse width modulator (PWM) allows programmers to send out a square wave to one of the output pins without having to write a busy loop. Example waves are shown in figure Listing 6-49 with duty cycles of 90%, 50% and 20%. The duty cycle is the configurable fraction of time the wave is high. PWM is prototypically used to dim an LED, sending it a square wave makes it flash very fast, faster than perceivable by the eye. The higher the duty cycle, the brighter the LED appears.

To control the modulator we provide three API functions: `setPinFrequency`, `analogWrite`, and `analogRead`. The interface for each of these functions is shown in figure Listing 6-49. With `setPinFrequency` we can modify the frequency of a certain pin. For example when the default frequency on pin `D1` is 31250 Hz a call to (`setPinFrequency D1 8`) will change the frequency on the pin to 31250/8 Hz. Setting the duty cycle is done with `analogWrite`, an argument value of 0 corresponds to a duty cycle of 0%, the value 255 represents a duty cycle of 100%. Finally, the `analogRead` function measures the voltage on a certain pin and returns it as an integral value.

```
 1 (module
 2   (; fade function ;)
 3   (func $main (type $vd->vd)
 4     (local $i i32) (; loop iterator ;)
 5     (call $pin_mode (i32.const 16) (i32.const 1))
 6     (loop $infinite
 7       (local.set $i (i32.const 0))
 8       (loop $increment
 9         (call $analog_write (i32.const 16) (local.get $i))
10         (local.set $i (i32.add (local.get $i) (i32.const 1)))
11         (i32.const 5)
12         (call $delay)
13         (br_if $increment       (; jump to line 8 if i<255 ;)
14                (i32.lt_s (get_local $i) (i32.const 255))))
15       (loop $decrement
16         (call $analog_write (i32.const 16) (local.get $i))
17         (local.set $i (i32.sub (local.get $i) (i32.const 1)))
18         (i32.const 5)
19         (call $delay)
20         (br_if $decrement        (; jump to line 15 if i>0 ;)
21                (i32.gt_s (local.get $i) (i32.const 0))))
22       (br $infinite))))
```

Listing 6-49. *Left*: Example of the PWM module in WARDuino. *Top right*: PWM API of WARDuino. *Bottom right*: Graphs of output voltages of over time for duty cycles set to 90%, 50% and 20%, the average output voltage is shown as a dashed line.

The left side of figure Listing 6-49 shows how we can use the PWM primitives to add a slow fade effect to a blinking led. Analogous to the code example for letting an LED light blink, the WebAssembly module contains a main function that takes no arguments and returns no values, but does contain one local variable `\$i`. Local variables in WebAssembly are always defined at the start of the function alongside the arguments and return values. The main function first sets the pin mode of the LED pin to output. After the correct mode is set, the code lets the LED fade on and off continuously in an infinite loop, starting on Listing 6-496. The body of our outer loop first initializes the variable `\$i` to zero on Listing 6-497. Then, a first inner loop increments the brightness of the LED from 0 to the maximum value 255 in steps of one, by writing the value of `\$i` to the pin with the `analog_write` primitive. Each iteration of the loop waits five milliseconds before continuing. At the end of this inner loop, the `br_if` instruction jumps back to the start of the loop if the loop iterator `\$i` is less than 255. After the first loop when `\$i` equals 255, a second inner loop decrements the brightness of the LED light in the same way. Once `\$i` hits zero, we have reached the endremote the cycle and the unconditional branch instruction (`br \$infinite`), at Listing 6-4922, jumps back to the start of the main loop.

F.3. Serial Peripheral Interface

The serial peripheral interface (SPI) is a bus protocol commonly used to communicate between a microcontroller and peripheral devices such as sensors, SD-cards, displays, and shift registers. The SPI communication protocol can be implemented in hardware or in software. When using the hardware implementation the programmer must use the dedicated SPI pins on the microcontroller. In software, the programmer is free to use any of the available input-output pins. Software implementations are however, significantly slower than making use of the hardware implementation.

