Voorblad van Plato downloaden

UNIVERSITEIT
GENT

Dankwoord

Waarom de keuze voor I2C? Hier wordt er gepraat over jezelf, bvb. wat ik persoonlijk bijgeleerd heb.

Permission for Usage

The author gives permission to make this master's thesis available for consultation and to copy parts of this master's thesis for personal use. Every other use is subject to copyright terms, in particular with regard to the obligation to explicitly state the source when quoting results from this master's thesis.

Abstract - Dutch

Abstract - English

Contents

List of Figures

List of Tables

List of Acronyms

ABI application binary interface. , 9

ACK Acknowledgement. , 11, 12

API Application Programming Interface. , 3, 4, 8

GPIO general-purpose input/output. , 14

HAL Hardware Abstraction Layer. , 14

HAT Hardware Attached on Top. , 19

I2C Inter-Integrated Circuit. , 1, 8, 11–14, 16, 17, 19

IDL Interface Description Language. , 5, 6

IoT Internet of Things. , 7

MCU Microcontroller Unit. , 11, 15, 19

NACK Negative-acknowledgement. , 11

OS Operating System. , 3, 9, 14, 15

SCL Serial Clock Line. , 11–13

SDA Serial Data Line. , 11, 12

SIG Special Interest Group. , 4

SMBus System Management Bus. , 12, 13

SPI Serial Peripheral Interface. , 14

SVD System View Description. , 14

SWD Serial Wire Debug. , 15

UART universal asynchronous receiver-transmitter. , 14

VM Virtual Machine. , 3, 5, 7

WAMR WebAssembly Micro Runtime. , 7, 27, 28

WASI WebAssembly System Interface. , 1, 3–7, 9

Wasm WebAssembly. , 1, 3, 4, 6, 7, 9, 10, 21–23, 28

WIT Wasm Interface Type. , xii, 4–6, 9, 16, 19–21, 23

List of Code Fragments

1

Introduction

There are countless software solutions running on critical infrastructure, where sudden failure could result in the loss of people's live, e.g. programs guiding surgeons during operations, infotainment systems inside cars or train control systems. Due to a variety of reasons, the longevity of this hardware often spans multiple decades [1]. Furthermore, in the automotive industry, there's a clear trend towards more advanced infotainment systems and more software-defined vehicles. Ideally, these applications should be kept up to date regularly with security enhancements or feature improvements. To this end, there's a rapidly growing practice of wirelessly distributing software updates to vechicles, called over-the-air [2]. When such an update fails to perform, it should be possible to roll back this change without impact to the end-user. Thus, security and reliability are of upmost importance in these use-cases. Additionally, these solutions are often tightly integrated with Internet of Things (IoT) devices with limited hardware capabilities. This augments the requirements with the demand for a minimal overhead.

WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fullfill all these requirements. Created to execute binary code alongside JavaScript inside the browser, it is now actively used outside this environment via WASI. This system interface is a set of APIs that facilitates interaction with the filesystem, HTTP calls, a Command-Line Interface etc. With the advent of its second preview release also came the release of the component model, an architecture for building interoperable Wasm libraries, applications, and environments. But currently, these technologies merely aim to, and thus not satisfy, these requirements. Specifically, there aren't any interfaces yet for the Inter-Integrated Circuit (I2C) protocol. This protocol defines a serial communication bus widely used in the IoT ecosystem, like smart cars, health machines etc.

The goal of this dissertation is to facilitate the I2C connection between Wasm applications and its underlying hardware and sensors, while still adhering to the requirements. Furthermore, this dissertation also tries to standardize this method. Effectively enabling its use in the industry and ensuring sustainability. This leads to the following research questions:

RQ1 How can a WebAssembly component control a device using I2C, while keeping the migration cost of existing applications to Wasm as low as possible?

RQ2 How can this method be standardized as part of WASI?

RQ3 How can capability-based security be applied to this API?

RQ4 What is the overhead of this method?

RQ5 How suitable is the WASI Component Model for microcontrollers with constrained resources.

Chapter 2 provides a writeup of all the necessary background knowledge. Chapter 3 gives an overview of the standardization effort. Chapter 4 demonstrates implementations that make use of I2C as part of Wasm. Following up, chapter 5 gives an in-depth review of the overhead.

2

Background

2.1 WebAssembly

Wasm is a binary instruction format for a stack-based Virtual Machine (VM). It is designed as a portable compilation target for programming languages. Binaries have a `.wasm` file extension, there's also a textual representation which has a `.wat` extension. This enables deployment on the web for client and server applications. Examples of web applications using this technology are Adobe Photoshop [3] and Google Earth [4].

Although the name implies it, Wasm is not merely limited to the web. There are runtimes that enable execution on a myriad of platforms, ranging from Linux devices to smartphones or even microcontrollers. Via a system interface that enables direct Operating System (OS) communication, called WASI.

2.1.1 JavaScript integration

Initially, Wasm was designed for near-native code execution speed in the web browser[1]. Therefore, it was designed to run alongside JavaScript, allowing both to work together. In this early stage, compiling to Wasm was supplemented with the generation of the required JavaScript glue code. For this, the emscripten [5] compiler is used.

Outside the web browser, there are also platforms that provide a JavaScript runtime environment, i.e. Node.js [6] and Deno [7]. They both, too, have Wasm support, but no manner of accessing OS functionality directly on its own. For this the WASI Application Programming Interface (API) needs to be utilized.

For `Node.js`, integration with WASI is experimental and does not provide the comprehensive security properties provided by dedicated WASI runtimes. It is uncertain whether full support will ever be implemented. In `Deno` official support has been deprecated due to a lack of interest [8].

[1] To be precise, it's the JavaScript engine inside the browser that added support for WebAssembly.

2.2 WebAssembly System Interface

WASI [9] is a modular collection of API proposals defined with the WIT IDL. It provides a secure and portable way to access several operating-system-like features such as filesystems, networking, clocks and random numbers. This collection is developed under the governance of the WASI Subgroup, a subgroup of the W3C's, the World Wide Web Consortium, WebAssembly Community Group.

This subgroup doesn't provide any implementations, for this the Bytecode Alliance [10] exists, a nonprofit organization of companies containing Fastly, Fermyon, Cosmonic, Intel, Microsoft, Siemens and more. Inside this organization, there's also a subcommunity of people interested in the combination of Wasm and embedded devices. To ratify this subcommunity, a request for a Special Interest Group (SIG) Embedded has been opened with the Bytecode Alliance.

2.2.1 Design principles

The following design principles are core:

- Capability-based security: All access to external resources is provided by capabilities.
- Interposition: A Webassembly instance can implement a given WASI interface, and the consumer WebAssembly instance can then use this implementation transparently.
- Compatibility: If possible, keep the API free of Compatibility concerns, and provide compatibility through libraries.
- Portability: The exact meaning of this is specific to each API, but in globo it means that no engine should need to implement every API in WASI.
- Modularity: The component model's worlds mechanism is used, in order to allow specific sets of APIs to be described which meet the needs of different environments. See chapter 2.3.

From these, capability-based security can be seen as the key defining feature. Capability-based security dictates that access to external resources must be denied, unless it is authorized via capabilities [11]. In the context of WASI, there are two kinds of capabilities:

- Handles: Dynamically indetify and provide access to resources via a 32-bit integer index. This is analogue to file descriptors.
- Link-time capabilities: Functions that require no handle arguments. These are used in situations where it's not necessary to identify more than one instance of a resource at runtime. Used sparingly.

For this security model to work correctly, its default to block all external accesses must be airtight, i.e. the runtime needs to be secure.

2.2.2 The standardization process

The WASI subgroup is further split up into the Community Group and the Working Group. The purpose of the Community Group is to attempt to address all concerns, but no 100% consensus is needed. The Working Group, on the other hand, is there to finalize and ratify mostly complete specifications plus test suites from the Community Group.

The process is split up into five stages of standardization:

Phase 0. Pre-Proposal: The Community Group decides whether the pre-proposal is in scope for WASI.

Phase 1. Proposement of the feature: An overview document must be produced that specifies the feature with reasonably precise and complete language.

Phase 2. Specification text is available: A test suite should be added, and it should pass on the prototype or some other implementation.

Phase 3. The specification gets implemented by engines.

Phase 4. The feature is being standardized: Ownership gets transferred from the Community Group to the Working Group, and two or more Web VM's have implemented the feature.

Phase 5. The feature is standardized: Editors perform final editorial tweaks and merge the feature into the main branch of the primary specification repository.

To go from the one stage to the following, a vote in the subgroup needs to be passed. Except to enter phase 0, here the proposal is still merely an idea.

2.2.3 WIT

WIT is an Interface Description Language (IDL). This means that it is a format that defines how the interface of a component should look like. To this end, it uses the following set of concepts: types, functions, interfaces, worlds and packages. From these, worlds are the most key.

A world describes the capabilities and needs of a component - it says which interfaces are available for outside code to call, the `exports`, and which interfaces it depends on, the `imports`. Thus, only the surface of a component is defined, not the internal behaviour. The internal behaviour is determined when the world is targeted by a component an application or library developer creates. For a component to run, its imports must be fulfilled, by a host or by other components.