spiBegin()	$() \to ()$
spiBitOrder(bitorder)	$\text{int} \to ()$
spiClockDivider(divider)	$\text{int} \to ()$
spiDataMode(mode)	$\text{int} \to ()$
spiTransfer8(data)	$\text{int} \to ()$
spiTransfer16(data)	$\text{int} \to ()$
spiBulkTransfer8(count,data)	$\text{int} \times \text{int} \to ()$
spiBulkTransfer16(count,data)	$\text{int} \times \text{int} \to ()$
spiEnd()	$() \to ()$

Table 6-18. API of the WARDuino SPI module

WARDuino's primitives governing access to the hardware SPI bus are shown in figure Table 6-18. The functions `spiClockDivider`, `spi\-Bit\-Order`, `spiDataMode` are configuration functions to specify how data will be transferred. Before actually using the SPI bus the programmer first needs to call the `spiBegin` which initializes the SPI module. Once initialised, the programmer can start transferring data to the peripheral device by using one of the transfer functions. We included two kinds of transfer functions one for 8-bit transfers and one for 16-bit transfers. For both variants we included a bulk mode which sends the same data a specific number of times. The inclusion of the bulk operations can improve the performance of a display driver greatly.

We have used the SPI module to implement a display driver in WARDuino. We leave out the specifics of that implementation here, not only for brevity, but because the code is originally written in C, rather than directly in WebAssembly like our other examples. We refer any interested reader to the first paper on WARDuino cite{gurdeep-singh19}.

F.4. Serial Port Communication

```
1 (module
2   (memory $text 1)          (;
    Initialize linear memory to one page ;)
3   (data (i32.const 0) "WARDuino") (;
    place text in memory at offset 0;)
4
5   (func $main (type $vd->vd)
6     (i32.const 0) (; start index of
      string ;)
7     (i32.const 8) (; string length ;)
8     (call $print)))
```

print(string)	int × int$^{\{string\}}$	→ ()
print_int(value)	i32	→ ()

Listing 6-50. API and example code of the Serial module in WARDuino.

Microcontrollers typically have at least one serial port. This port is used for flashing code to a microcontroller. Developers also regularly use this port for printing debug or log messages to a computer during development. The Arduino's `Serial` library is therefore indispensable for many programmers. We use it to add two print primitives to WARDuino to print numeric values and strings to the serial port. That latter feature is not as straightforward as it may seem because WebAssembly only supports basic numeric types, and not strings.

Fortunately, we can represent strings in WebAssembly by storing them as UTF-8 encoded bytes in WebAssembly's linear memory. Memory in WebAssembly is called linear memory because it is simply one long continuous buffer that can grow in increments of 64 kiB pages. Currently, WebAssembly only supports one memory per module, but memories are importable. Saving strings in memory is not enough, we also need a way to work with them, specifically, we need a way of referring to a string. To pass a string as an argument to a function, it can be represented as a tuple containing its offset in WebAssembly memory together with its length. This is illustrated in figure Listing 6-50, which shows the interface of our two serial bus primitives. One primitive simply prints a numeric value, the other prints a string from linear memory. The example program on the left side of the figure shows how we can print a string to the serial port in WebAssembly. The code starts on line 2 by declaring a WebAssembly linear memory with the label \\`$text`, followed by an initial size of one memory page (64 kiB). This is more than enough space to store the simple message in the data section on the next line. This section is similar to the data sections found in native executable files. The string is written at offset 0 in linear memory at initialization time. Not much more is needed to print the text in memory to the serial port, the main function simply places the indices and length of the string on the stack and calls the print primitive.

F.5. Wireless Networks

Applications for embedded devices often communicate with other devices. To accommodate this, many microcontrollers come with a Wi-Fi chip to connect to a wireless network. We have extended WARDuino with the necessary primitives for connecting to a wireless network. Because we use Arduino to implement these primitives in WARDuino, it makes sense to mirror the underlying Arduino interfaces for connecting. This way we do not unnecessarily introduce entirely new interfaces. Unsurprisingly, the Arduino functions use strings to specify parameters such as the network SSID and password. We represent those strings as pairs of integers as discussed in the section on the serial port communication module.

```
(module
  (; memory ;)
  (memory $credentials 1)
  (data (i32.const 0) "SSID")
  (data (i32.const 6) "P4S5W0RD")

  (; connect function ;)
  (func $main (type $vd->vd)
    (loop $until_connected
      (i32.const 0) (; ssid start address ;)
      (i32.const 4) (; ssid string length ;)
      (i32.const 6) (; password start address ;)
      (i32.const 8) (; password string length ;)
      (call $connect)
      (i32.ne (call $status) (i32.const 3))  (; true if failed ;)
      (br_if $until_connected))
    (i32.const 10)  (; arg1 of print: buffer offset,      ---  ;)
    (i32.const 10)  (; arg1 of localip: buffer offset,        | ;)
    (i32.const 20)  (; arg2 of localip: buffer length         | ;)
    (call $localip) (; return value becomes arg2 of print ---  ;)
    (call $print)
  ))
```

connect	$\text{ssid}_{\{start\}}, \text{ssid}_{\{length\}}$ $\text{pass}_{\{start\}}, \text{pass}_{\{length\}}$	$\text{int}^4 \rightarrow ()$
status()		$() \rightarrow \text{int}$
localip($\text{ip}_{\{start\}}$, $\text{ip}_{\{max\_length\}}$)		$\text{int}^2 \rightarrow \text{int}$

Listing 6-51. API and example code of the Wi-Fi module in WARDuino. $\text{int}^2 = \text{int} \times \text{int}$

Figure Listing 6-51 shows the interfaces of the wireless networking primitives on the right. Because these primitives take strings as arguments, the number of integer parameters can get relatively high. To keep the description of the API compact, we abbreviate long chains of the same type with the power notation. For instance, the `connect` primitive that connects to a Wi-Fi network has type $\text{int}^4 \rightarrow ()$. This notation represents four integer arguments, or two strings in this case, and no return value. The first string argument contains the SSID of the network to connect to, the second argument contains the password used to authenticate. The `status` primitive returns an integer indicating the status of the network connection. If there is an active connection it will return 3. Our `localip` primitive retrieves the IP address of the device. This primitive takes two integer arguments representing a memory slice where a string can be stored. Because WebAssembly only supports one memory per module, the returned string needs to be saved in the memory defined by the module calling `localip`. To know where in this memory the primitive can safely write its string return value, we require a memory slice as argument. Once the IP address is written to the memory slice, `localip` returns the size of string it has written. This methodology is comparable to how C functions take a character buffer as an argument to write their result to.

A small piece of WebAssembly code that connects to a Wi-Fi network and prints the IP address is shown on the left of figure Listing 6-51. The code first declares a memory of one page (64 kiB) and writes the network SSID and password to it (lines 3-5). The main function starts by connecting to the Wi-Fi network in the `\$until_connected` loop (lines 9-16). At the start of the loop, `const`-instructions place the offsets and lengths of the two strings on the stack. Then we call the `connect` primitive, which tries to connect to the given network. The call blocks execution until it finishes or fails. We check whether a connection was successfully established by verifying that the `status` primitive returns 3 (connected). If not, the `br_if` instruction on line 16 jumps back to the start of the loop, and the program retries connecting to the network. Once connected, we print the IP address of the device by combining `localip` and `print`. The `localip` primitive returns the length of the string it wrote to the memory slice it received as argument, zero indicates a failure to retrieve the local IP address. Because WebAssembly is a stack based language, we can push the start index of the response buffer of `localip` to the stack before pushing the arguments to `localip`. When `localip` returns, it will have popped its two arguments off the stack and pushed the length of the IP address back to the stack. Now, the stack holds the right arguments for the `print` primitive once execution gets to line 21. If the print primitive gets a zero length argument it will simply not print anything, so we do not need to check in WebAssembly whether an IP address was actually retrieved.

F.6. Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) cite{fielding14} drives the modern web. Developers can use HTTP to access the entire web from a WebAssembly program running on a microcontroller with WARDuino. To keep the module small, we only add the most fundamental HTTP requests, GET, PUT, POST. % get, put, post. As before, we give the interface of the primitives in figure Listing 6-52. String arguments are given as pairs of integers representing memory slices. If the primitive returns a string, an extra pair of integers, pointing to a free slice of memory, is added to the arguments.

```
1  (module
2    (; Memory ;)
3    (memory $url 1)
4    (data (i32.const 0)
5          "http://www.arduino.cc/asciilogo.txt")
6
7    (func $main (type $vd->vd)
8      (loop $loop
9        (i32.const 40)  (; response_start for print ;)
10       (i32.const 0)   (; url_start ;)
11       (i32.const 35)  (; url_length ;)
12       (i32.const 40)  (; response_start ;)
13       (i32.const 200) (; response_length ;)
14       (call $get)
15       (call $print)
16       (i32.const 1000)
17       (call $delay)
18       (br $loop))))
```