On the other hand, a world defines an environment in which a component can be instantiated, and its functionality can be invoked.

Regarding the other concepts, functions can only be declared as part of an interface, or as an import or export in a world. Finally, a package is not a world, but can be seen as more like a namespace. It's a way of grouping related interfaces and worlds together for ease of discovery and reference.

WIT also provides built-in types, including primitives like signed and unsigned integer types, floats, strings, and more complex types like results, options and lists. In these, there are two small intricacies worth pointing out. First, both the `char` and `string` types are Unicode. Second, there's the user-defined `resource` type. This type can be seen as an object that implements an interface, and therefore behaviour is only exposed through methods.

To make managing dependencies inside your WIT definition easier, the `wit-deps` [12] project can be used. It makes it possible to lock your dependencies to a certain version and to check if they're the most recent one.

`witx`

In older tooling, it is possible to come across `witx` instead of WIT. This was the IDL used during `Preview 1`. It was derived from `wat`, see chapter 2.1, and had a low-level C-like type system that emphasized raw pointers, and callees were expected to have access to the entir lineair memory of the caller.

2.2.4 Versions

At the time of writing, WASI is in `Preview 2`. The flagship feature of this preview is the release of the component model, see section 2.3. Furthermore, two WIT worlds are now included:

- `wasi-cli`: A command-line interface, roughly corresponding to POSIX.
- `wasi-http`: An HTTP proxy.

The major banner of the upcoming 0.3 is asynchronous support. The exact details of what this asynchrony entails is yet to be determined. Following up will be a stable 1.0 release.

2.2.5 Runtimes

A runtime system is a binary that is accountable for the running of Wasm binaries. Analogue to the role of a hypervisor for a virtual machine. Depending on the capabilities of the runtime, it contains the following noteworthy high-level items:

Engine: An engine stores and configures global compilation settings like optimization level, enabled wasm features, etc.

Module: This represents a compiled form of the input preview 1 wasm module.

Component: A compiled preview 2 component ready to be instantiated.

Linker: A component-style location for defining host functions. This is not the same as `wasmtime::Linker` for modules.

Instance: An instance of a component, or module, is the actual object on which the functions are called.

Store: The store owns the instances, functions, globals, etc.

The Bytecode Alliance maintains two runtimes, Wasmtime [13] and WAMR [14].

Wasmtime can be seen as a general-purpose runtime, focussing on server-side and non-web embeddings with components. It has full component model support, first-class support for eight languages, and community support for a further two. This makes it the de-facto runtime.

On the other hand, WebAssembly Micro Runtime (WAMR) is specifically designed to be as lightweight as possible, targeting embedded devices and the edge. This translates itself into the provided features and the supported guest languages. Support for the component model is planned for the end of 2024, and it only has robust support for C/C++. A toolkit for Rust has been published in March 2024 [15], but this is still novel.

In Rust, the standard library, abbreviated as `std`, is a set of minimal shared abstractions for the broader Rust ecosystem. This library is enabled by default, and can be opted out via the `no_std` attribute. Rejection is useful when targeting a platform that does not support the library or purposfully doesn't use the capabilities of `std`. Historically, the community surrounding Wasmtime has been strongly opposed to the inclusion of a `no_std` build. This is no longer the case. When such a build will eventually be available, the gap between Wasmtime and WAMR will be reduced.

Besides these two, there are numerous ones provided by other parties, in varying degrees of completeness, targeting other use-cases. There's for example Jco [16], specialized for JavaScript, componentize.py [17], for Python, or Chicory [18], which runs on the Java VM. These are out-of-scope for this dissertation.

2.2.6 Alternatives

Inside the Wasm ecosystem, there's mechanoid [19], a framework for applications on embedded systems and IoT devices that had its first release in March 2024. The framework itself is written in Go, and it has builtin support for the wazero [20] and wasman [21] runtimes. Both runtimes specifically designed for Go developers. It doesn't use WASI, nor does it have any plans for this. This makes it unsuitable for us.

2 Background

In the Node.js ecosystem, the most prominent package available for an I2C connection is `i2c-bus` [22]. This is an addon written in C++ that operates directly on a given file descriptor.

For the web, `chirimen-drivers` [23] could be used, but the documentation is completely written in Japanese. Alternatively, combined use of the WebUSB and Web Serial API could be done, but unfortunately the Web Serial API is only available in Chrome, Edge and Opera.

On Windows, I2C communication can be performed via the
textttHIDI2C.sys [24] driver. Linux allows access to a device from userspace, through the `/dev` [25] interface. For this, the `i2c-dev` kernel module needs to be loaded. Each registered adapter gets a number, starting at 0. To start the communication, first the device file needs to be opened and then the target address must be set with `ioctl(file, I2C_SLAVE, addr)`. Finally, on macOS the connection is controlled with the `IOI2CInterface.h` [26] interface inside the `IOKit` framework.

2.3 The Component Model

The WebAssembly Component Model is an architecture for building interoperable Wasm liraries, applications and environments. These components can be seen as a wrapper around a core module, or other components, which express their interfaces and dependencies via WIT and the canonical application binary interface (ABI). Unlike core modules, components may not export Wasm memory, reinforcing Wasm sandboxing and facilitating interoperation between languages with different memory assumptions.

An ABI can be seen as an agreement on how to pass around data in a binary format, specifically concerned with the data layout at the bits-and-bytes level. The Canonical ABI defined by the component model, specifies how the WIT type definitions are translated to bits and bytes. Internally, a C and a Rust component might represent strings in a quite different way, but the canonical ABI provides a format for them to pass strings across the boundary between them.

In regard to WASI, the component model is the staple feature of its second preview, but it is possible to make use of the WASI interfaces without the component model, and thus this model is entirely optional. By way of comparison to a traditional OS, the Component Model fills the role of an OS's process model, defining how processes start up and communicate with each other, while WASI fills the role of an OS's many I/O interfaces.

To compose multiple components together `wasm-tools` [27] can be used, or visually using the builder [28] app. Specifically for Rust, `cargo-component` [29] is also an option.

2.3.1 Toolchain

In Rust, `cargo-component` can be used to compile code to a preview 2 component. In essence, compiling to `Preview 2` means compiling to `wasm32-wasi` and then converting it to a component via an adapater and the `wasm-tools component new` subcommand. This component then adheres to the WIT interface specified in the configuration file. The adaption is needed because there's no first-class support for `Preview 2` yet. Mainstream support for this is planned for early 2025 [30].

Under the hood, `cargo-component` relies upon `wit-bindgen` [31] for binding with the interface. Besides Rust, `wit-bindgen` also supports the following languages: C, Java, Go and C#. For JavaScript, ComponentizeJS [32] can be used.

Adapter modules

The Wasmtime runtime publishes adapter modules with each release, they provide the bridge between the `Preview 1` ABI and the `Preview 2` ABI. The following three modules are provided:

- Command: For command-line applications

- Reactor: Applications that don't have a `main` function
- Proxy: For applications fed into `wasmtime serve`

The `wasmtime serve` subcommand runs a component inside the `wasi:http/proxy` world, supporting the sending and receiving of HTTP requests.

2.3.2 Running components

Running a component is done by calling one of its exports. This can require a custom host, otherwise the `wasmtime` command line can be used.

The job of a custom host is to load a component and execute it through the usage of a Wasm runtime. See section 2.2.5 for a shortlist of the available ones. To guarantee a correct execution, it is important to make sure that any missing interface imports are filled in here, see the earlier section 2.2.3. When using `wit-bindgen`, this is done via the `with` option inside the `bindgen` [33] macro.

When the component exports the `wasi:cli/run` interface, and imports only interfaces listed in the `wasi:cli/command` world, it is considered a command component. Command components can be executed by the `wasmtime run` subcommand. This will compile the module to native code, instantiate it and optionally execute an export.

2.4 I2C fundamentals

As we will be leveraging the I2C protocol, it is worth looking into the inner workings.

I2C [34] is a serial communication bus invented in the eighties by Philips Semiconductors. It uses only two bidirectional lines, a Serial Data Line (SDA) and a Serial Clock Line (SCL). Typically, 7-bit addressing is used, but there exists a 10-bit extension. This extension is fully backwards compatible, allowing a software-emulated 10-bit addressing implementation if the hardware only supports 7-bit addressing.

This communication bus has no minimum frequency, but can go as fast as 5 Mbit/s. Not every Microcontroller Unit (MCU) supports every frequency though, for example the PCF8523 Real-Time Clock only supports up to 1 MHz.

Besides a 0 or 1 data bit, there are two special START and STOP signals which act as message delimiters.

2.4.1 Operations

A node on the bus can have one of two roles[2]:

- Controller: Generates the clock, via required minimum periods for the low and high phases of the SCL, and initiates communication with targets.
- Target: Receives the clock and responds when addressed by the controller.

Any number of any type can be present, and these may be changed between messages. They can also both receive and send data, when in the corresponding mode.

Initial communication is established by a controller that sends a START followed by the address of the target it wishes to communicate with, which is finally completed by a single bit indicating if it wishes to write (0) or read (1) from the target. If the target exists on the bus, it will respond with an acknowledgement. This Acknowledgement (ACK) corresponds with transmitting a single 0 bit, there is also a Negative-acknowledgement (NACK) which is a single 1 bit.

Further communication is performed by one party sending data, most significant bit first, and the other sending an ACK bit.

Bus sharing

In the case of multiple targets linked with one controller, the controller needs to indicate which target it wants to interact with. To achieve this, each target compares the address sent by the controller with its own. If the

[2] In earlier literature, the terminology master and slave were used for respectively controller and target.

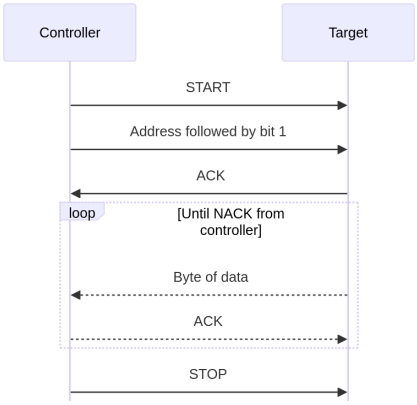


Figure 2.1: Example sequence of a read operation.

address matches, it sends a low voltage ACK bit back to the controller. If the address doesn't match, the target does nothing and the SDA line remains high.

When there are multiple controllers, issues can arise, precisely when they try to send or receive data at the same time over the SDA line. To solve this problem, each controller needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, another controller is in control of the bus, and it should wait until a STOP has been received to send the message. If the SDA line is high, then it's safe to transmit the message.

Methods

The protocol defines three basic types:

- Write: Controller writes bytes to the target.
- Read: Controller reads bytes from the target until a given buffer is full.
- Write-read: Controller first writes bytes and then reads enough bytes to fill the buffer in a single transaction.

It is also possible to combine a list of write and read operations inside a transaction contract. Figure 2.1 showcases more in detail how a read operation works.

2.4.2 SMBus

The System Management Bus (SMBus) [35] is a subset derived from I2C by Intel. Its main application is to monitor critical parameters on PC motherboards and in embedded systems. On the surface they are quite similar, but there are some subtle differences worth mentioning. For one, SMBus will time out when SCL is held low for more than 35 milliseconds. I2C doesn't have an established timeout value, implicating that a target or controller can

hold SCL as long as necessary to process data.
On account of this timeout, SMBus has a minimum clock speed of 10 kHz. Leading to a maximum of 100 kHz. As stated earlier, I2C can go as fast as 5 Mbit/s and no minimum frequency is specified.

Another difference is in terms of voltage levels. For I2C the typical levels are +5 V, +3.3 V or even +1.8 V and below. In contrast, in an SMBus system the supply ranges are restricted between +1.8 V and +5 V. In general, even with the different specifications for the input logic voltage thresholds, I2C and SMBus devices will be interoperable over the supply voltages permitted by the SMBus specification.

Sometimes libraries that provide methods for I2C communication, also provide ones for SMBus. But, thus, in the context of driving an I2C target device, these can be safely ignored.

2.5 Interfacing hardware

Embedded devices have a high degree of diversity of possible constraints, e.g. 64-bit support, memory size and the availability of hardware units like a memory protection unit. Making it difficult for drivers to support any number of target platforms, unless these platforms are abstracted away behind a shared API. This is the purpose of a Hardware Abstraction Layer (HAL). It is important that this layer hides device-specific details and that it is generic across devices.

For Rust, this HAL is, aptly, named `embedded-hal` [36] and provides traits for using peripherals commonly available in microcontrollers such as general-purpose input/output (GPIO), universal asynchronous receiver-transmitter (UART), Serial Peripheral Interface (SPI) or I2C. There exists many crates that implement these interfaces for a certain microcontroller family or a system running some OS. Furthermore, there are also loads of driver crates that use the `embedded-hal` interface to support all these families and systems. A curated list can be found in the Awesome Embedded Rust [37] repository.

Sadly, the notion of a community-wide shared interface is not universally present in all embedded communities. The C/C++ community is such an example, where there isn't one HAL to rule them all.

2.5.1 Different versions of `embedded-hal`

Unfortunately, there are two major versions of `embedded-hal`, i.e. 0.2.7 and 1.0, which are incompatible with one another. As version 1.0 was only released on the ninth of January 2024, it is still fairly novel. Thus, crates have a wildly varying degree of compliance with this version.

Broadly speaking, there are four major changes [38]. Firstly, traits have been simplified and others have been merged to remove interopability gotchas. Secondly, async versions of the blocking traits are now available in the `embedded-hal-async` crate. Thirdly, there is now support for SPI bus sharing. Lastly, there is improved error handling.

There is a crate [39] that tries to provide a compatibility layer between these two versions, but the latest supported version is merely a release candidate of 1.0. Thus, the crate is not really practically useful.

2.5.2 Peripheral Access Crates

System View Description (SVD) files are `XML` files typically provided by silicon vendors which descibe the memory map of a device. Via the `svd2rust` crate [40] it is possible to generate a mostly-safe Rust wrapper. Further discussion is out of scope as this is a very thin wrapper, and usually depended upon by HAL authors.

2.5.3 Running the solution

When the target platform is an OS, it is typically fairly easy to build and execute a software solution, plainly by doing this on the target device itself, or by cross-compilation from a morepotent device. This is not the case for an MCU, here, only cross-compilation is possible. Due to the constrained nature of memory on an MCU, the memory-layout also needs to be specified.

In the case of a Raspberry Pi Pico, compilation results in an `UF2` and an `ELF` file. The former is a file format developed by Microsoft for flashing microcontrollers over mass storage connections. The latter is used by the debugger.

To pogram the flash on the Pico, the `BOOTSEL` button needs to be held. Forcing it into USB Mass Storage Mode. Then, you can move a `UF2` file onto it. Whereupon the `RP2040` processor of the Pico will reboot, unmount itself, and run the flashed code. Other boards could require pulling down the flash `CS` pin, which is how the `BOOTSEL` button works on the pico, using an exposed Serial Wire Debug (SWD) interface, also an option for the Pico, or have a reset button that needs to be double-pressed.

SWD is a standard interface on Cortex-M based microcontrollers, which the host machine can use to reset the board, load code into flash, and set the code running. Without the need to manuallyl reset the board or hold the `BOOTSEL` button. The easiest way to connect with this interface on a Pico is to make use of a debug probe via `probe-rs` [41]. This also unlocks the ability to print to `STDOUT` or even utilize the Debug Adapter Protocol [42].

3

Architecture and standard

Currently, the I2C proposal [43] is in the first phase, with ongoing effort to fullfill the criteria to pass the vote to the second phase. Specifically, the following requirements are yet to be met:

- A broad enough of a consensus needs to be reached on the capability criteria
- Some pull requests and issues are still open that iterate on the design of the feature

This effort is led under the guidance of certain champions, for this proposal these are Friedrich Vandenberghe, Merlijn Sebrechts and Maximilian Seidler. Both Friedrich and Merlijn are from UGent, Maximilian is from Siemens. This mix of academians and people from the industry ensures ongoing standardization effort and actual usage of the feature.

At the start of this thesis the proposal was merely an idea, but now it contains three WIT files that follow the component model: `delay`, `i2c` and `world`. The first two closely follow the corresponding interfaces from `embedded-hal` version 1, `embedded-hal` is explained in section 2.5, but this crate is not necessary for an implementation. The philosphy behind this is the same as for why one would want to have an `embedded-hal`.

```
package wasi:i2c@0.2.0-draft;

world imports {
    import i2c;
    import delay;
}
```

Codefragment 3.1: The `world.wit` file inside the `wasi-i2c` proposal.

The proposal defines two handles, for an explainer on handles see 2.2.1, one for I2C and one for delays. These provide a pretty broad access, e.g. it is not possible to limit the I2C connection to certain addresses. A more fine-grained control model is possible, but there's currently no demand for this.

Inside the `i2c` interface, no explicit constructor is defined because `i2c` resources can be constructed in many different ways, so worlds that include this interface should also include a way to obtain I2C handles. Typically, this

is done by either having handles passed into exported functions as parameters, or by having handles returned from imported functions. The definition for `world.wit` can be found in codefragment 3.1. The purpose of this world is to provide a way to import all the defined interfaces in the proposal at once.

Although delays are not inherently linked with I2C, some guests implementations require this, and thus it is part of the proposal. In the future, it could be that delays get split up into their own proposal, or that this proposal gets merged with the other embedded-related proposals. The biggest drawback to a merger would be that this, possibly, could heavily increase the burden on the champions. Furthermore, this would require that all the champions are interested in maintaining all of these other communication protocols. On the other hand, this would centralize the proposals and could make it easier for potential implementors.

3.1 Asynchronous I2C

Currently, the proposal is purely synchronous. To provide an asynchronous (async) API, there are two possible ways to tackle this:

Explicit A seperate WIT API could be provided that depends on `wasi-io` [44] polling. This could be implemented with the current tooling.