get($\text{url}_{\{\text{start}\}}, \text{url}_{\{\text{length}\}}$ $\quad \text{int}^4 \rightarrow \text{int}$
$\text{response}_{\{\text{start}\}}, \text{response}_{\{\text{length}\}}$)

put($\text{url}_{\{\text{start}\}}, \text{url}_{\{\text{length}\}}$ $\quad \text{int}^6 \rightarrow \text{int}$
$\text{payload}_{\{\text{start}\}}, \text{payload}_{\{\text{length}\}}$)
$\text{content type}_{\{\text{start}\}}, \text{content type}_{\{\text{length}\}}$)

post($\text{url}_{\{\text{start}\}}, \text{url}_{\{\text{length}\}}$ $\quad \text{int}^8 \rightarrow \text{int}$
$\text{payload}_{\{\text{start}\}}, \text{payload}_{\{\text{length}\}}$)
$\text{content type}_{\{\text{start}\}}, \text{content type}_{\{\text{length}\}}$)
$\text{response}_{\{\text{start}\}}, \text{response}_{\{\text{length}\}}$)

Listing 6-52. API and example code of the HTTP module in WARDuino.

The code example in figure Listing 6-52 prints an ASCII version of the Arduino logo retrieved from the internet with an HTTP GET request. To do this, it first adds the URL of the ASCII art logo in WebAssembly linear memory (lines 3-5). The main function will repeatedly retrieve the logo in an infinite loop that starts on line 8. Before the code pushes the four integers arguments for the `get` primitive, it pushes the start index of the response buffer onto the stack. This is the same trick we used in the previous example using the `print` primitive. By pushing this value now, and the `get` primitive pushing the length of the result, we can call the `print` primitive immediately without having to reorder the stack first. After the ASCII text has been printed to the serial port, the microcontroller waits for 1 second before starting the entire procedure again.

F.7. MQTT Protocol

HTTP was designed for the web and is not optimized for an embedded context cite{naik17}. More suitable protocols have been developed for IoT applications, such as the widely used MQTT cite{banks14} protocol. This is one of the most mature and widespread IoT protocols at the time of writing. It is more lightweight in several aspects compared to HTTP. The message overhead is a lot smaller, since headers only require 2 bytes per message. Another important difference with the client-server approach of HTTP, is the client-broker architecture of MQTT. By using a publish-subscribe paradigm, MQTT reduces the number of messages a microcontroller needs to send. The publish-subscribe paradigm is commonly used in IoT contexts because its simplicity and effectiveness at reducing network traffic cite{gupta21,sidna20}. The main idea of this paradigm is to disconnect communication in time and space. This means, that entities do not have to be reachable at the same time, and do not need to know each other, to communicate. Consequently, entities are free to halt execution or sleep. They may send and process messages whenever they choose to. This is the great advantage of MQTT over HTTP for constrained devices. We have added the basic MQTT operations to WARDuino. The implementation is backed by Nick O'Leary's[14] Arduino library for MQTT messaging.

Because MQTT clients do not know each other, they communicate through a shared third party, the MQTT Broker. Communication starts when an MQTT client opens a persistent TCP connection with the MQTT Broker and sends an arbitrary string as its unique identifier to the server. Once connected, the MQTT client can both publish messages or subscribe to topics. The broker filters incoming (published) messages based on their topics and sends them asynchronously to every connected client subscribed to those specific topics. Topics need not be initialized, clients can send messages to any topic string of the right form. \