Integrated We could also wait for the Component Model to natively integrate async. The major upside to this approach is that a single WIT description can describe both a sync and async API. Caller and callee can each independently chose if they want to be sync or async, and they can be linked.

As there's no immediate need for this, the champions have decided that it is best to wait until the integrated option is mainstream.

3.2 Portability criteria

An important part of a proposal is its portability criteria, with these the champions show that their specification isn't overspecified for a specific platform. Commonly, these are two complete independent implementations, but due to the diversity in platforms that can interact with I2C this isn't suitable for the `wasi-i2c` proposal.

The champions, together with people from Siemens, Intel, Aptiv, Xiaomi, Amazon, Midokura, Sony, Atym and Bosch, have agreed upon a set of criteria for the proposal. These criteria can be found in table 3.1. For each criterium, reference hardware is provided, without these there would still be a lot of possible variety in RAM size, flash size and CPU speed. Herein Intel x86 is missing, but support for this is implied as this is the used architecture for developing Wasmtime. The Nucleo is chosen because this is the reference board for Siemens. The other two are chosen to showcase support for both RTOS and Linux, and ARM and RISC-V. As an extra requirement besides the

Platform	Architecture	Reference hardware
Linux	ARM64	Raspberry Pi 3 Model B
RTOS	RISC-V	ESP32-C2
RTOS	ARM32	Nucleo F412ZG

Table 3.1: `wasi-i2c` portability criteria

targets, it is important that the interface is designed in such a way that memory usage is limited to the bare needed minimum. To ensure enough RAM left for the application itself.

4

Implementations

Implementations are provided to ascertain the soundness of the WIT interfaces from `wasi:i2c`. These solutions reside inside the `i2c-wasm-components` repository [45]. Figure 4.1 contains the parts that are noteworthy.

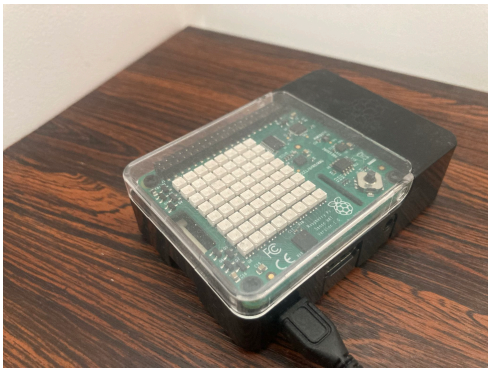
```
i2c-wasm-components
├── native
│   ├── hat
│   └── segment_led
│       ├── pi
│       └── pico
├── wamr
│   ├── guest
│   └── host
└── wasmtime
    ├── guest
    │   ├── hat
    │   └── segment_led
    ├── host
    ├── wit
    ├── empty.wasm
    └── empty.wat
```

Figure 4.1: Directory structure of the implementations repository.

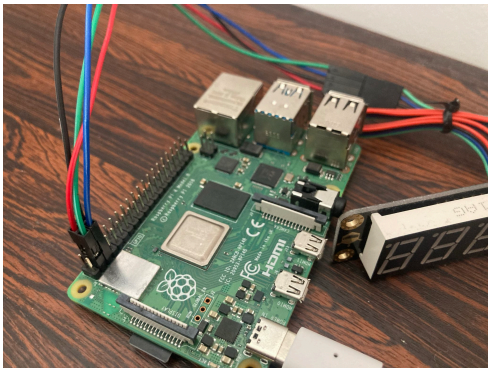
As stated in section 2.4.1, there are three basic types of methods, from which two are implemented, write and read. For writing, a 4-digit 7-segment display[1] is used, and for reading a HTS221 sensor, from which the current temperature and humidity can be read. These are respectively `segment_led` and `hat` in the implementation.

[1] Because each digit is represented by seven segments, parts of the latin alphabet can also be shown.

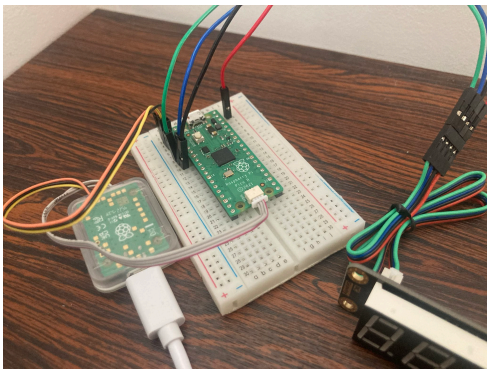
The sensor is part of a Hardware Attached on Top (HAT) mounted atop of a Raspberry Pi 3 Model B via its 40-pin GPIO header, see Figure 4.2a. Although it is connected this way, reading from the sensor is still done via I2C. The display, on the other hand, is a standalone device and can thus easily be hooked up with a different controller. As a controller, we use a Raspberry Pi 4 Model B, Figure 4.2b, and a Raspberry Pi Pico, Figure 4.2c. The extra device the Pico is connected with is a debug probe, see section 2.5.3. Both the Pi 3 and the Pi 4 run on an ARM64 architecture and a Linux platform, contrarily, the Pico is an MCU targeting RP2040. Table 4.1 shows a concise overview. The used platform and architecture combinations are different from the portability criteria because these were decided before the criteria were decided upon.



(a) Pi 3



(b) Pi 4



(c) Pi Pico

Figure 4.2: Setups of....

Connected device	Hardware
HTS221 sensor	Raspberry Pi 3 Model B
4-digit 7-segment display	Raspberry Pi 4 Model B
4-digit 7-segment display	Raspberry Pi Pico

Table 4.1: Concise overview of the setups

4.1 Native development in Rust

The Pi 4 can build the codebase, but the other two lack the capabilities for this, thus cross-compilation needs to be performed. Fortunately, Rust provides great support for this via the `target` flag. ARM64 can be targeted with `aarch64-unknown-linux-gnu`, and RP2040 with `thumbv6m-none-eabi`.

As the Pico is an MCU, it has some further requirements. First, it only supports `no_std` build, see section 2.2.5. Second, the memory layout also needs to be specified inside a file called `memory.x`.

To easily interface with the HTS221 sensor, the homonymous `hts221` Rust crate [46] can be used. Unfortunately, for `embedded-hal` version 1 integration, a fork [47] had to be created. The `hts221` crate requires an I2C connection that follows the HAL specification, for this the `linux-embedded-hal` crate [48] is used.

For the implementation running on the Pi 4 we write the current local time. To establish the I2C connection the `rppal` crate [49] is used. This library also follows the `embedded-hal` API, but is specifically tailored for Raspberry Pi devices. Due to the limited capabilities of the Pico, a simple incrementing counter is instead displayed. For the connection, the `rp-pico` crate [50] is now utilized.

4.2 Embedded driver development in WebAssembly

To make the move from native to WebAssembly, everything related to the I2C connection itself is kept host-side. The device-specific logic, like reading the current temperature, is moved to the guest. Ideally, each implementation, conceptually, comes down to the schematic defined in Figure 4.3. But due to the myriad of differences between Wasmtime and WAMR, outlined in section 2.2.5, no WebAssembly-specific code can be reused between them. This implementation, therefore, keeps everything each runtime interacts with inside two seperate directories.

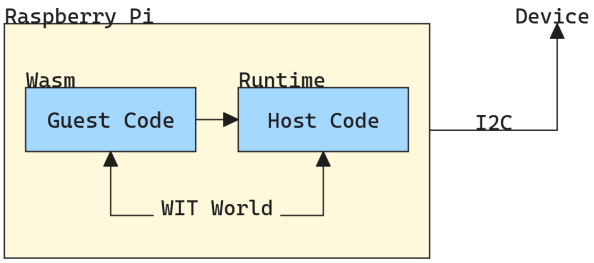


Figure 4.3: Schematic overview of an implementation in WebAssembly.

4.2.1 Schematic implementation for Wasmtime

The Wasmtime implementation consists of three directories: `guest`, `host` and `wit`. The `wit` directory specifies two worlds, corresponding with the two devices being interacted with. This implementation thus contains two implementations that follow the schematic described by Figure 4.3.

WIT world

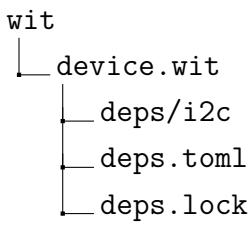


Figure 4.4: Structure of the `wit` subdirectory inside Wasmtime

```
world sensor {
    import wasi:i2c/i2c@0.2.0-draft;
    export hts: interface {
      use wasi:i2c/i2c@0.2.0-draft.{i2c, error-code};
      get-temperature: func(connection: i2c) -> result<string, error-code>;
      get-humidity: func(connection: i2c) -> result<string, error-code>;
    }
}

world screen {
    include wasi:i2c/imports@0.2.0-draft;
    export display: interface {
      use wasi:i2c/i2c@0.2.0-draft.{i2c};
      use wasi:i2c/delay@0.2.0-draft.{delay};
      write: func(connection: i2c, delay: delay, message: string);
    }
}
```

Codefragment 4.1: The WIT worlds to which guest and host bind.