[14]Documentation at: https://github.com/knolleary/pubsubclient

```
(module
  (memory $url 1)
  (data (i32.const 0)
        "broker.hivemq.com")
  (data (i32.const 20) "mcu")
  (data (i32.const 25) "helloworld")

  (; callback function ;)
  (func $callback (type $int->int->int->int->vd)
    (call $print (local.get 2) (local.get 3)))
  (; add callback to callbacks table ;)
  (table $callbacks 1 funcref)
  (elem (i32.const 0) $callback) (; fidx = 0 ;)

  (; (re)connect function ;)
  (func $reconnect (type $vd->vd)
    (call $poll)
    (loop $until_connected
      (; connect to MQTT ;)
      (i32.const 20)   (; client id start ;)
      (i32.const 3)    (; client id length ;)
      (call $connect)
      (i32.ne (call $connected) (i32.const 1))
      (br_if $until_connected)))

  (func $main (type $vd->vd)
    (i32.const 0)    (; url start ;)
    (i32.const 17)   (; url length ;)
    (i32.const 1883) (; port ;)
    (call $init)
    (call $reconnect)

    (loop $try_subscribing
      (i32.const 25) (; topic start ;)
      (i32.const 10) (; topic length ;)
      (i32.const 0)  (; fidx ;)
      (call $subscribe)
      (i32.const 1)
      (br_if $try_subscribing (i32.ne)))

    (loop $waitloop
      (call $delay (i32.const 1000))
      (call $reconnect)
      (br $waitloop))))
```

init(	$\text{server}_{\{start\}}, \text{server}_{\{length\}}$, port)	$\text{int}^3 \rightarrow ()$
connect(	$\text{id}_{\{start\}}, \text{id}_{\{length\}}$)	$\text{int}^2 \rightarrow \text{int}$
poll()		$() \rightarrow \text{int}$
connected()		$() \rightarrow \text{int}$
subscribe(	$\text{topic}_{\{start\}}, \text{topic}_{\{length\}}$, fidx)	$\text{int}^3 \rightarrow \text{int}$
unsubscribe(	$\text{topic}_{\{start\}}, \text{topic}_{\{length\}}$, fidx)	$\text{int}^3 \rightarrow \text{int}$
publish(	$\text{topic}_{\{start\}}, \text{topic}_{\{length\}}$, $\text{payload}_{\{start\}}, \text{payload}_{\{length\}}$)	$\text{int}^4 \rightarrow \text{int}$

Signature of MQTT callback functions:

fn_name($\text{topic}_{\{start\}}, \text{topic}_{\{length\}}$ $\text{payload}_{\{start\}}, \text{payload}_{\{length\}}$)	$\text{int}^4 \rightarrow ()$

Listing 6-53. API and example code of the MQTT module in WARDuino.

The first four MQTT primitives shown on the right side of figure Listing 6-53, are administrative. The `init` function sets the URL and port of the MQTT broker. By calling the `connect` primitive with a client ID string, represented by a memory slice, a connection is established. This primitive returns the status of the connection with the server (one if connected, else zero). We give developers full control over the frequency with which the device checks for new messages. They can trigger a check by calling our `poll` primitive without arguments. Such a call will process all incoming messages and invoke their callbacks, the return value is the status of the connection. The `poll` primitive needs to be called regularly to maintain the connection to the broker. Getting the connection status can also be done without processing messages by using the `connected` primitive.

The remaining primitives encompass the core MQTT operations: `subscribe`, `unsubscribe`, and `publish`. They all return a boolean value to indicate success (1) or failure (0). Our `subscribe` primitive takes a topic string, and the function index of a callback function that will handle any incoming message matching the specified topic. A callback function must be of the type $\text{int}^4 \to ()$. It takes two strings as argument: the topic and the payload of the received message. The callback function can interact with the memory of the module but must not return a value. To assign a function index to a function, it must be stored in a `table` of the WebAssembly module. The function index is simply its index in the callback's table. Whenever a message arrives from the server for a subscribed topic, the appropriate callback functions will be executed by WARDuino. Our `unsubscribe` primitive permits removing specific callback functions from specific topics. If all callbacks to a topic are removed, the MQTT broker is informed that we no longer wish to get messages for that topic. Aside from subscribing to topics, we can also send payloads for topics to the MQTT Broker. This is done with the `publish` primitive that takes the same arguments as a callback function: a topic string, and a payload of the message to be published.

Figure Listing 6-53 shows an example MQTT program on the left, which subscribes to the `helloworld` topic of an MQTT broker. Our small WebAssembly program will print the payload of each message it receives. The code starts by declaring all the static strings used in the program (lines 2-6). Our entry point is the main function defined on lines 26 to 44. First we initialize the MQTT module with the URL and port of the broker using the `init` primitive (27-30). Note that we have omitted the Wi-Fi connection code for brevity, as we have already shown how to connect to a Wi-Fi network in figure Listing 6-51. Once our module is initialized, we connect it to the MQTT broker by using the \\`$reconnect` function. This function is defined on lines 16 to 24. It calls the administrative `poll` primitive and tries to connect to the broker until successful. After calling \\`$reconnect`, our main function continues by subscribing to the `helloworld` topic (lines 33-39). This is done in a loop labeled `\$try_subscribing` which calls the `subscribe` primitive repeatedly until it returns 1 (success). In WebAssembly we cannot pass a function directly to another function. Instead, we must add the function to a table of function references. The code declares such a table of size one on line 12. On the next line the element section adds our callback function to the `\$callbacks` table at index zero. This is the zero we use on line 36 to refer to it. Lines 9 to 10 define the callback function we stored in our table. It takes two arguments, a message topic and a payload. With the `local.get` instruction, the function places its last two arguments, corresponding to the payload string, on the stack and then it calls the `print` primitive. Our main function ends with an infinite loop on lines 41 to 44 that calls `\$reconnect` every second to check if the connection is still live and reconnect if necessary.

G. Over-the-air Updates Defined Orthogonally

The formalization of the over-the-air updates is presented in Section 3.8.4 as an addition to the remote debugger semantics (Section 3.8.2). However, we designed the update semantics to be orthogonal to the debugging system, making it easy to define a version of the over-the-air updates that does not rely on the remote debugger. The rules below show the update system as a standalone semantics on top of the WebAssembly semantics.

All parts of the debugger semantics are removed, and a new vm-run rule is introduced. Contrary to the semantics shown in Section 3.8.2, the state s is now only extended with the incoming messages.

G.1. Microbenchmarks

Our microbenchmarks are implemented as described below.

tak a popular function from the Gabriel Benchmarks, contains an implementation of Takeuchi's tak function, specifically $\tau(18, 12, 6)$.

catalan computes various Catalan numbers. The n-th Catalan number is commuted with: $C_n = \frac{1}{n+1}\binom{2n}{n}$. The benchmark implements this formula up to the 17th Catalan number (to avoid overflow).

fac implements a recursive implementation for calculating the integer factorial function. In the benchmark we calculate $(n \bmod 12)!$ for $n \in [0, 1000[$.

fib determines the value of the n^{th} Fibonacci number iteratively, for all $n \in [1000, 1050[$.

gcd computes the greatest common denominator of two numbers. In the benchmark we calculate the gcd of all whole numbers in $[4000, 5000[$ and 12454.

primes verifies if numbers are prime by looking for divisors. The benchmark consists of finding and calculating the sum of the first 127 primes.

G.2. Auxiliary out-of-place debugger rules

G.3. Auxiliary multiverse debugger rules

In this appendix, we present the auxiliary debugger rules for the multiverse debugger for WebAssembly, omitted from cref{sec:multiverse-debugger} in the main text for brevity. These are the rules for the step forward operations on primitive calls, and the run variant of the textsc{step-mock} rule.

G.4. Proofs and auxiliary lemmas for the multiverse debugger

In this appendix, we present the lemmas and proofs for the multiverse debugger semantics for WebAssembly, omitted from cref{sec:correctness}. The first lemma states that the mocking of input values will not introduce states in the multiverse debugger that cannot be observed by the underlying language semantics. Since the input values accepted by the textsc{register-mock} rule must be part of the codomain of the primitive, this will always be the case.

A second lemma crucial to the soundness of the debugger, states that for any debugging state, there is a path in the underlying language semantics from the start to every snapshot in the snapshot list.

Now we give the proof for debugger soundness, where the snapshot soundness lemma will be crucial. % followed by the auxiliary lemmas for snapshot soundness (cref{lemma:snapshot-soundness}), checkpoint existence (cref{lemma:checkpoint-existence}), and deterministic path (cref{lemma:deterministic-path}).