Both the guest components, as the Wasmtime host implementation, make use of the WIT from codefragment 4.1, as stipulated in section 2.3.1 and corresponding with the `device.wit` from Figure 4.4. Herein, `wasi:i2c` is the proposal, see chapter 3. The display is represented by the `screen` world, which needs both `i2c` and `delay` from the proposal. The HTS221 is mapped with the `sensor` world, that only needs `i2c`. Therefore, the former just includes the imports from `wasi:i2c`, while the latter only imports `i2c`. The idea behind this WIT

description is to keep it as generic as possible, this is why the `write` function accepts a string while the display only has four digits.

The `deps/i2c` contains the WIT interfaces from the proposal, both the `deps.lock` and `deps.toml` are files required for the correct workings of `wit-deps`.

Guest code

To reduce the size of the generated Wasm binaries, the guest implementations don't make use of Rust's std. For the generation of the guest components `cargo-component`, see section 2.3.1, requires an adapter. Typicallly this adapter enables the usage of the std, but as the guest implementations doesn't make use of this an adapter is no longer needed. To solve this conundrum, an empty adapter is given to `cargo-component` via `empty.wasm` and `empty.wat`. Note that this uses `#![feature(...)]`, which requires the usage of the nightly version of Rust.

It would be ideal if the Wasm implementation of the sensor read would still use the `hts221` crate. For this, code needs to be supplemented to the generated bindings to make them compatible with the `embedded-hal` API. This supplementation is done via the `add_i2c_hal` macro inside the `wasi-embedded-hal` crate [51].

```
#![no_std]
#![no_main]
mod bindings;

// To make the generated bindings compatible with the `embedded-hal` API
add_i2c_hal!(i2c);

struct Component {}

impl Guest for Component {
  // Omitting implementations of the exports inside of the binded world
}

// Omitting the definition of a global alllocater and a panic handler,
//  needed by Rust because of `no_std`.

bindings::export!(Component with_types_in bindings);
```

Codefragment 4.2: Stripped down version of a guest component for Wasmtime.

Codefragment 4.2 describes how, conceptually, each component implementation looks like.

Figure 4.5: Structure of the Wasmtime host directoy structure vis-à-vis with its logical structure.

Host code

The WIT interface imports (parts of) the `wasi:i2c` interfaces. Therefore, it is not possible to run components via the `wasmtime` command line, explained in section 2.3.2, and a custom host is necessary. The most basic version of this custom host would simply be a standalone implementation of each world. Every world implementation then has their own fulfillment of the imports, resulting in lots of code redudancy. Therefore, a more involved host implementation, like the structure defined in Figure 4.5, is more practical. Here, the world implementations can share their usage of `device.rs` for the fulfillment of the imports via the `with` keyword, as demonstrated inside codefragment 4.3. It is important that these paths be fully qualified. Otherwise errors will be thrown by the Rust compiler.

Each world implementation in itself is a module that splits the standard procedure, see 2.2.5, for executing a method from a Wasm binary in two functions:

1. `new`: Responsible for adding the implemented i2c and delay structs from device to the linker, creation of the host state and populating it, creation of the store and instantiation of the bindings.
2. `run`: Executes a certain function from the component. The parameters are hardcoded.

The purpose of this deviation from the standard procedure is twofold. First, it allows calling the exported function without the need to instantiate the component each time. Second, it provides a shared function signature for running a component.

4.2.2 Schematic implementation for WAMR

This implementation represents the schematic from Figure 4.3 for WAMR, hower there's no support for preview two. This aplication, thus, cannot make use of WIT, nor pass a proper I2C connection to the guest because there's no notion of a WIT resource. Therefore, the connection is now kept global. In Rust, there's no default way to keep a variable global, but for this implementation the `once_cell` [52] crate is used.

All guest-to-host and host-to-guest communication is considered unsafe on a memory-safety basis, this is because WAMR is written in C, a language infamous for how unsafe it is. Furthermore, the WAMR Rust SDK mandates

```
// device.rs
bindgen!({
    path: "../wit/deps/i2c",
    world: "wasi:i2c/imports",
    with: {
        "wasi:i2c/i2c/i2c": device::I2c,
        "wasi:i2c/delay/delay": device::Delay,
    }
});
// display.rs
bindgen!({
    path: "../wit",
    world: "screen",
    with: {
        "wasi:i2c/i2c/i2c": device::I2c,
        "wasi:i2c/delay/delay": device::Delay,
    }
});
```

Codefragment 4.3: Usage of the `with` keyword inside the `bindgen`'s of `device.rs` and `display.rs`.

that imports and exports are called and developed as extern C functions. This leads to a module implementation conceptualized by codefragment 4.4. The host cannot directly bind with the resulting binary, instead it has to register the import and find the exports. For passing parameters to the exports, a vector of `WasmValues` is passed. A `WasmValue` can be a void, 32-bit signed integer, 64-bit signed integer, 32-bit float, 64-bit float or a 128-bit signed integer. Each value is underlying stored as an array of 32-bit unsigned integers. The exported `setup`function can then be called with an array containing a `WasmValue::Void`, for `write` a vector of four `WasmValue::I32`'s are passed.

On the Raspberry Pi Pico, there's no filesystem, thus no capability of reading the Wasm binary from disk. Instead, a `hexdump` can be taken from the binary, and then these bytes can be hardcoded as a vector of unsigned 8-bit integers inside the host.

Under the hood, Rust uses `rust-lld` for the cross-compilation to the Pico [53], but WAMR uses the `arm-none-eabi-gcc` linker. This results in a CMAKE error when building the SDK [54], and thus disabling cross-compilation.

```rust
#![no_std]
#![no_main]

#[link(wasm_import_module = "host")]
extern "C" {
    // The term _slave_ is still used because `rppal` still uses this.
    fn host_i2c_write(slave_address: u16, data: u8);
}

#[export_name = "setup"]
pub fn setup() { /* Omitting definition */ }
#[export_name = "write"]
pub fn write(d0: i32, d1: i32, d2: i32, d3: i32) { /* Omitting definition */ }

// Omitting the definition of a global alllocater and a panic handler,
//  needed by Rust because of `no_std`.
```

Codefragment 4.4: Stripped down version of a guest module for WAMR.

5

Evaluation

5.1 Evaluation setup

The evaluation setup is based upon the setups used for the implementations, as shown in Chapter 4. It consists of the following:

- A Raspberry Pi 4 Model B with 8 GB of RAM hooked up with a 4-digit 7-segment LED
- A Raspbery Pi 3 Model B with an attached HTS221 sensor

The display is evaluated natively and with Wasmtime and WAMR as a runtime. The sensor will be evaluated in the same manner, but without the WAMR runtime.

The execution time is measured with the `criterion.rs` crate [55]. This crate performs a hundred measurements, each containing many iterations of a routine, and then accumulates this into a probability density function. It indicates the estimated probability of an iteration taking a certain amount of time. Furthermore, the plot also contains a vertical line, this indicates the mean execution time. Thanks to the immense range of factors that can influence the execution time, outliers will always occur, and the resulting density functions aren't set in stone. Due to these outliers, sometimes a smaller extra peak can appear. Before measuring, Criterion first performs a warmup phase. Here, the routine is executed repeatdly to give the system time to adapt to the new workload. This helps prevent things like cold caches.

For memory profiling, `dhat-rs` [56] is used. The profiler makes use of a global allocator that tracks allocations and deallocations on the heap. After execution, the total number of bytes, the maximum amount and the size at the end is printed. Together with the number of allocations.

The `flamegraph` crate [57] is used to provide an in-depth overview of the functions an implementation uses under the hood. This overview is provided in fashion of flamegraphs. Many times per second, the threads in a program are interrupted and the current location in the code is recorded, along with the chain of functions that were called to get there. These samples are then processed and stacks that share common functions are added

together. Then a figure is generated showing the call stacks that were measured. The x-axis doesn't show the passing of time. The left to right ordering has no meaning. The width of each block shows the total time that that function is on the CPU. A wider block means that it consumes more CPU per execution than other functions, or that it's called more. Each block's color is chosen at random.

5.2 Functional evaluation

This dissertation evaluates the applications on their execution time and used memory compared between running natively versus inside the Wasmtime and WAMR runtimes.

For this, two scenarios are tested:

- On the Raspberry Pi 3 the temperature gets read out. This value is manually verified with the native implementation.
- On the Raspberry Pi 4 the string `"1234"` gets written to the display. This is visually verified, see Figure 5.1.

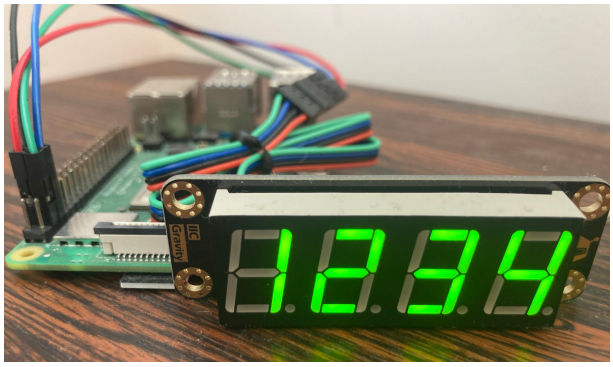


Figure 5.1: Visual verification of the display

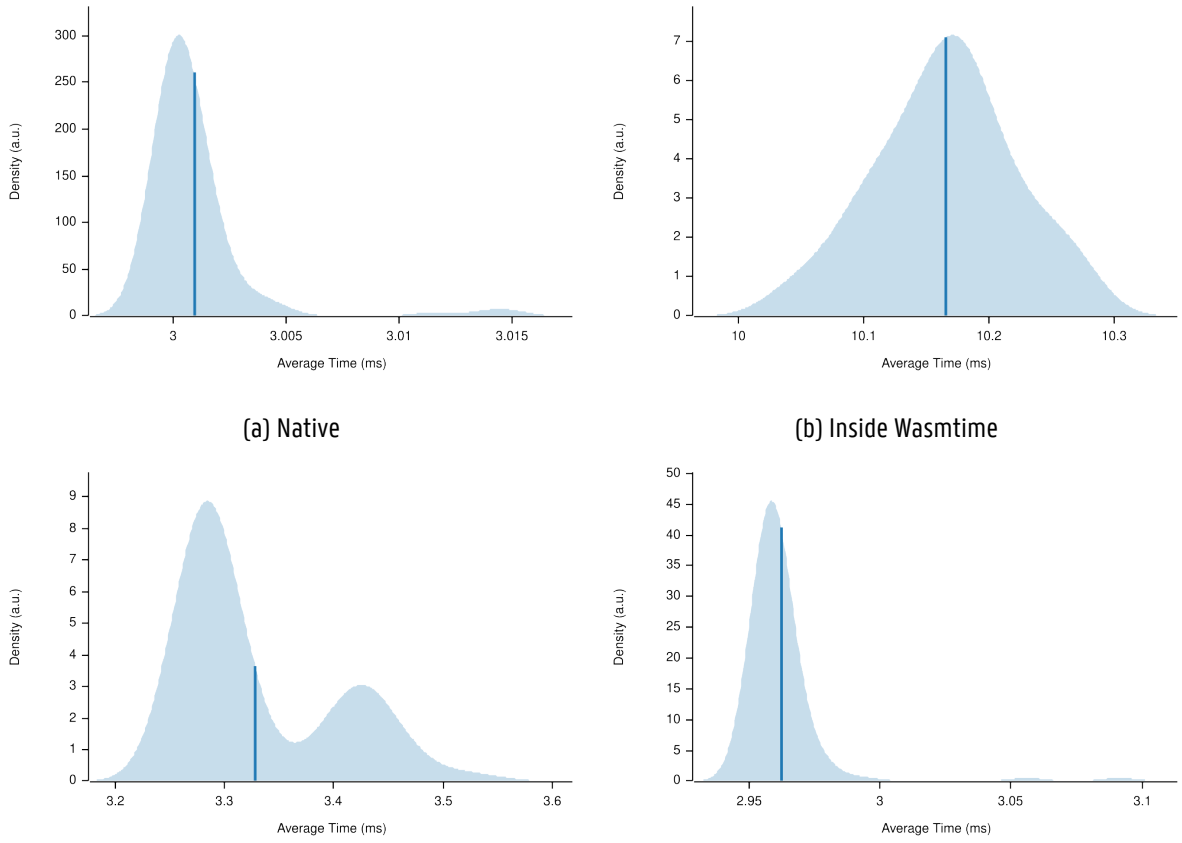
5.3 Benchmarks

It is expected that running in Wasmtime will perform worse due to the imposed overhead on both execution time and memory usage, but that this overhead will be negligible. For WAMR, an even slimmer overhead is expected.

First, the execution time is inspected. Both Figure 5.3 and Figure 5.2 showcase an astonishing longer execution time when running inside Wasmtime. The flamegraph in Figure 5.6b gives the probable explanation, namely writing to the display is done inside the `_start` block, almost all the other blocks are related to Cranelift.

Bytecode Alliance's Cranelift is a compiler backend that is among others in use by Wasmtime for just-in-time and ahead-of-time compilation. Via its ahead-of-time functionality, it is possible to greatly reduce the average exe-

cution time. There are two options to make use of this, through `wasmtime compile` or the `Component::serialize` function inside Wasmtime. To load such a precompiled file, the `Component::deserialize_file` function needs to be used, instead of `Component::from_file`. This file contains native, non-portable binary code which will not work on a different architecture, and might not even work across different processor models within the same architecture. Thus, it is not feasible to distribute a precompiled Wasm binary, instead of the Wasm binary itself[1].



(a) Native

(b) Inside Wasmtime

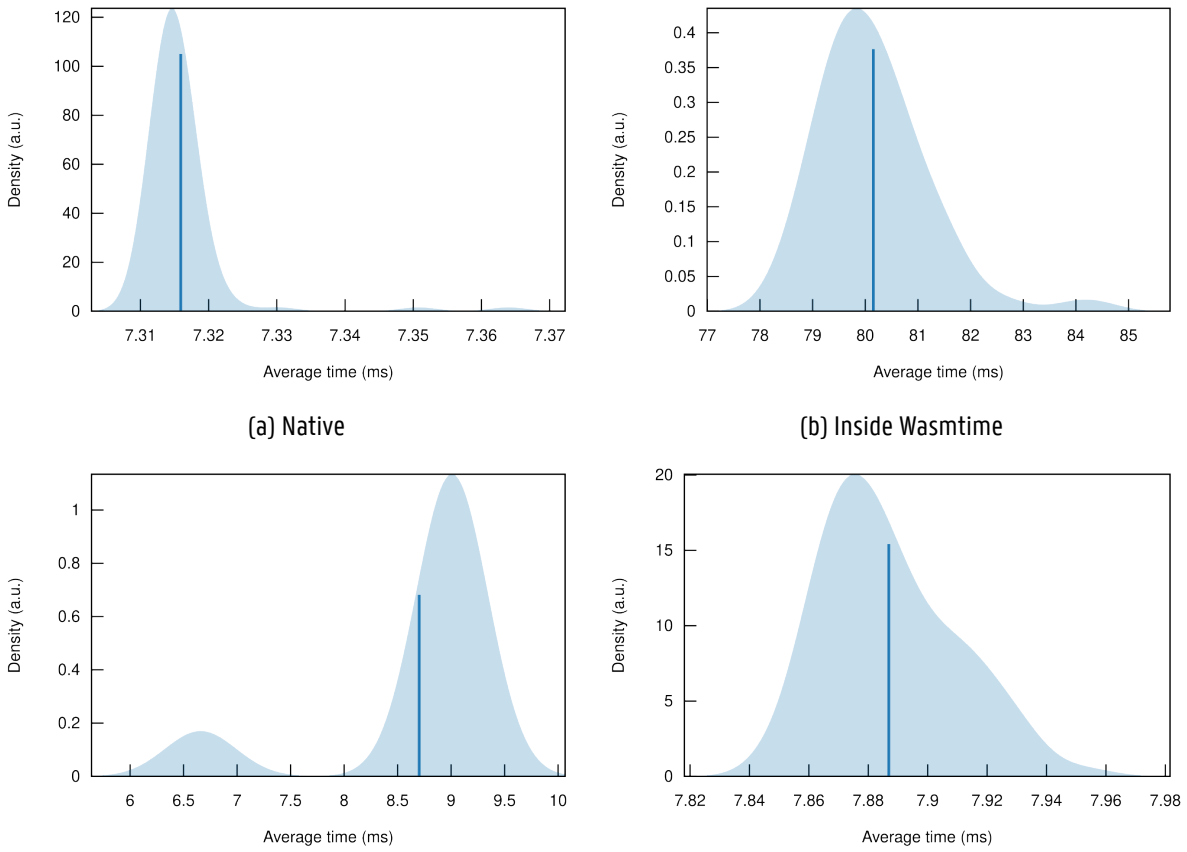
(c) Inside Wasmtime (precompiled)

(d) Inside Wasmtime (precompiled & no command world)

Figure 5.2: Probability density function of execution time when writing to the display on....

Figure 5.2c shows the execution time with a precompiled binary. It now is far more comparable to the average time of the native execution, with only an estimated slower mean run time of 0.3062 milliseconds. The same holds true for reading the sensor, Figure 5.3c, where the slowdown is now merely 1.3859 milliseconds. But, when compared to the experienced slowdown of the display implementation, this is still too much. The flamegraph inside Figure 5.7 shows that nearly 14% of the time is spent with adding the command world to the linker, using the `command::sync::add_to_linker` function, while this world technically isn't needed because of the

[1]Technically, it is possible, but this requires careful configuration and is too complex for this dissertation.



(a) Native

(b) Inside Wasmtime

(c) Inside Wasmtime (precompiled)

(d) Inside Wasmtime (precompiled & no command world)

Figure 5.3: Probability density function of execution time when reading from the sensor on....

custom host setup. Without this world, reading the sensor in Wasmtime is now only 0.571 milliseconds slower, Figure 5.3d, and writing to the display is 38.6 microseconds faster than native, Figure 5.2d. The executed functions, see Figure 5.6a and Figure 5.6c, do not show any prominent differences. Thus, the speedup could be due to noise or kernel intrinsics.

Comparing WAMR with the fastest Wasmtime version for writing to the display, thus Figure 5.4 with 5.2d, shows that Wasmtime outperforms with 88.2 microseconds. This is not significant enough of a difference for one to be faster or slower than the other. Looking at the flamegraph for WAMR, Figure 5.6d, it is apparant that a considerable amount of time is spent on a page translation fault.

When considering the memory usage, three phenomena are striking. First, the performed optimizations to make the Wasmtime implementation run faster also greatly benefit the memory usage. Second, WAMR has near-native memory usage, while Wasmtime uses nearly four times more memory when compared to the native version. Third and last, the memory usage of the fully optimized version of Wasmtime is fairly similar between writing to the display and reading from the sensor. This showcases a baseline amount of memory required to run the component

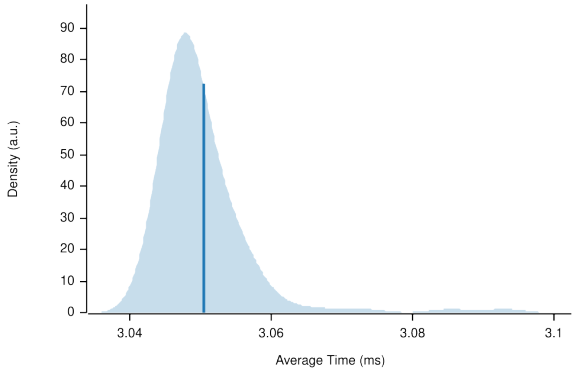


Figure 5.4: Probability density function of the execution time when writing to the display inside WAMR

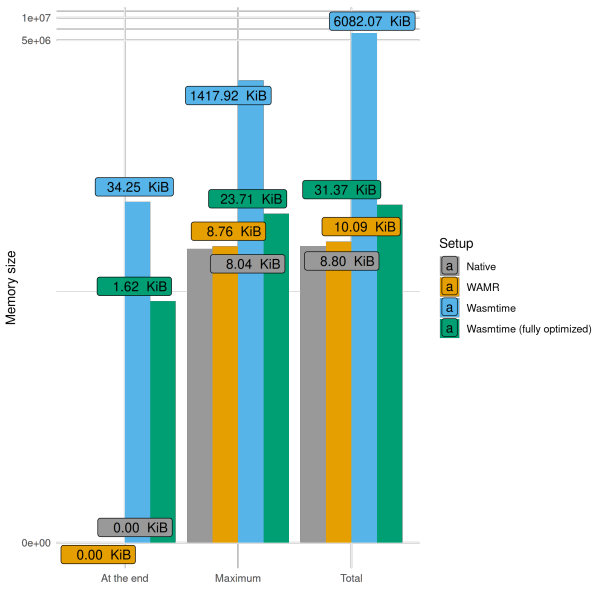


Figure 5.5: Used memory size when writing to the display.

	Total	Maximum	At the end
Native	8.8 KiB	8.4 KiB	0 B
Wasmtime	5.94 MiB	1.39 MiB	34.25 KiB
Wasmtime (fully optimized)	31.17 KiB	23.71 KiB	1.62 KiB
WAMR	10.9 KiB	8.76 KiB	0 B

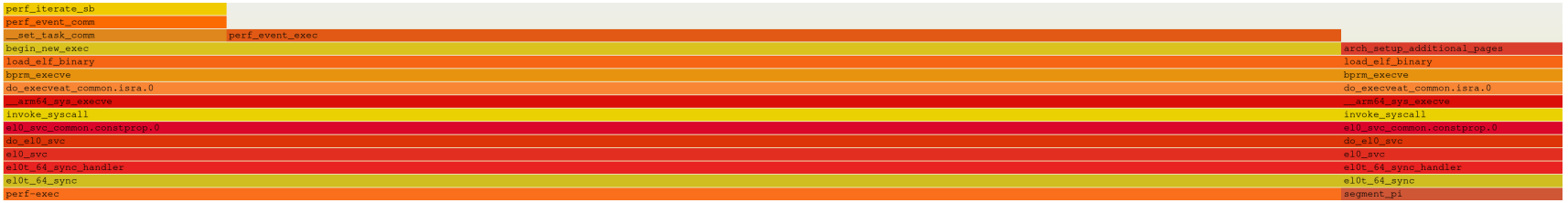
Table 5.1: Used memory for writing to the display

	Total	Maximum	At the end
Native	1.24 KiB	1.03 KiB	1 KiB
Wasmtime	24.67 MiB	2.66 MiB	34.25 KiB
Wasmtime (fully optimized)	37.19 KiB	29.58 KiB	1.62 KiB

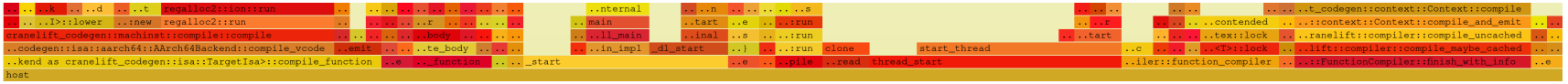
Table 5.2: Used memory for reading from the sensor

model.

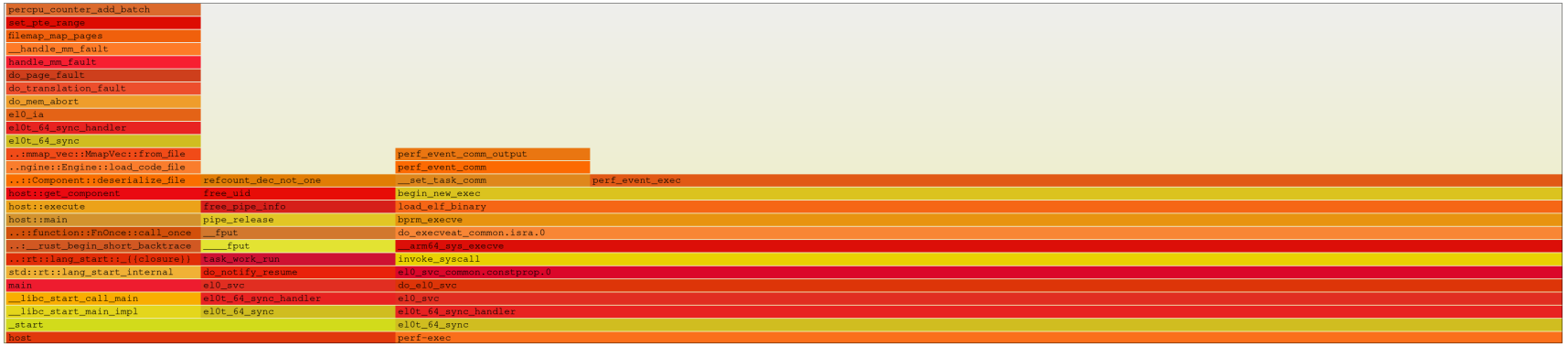
To conclude, both solutions for Wasmtime greatly benefitted from precompiling the Wasm binary. Furthermore, precompilation makes Wasmtime on par with WAMR and native time-wise, but Wasmtime still requires a substantial larger amount of memory to succesfully run.



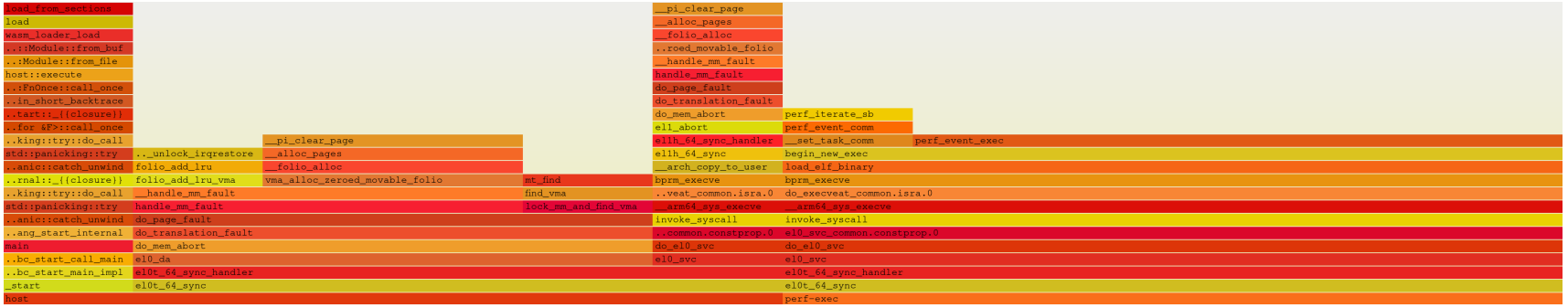
(a) Native



(b) Inside Wasmtime



(c) Inside Wasmtime (precompiled binary & without command world)



(d) Inside WAMR

Figure 5.6: Flamegraphs of writing to the display

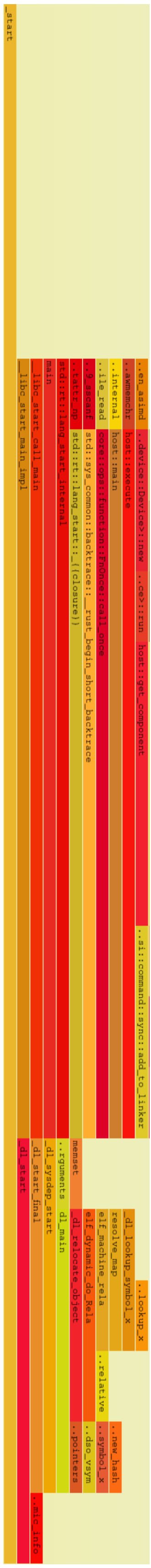


Figure 5.7: Flamegraph of reading the sensor inside Wasmtime with a precompiled component.

Conclusion

[Terugblikken naar de introductie en wat ik allemaal bijgeleerd heb.]

Vertellen over SPI. [Het nog verder advancen van de proposal enbespreken van dingen waar ik niet klaar mee geraakt ben.]

References

[1] S. Harding, "5.25-inch floppy disks expected to help run san francisco trains until 2030," *Ars Technica*, 2024. [Online]. Available: https://arstechnica.com/gadgets/2024/04/5-25-inch-floppy-disks-expected-to-help-run-san-francisco-trains-until-2030/

[2] Aptiv, "What is over-the-air (ota)?" *Mobility Insider*, 2020. [Online]. Available: https://www.aptiv.com/en/insights/article/what-is-over-the-air-(ota)

[3] P. Clark, "Adobe photoshop streamlines power and precision for the web; now includes widely popular adobe firefly powered features," https://blog.adobe.com/en/publish/2023/09/27/photoshop-streamlines-power-precision-web, 2023.

[4] G. Shah, "Welcome home to the new google earth," https://blog.google/products/earth/welcome-home-new-google-earth/, 2017.

[5] emscripten core, "emscripten," https://github.com/emscripten-core/emscripten.

[6] O. Foundation, "Node.js," https://nodejs.org/en.

[7] R. Dahl, B. Belder, and B. Iwańczuk, "Deno 1.0," https://deno.com/blog/v1, May 2020.

[8] A. Jiang, B. Iwańczuk, D. Sherret, K. Whinnery, L. Kettmeir, M. Mastracci, N. Rahman, Y. Hinosawa, and M. Hagemeister, "Deno 1.38: Html doc generator and hmr," https://deno.com/blog/v1.38#deprecation-of-stdwasi, 2023.

[9] P. Hickey, J. Konka, D. Gohman, S. Clegg, A. Brown, A. Crichton, L. Clark, C. Ihrig, P. Huene, Y. Yuji, D. Vasilik, J. Triplett, S. Rubanov, S. Akbary, M. Frysinger, A. Turner, A. Zakai, A. Mackenzie, B. Brittain, C. Beyer, D. McKay, L. Wang, M. Mielniczuk, M. Berger, PTrottier, P. Sikora, T. Schneidereit, katelyn martin, and nasso, "Webassembly/wasi: snapshot-01," *Zenodo*, Dec. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4323447

[10] L. Clark, "Announcing the bytecode alliance: Building a secure by default, composable future for webassembly," https://bytecodealliance.org/articles/announcing-the-bytecode-alliance, 2019.

[11] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM*, vol. 9, no. 3, p. 143–155, mar 1966. [Online]. Available: https://doi.org/10.1145/365230.365252

[12] R. Volosatovs, "wit-deps," https://github.com/bytecodealliance/wit-deps.

[13] L. Clark, "Wasmtime reaches 1.0: Fast, safe and production ready!" https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready, 2022.

[14] B. Alliance, "wasm-micro-runtime," https://github.com/bytecodealliance/wasm-micro-runtime.

[15] ——, “wamr-rust-sdk,” https://github.com/bytecodealliance/wamr-rust-sdk.

[16] Y. Wuyts, “Announcing jco 1.0,” https://bytecodealliance.org/articles/jco-1.0, 2024.

[17] B. Alliance, “componentize-py,” https://github.com/bytecodealliance/componentize-py.

[18] B. Eckel and A. Peruffo, “Chicory: Creating a language-native wasm runtime,” https://www.youtube.com/watch?v=jrnMhWiV23w, 2024.

[19] hybridgroup, “Mechanoid,” https://mechanoid.io/.

[20] A. Cole, “Introducing wazero from tetrate,” https://tetrate.io/blog/introducing-wazero-from-tetrate/, 2023.

[21] hybridgroup, “wasman,” https://github.com/hybridgroup/wasman.

[22] fivdi, “i2c-bus,” https://github.com/fivdi/i2c-bus/tree/master.

[23] C. O. Hardware, “Chirimen drivers,” https://chirimen.org/chirimen-drivers/.

[24] M. Hopkins, “Introduction to hid over i2c,” https://learn.microsoft.com/en-us/windows-hardware/drivers/hid/hid-over-i2c-guide, 2024.

[25] T. kernel development community, “I2c device interface,” https://www.kernel.org/doc/html/v5.4/i2c/dev-interface.html.

[26] “Ioi2cinterface.h user-space,” https://developer.apple.com/documentation/iokit/ioi2cinterface_h_user-space.

[27] B. Alliance, “wasm-tools,” https://github.com/bytecodealliance/wasm-tools.

[28] P. Huene, D. Anderson, and J. Mart, https://wasmbuilder.app/.

[29] B. Alliance, “cargo-component,” https://github.com/bytecodealliance/cargo-component.

[30] Y. Wuyts, “Changes to rust’s wasi targets,” https://blog.rust-lang.org/2024/04/09/updates-to-rusts-wasi-targets.html, 2024.

[31] B. Alliance, “wit-bindgen,” https://github.com/bytecodealliance/wit-bindgen.

[32] ——, “Componentizejs,” https://github.com/bytecodealliance/ComponentizeJS.

[33] wasmtime, “Macro wasmtime::component::bindgen,” https://docs.rs/wasmtime/latest/wasmtime/component/macro.bindgen.html.

[34] *I2C-bus specification and user manual*, NXP Semiconductors, 10 2021, rev. 7.

[35] *System Management Bus (SMBus) Specification*, System Management Interface Forum, 1 2022, rev. 3.2.

5 References

[36] R. Embedded, “embedded-hal,” https://github.com/rust-embedded/embedded-hal.

[37] R. E. R. Team, “Embedded rust,” https://github.com/rust-embedded/awesome-embedded-rust.

[38] E. R. W. Group, “embedded-hal v1.0 now released!” https://blog.rust-embedded.org/embedded-hal-v1/, 2024.

[39] R. Kurte, “Embedded hal compatibility layer,” https://github.com/ryankurte/embedded-hal-compat.

[40] J. Aparicio, “svd2rust,” https://crates.io/crates/svd2rust.

[41] probe rs, “probe-rs,” https://probe.rs/.

[42] Microsoft, “Debug adapter protocol,” https://microsoft.github.io/debug-adapter-protocol/.

[43] F. Vandenberghe, L. Clark, and J. Schilling, “Wasi i2c,” https://github.com/WebAssembly/wasi-i2c.

[44] D. Gohman, “Wasi i/o,” https://github.com/WebAssembly/wasi-io.

[45] F. Vandenberghe, “i2c-wasm-components,” https://github.com/Zelzahn/i2c-wasm-components.

[46] D. Gallagher, “hts221,” https://github.com/danielgallagher0/hts221.

[47] F. Vandenberghe, “hts221,” https://github.com/Zelzahn/hts221.

[48] R. E. L. team, “linux-embedded-hal,” https://github.com/rust-embedded/linux-embedded-hal.

[49] R. van der Meer, “Rppal - raspberry pi peripheral access library,” https://github.com/golemparts/rppal.

[50] rp rs, “rp-pico,” https://github.com/rp-rs/rp-hal-boards.

[51] F. Vandenberghe, “wasi-embedded-hal,” https://github.com/Zelzahn/wasi-embedded-hal.

[52] A. Kladov, “once_cell,” https://github.com/matklad/once_cell.

[53] E. R. W. Group, “Psa: Cortex-m breakage (lld as the default linker),” https://blog.rust-embedded.org/2018-08-2x-psa-cortex-m-breakage/, 2018.

[54] F. Vandenberghe, “Compiling wamr-sys for thumbv6m-none-eabi fails,” https://github.com/bytecodealliance/wamr-rust-sdk/issues/30, 2024.

[55] B. Heisler, “criterion.rs,” https://github.com/bheisler/criterion.rs/.

[56] N. Nethercote, “dhat-rs,” https://github.com/nnethercote/dhat-rs.

[57] flamegraph rs, “flamegraph,” https://github.com/flamegraph-rs/flamegraph.