Advancing the I2C proposal for WebAssembly System Interface

Friedrich Vandenberghe dr. ing. Merlijn Sebrechts dr. ing. Tom Goethals

prof. dr. ir. Filip De Turck prof. dr. Bruno Volckaert

***Abstract*— W3C WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fulfil the need for technology that is secure, reliable and easily updatable, while requiring only a minimal overhead. But currently, there aren't any interfaces yet for the I2C protocol. This paper provides the `wasi-i2c` proposal, which is in the second phase of the WASI standardization process. The provided implementations ascertain the feasibility of `wasi-i2c`, and also showcase a negligible overhead. Except for the memory usage of the Wasmtime runtime.**

***Index terms*—WebAssembly, I2C, WASI, Embedded devices**

I. Introduction

There are countless software solutions running on critical infrastructure. Additionally, consumer products ranging from baby-monitors to smart-watches are omnipresent in everyone's life. That's why the European Parliament has approved the Cyber Resilience Act on the twelfth of March 2024 [1]. This Act will obligate manufacturer's to ensure, among others, that for the duration of the support period, vulnerabilities are handled effectively. Furthermore, security updates need to be made available to users for the time the product is expected to be in use [2]. Besides, in the automotive industry, there's a rapidle growing practice of wirelessly distributing software updates to vehicles, called over-the-air [3]. 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 frequently tightly integrated with IoT devices with limited hardware capabilities. This augments the requirements with the demand for a minimal overhead.

W3C WebAssembly (Wasm), and WebAssembly System Interface (WASI), is the technology that aims to fullfill all these requirements. Created to execute binary code 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, there aren't any interfaces yet for the I2C protool. This protocol defines a serial communication bus widely used in the Internet of Things (IoT) ecosystem.

The goal of this paper is to facilitate the I2C connection between Wasm applications and its underlying hardware and sensors, while still adhering to the requirements. Furthermore, this method is also standardized. 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.

Section II provides an overview of the standardization effort. Section III demonstrates implementations that make use of I2C as part of Wasm. Following up, Section IV gives an in-depth review of the overhead.

II. Architecture and standard

At the start, the I2C proposal [4] was merely an idea, lacking any proposal or implementation, and thus still residing in phase zero. But now it is in the second phase, with ongoing effort to fullfill the criteria to pass the vote to the third phase. The following requirements are yet to be met:

- The proposal lacks a test suite that covers the feature.
- Some pull requests and issues are still open that iterate on the design of the feature.
- Updates to the reference interpreter are not yet required, but recommended.

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.

The proposal contains three Wasm Interface Type (WIT) files that follow the component model: `delay`, `i2c` and `world`. The first two closely follow the corresponding interfaces from

`embedded-hal` version 1, but this crate is not necessary for an implementation. They also define two handles, which provide pretty broad access. A more fine-grained control model is possible, but there's currently no demand for this.

A. Asynchronous I2C

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

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

B. Portability criteria

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

Table 1: `wasi-i2c` capability criteria

The portability criteria show that the proposal isn't overspecified for a particular platform. 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 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 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.

III. Implementations

The implementations ascertain the soundness of the WIT interfaces from `wasi:i2c` [5]. Inside I2C there are three basic types of methods, from which two are implemented, write and read. For writing, the implementations use a 4-digit 7-segment display, and for reading a HTS221 sensor, from which the current temperature and humidity can be read.

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 2: Concise overview of the setups

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. 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 and a Raspberry Pi Pico. Both the Pi 3 and the Pi 4 run on an ARM64 architecture and a Linux platform, contrarily, the Pico is a microcontroller targeting RP2040. An overview can be found in Table 2. The used platform and architecture combinations are different from the portability criteria because these were decided before the criteria were decided upon.

A. Native implementations

The Pi 4 can build the codebase, but the other two require cross-compilation. In Rust, this is performed with the `target` flag. For ARM64 this is `aarch64-unknown-linux-gnu`, and for RP2040 it is `thumbv6m-none-eabi`. The Pico has some further requirements. First, it only supports `no_std` builds. Second, the memory layout needs to be specified inside a file called `memory.x`.

To easily interface with the hts221 sensor, the `hts221` Rust crate can be used. For `embedded-hal` version 1 integration, a fork had [6] to be created. Furthermore, to have an I2C connection, the implementation uses `linux-embedded-hal`. For the implementation running on the Pi 4 we write the current local time. In this case, the I2C connection uses `rppal`. This library also follows the `embedded-hal` API, but is specifically tailored for Raspberry Pi devices. On the Pico, a simple incrementing counter is instead displayed. For this, it uses the `rp-pico` crate.

B. Implementations inside WebAssembly

To make the move from native to WebAssembly, everything related to the I2C connection itself is kept host-side. The device-specific logic, is moved to the guest. Ideally, each implementation, conceptually, comes down to the scheme of a guest component and a host runtime glued together via a WIT world. But due to the myriad of differences between Wasmtime and WAMR, no WebAssembly-specific code can be reused between them. This implementation, therefore, keeps everything each runtime interacts with inside two seperate directories.

1) *Implementations inside 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.

WIT world:

```
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>;
    }
}
```

Listing 1: One of the WIT worlds to which guest and host bind.

Both the guest components, as the Wasmtime host implementation, bind to worlds akin to the one from Listing 1. The HTS221 maps to this world, and `wasi:i2c` is the proposal.

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, an adapter is required. Typicallly this adapter enables the usage of the std, but in this case, an empty adapter is given. Note that this uses `#![feature(...)]`, which requires the usage of the nightly version of Rust. Listing 2 describes how, conceptually, each component implementation looks like.

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, 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 fulfilment of the imports, resulting in lots of code redudancy. Therefore, a more involved host implementation, is more practical. Here, the world implementations can share their usage of the I2C bindings for the fulfilment of the imports via the `with` keyword. It is important that these paths be fully qualified. Otherwise, errors will be thrown by the Rust compiler.

2) *Implementations inside WAMR:*

WebAssembly Micro-Runtime (WAMR) has no support for WASI preview two. This application, 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 here the implementations use the `once_cell` crate.

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

// To make the generated bindings compatible with the
`embedded-hal` API
// Imported from the `wasi-embedded-hal` crate
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);
```

Listing 2: Stripped down version of a guest component for Wasmtime.

```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`.
```

Listing 3: Stripped down version of a guest module for WAMR.

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. Furthermore, the WAMR Rust SDK mandates that imports and exports are called and developed as extern C functions. This leads to a module implementation conceptualized by Listing 3. 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 [7], but WAMR uses the `arm-none-eabi-gcc` linker. This results in a CMAKE error when building the SDK [8], and thus disabling cross-compilation.

IV. Evaluation

A. Evaluation setup

The evaluation setup is based upon the setups used for the implementations, as shown in Section III. 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 setup evaluates the display natively and with Wasmtime and WAMR as a runtime. The same applies for the sensor, but without the WAMR runtime.

B. Functional evaluation

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.

C. Benchmarks

Running in Wasmtime should perform worse due to the imposed overhead on both execution time and memory usage, but this overhead will be negligible. For WAMR, there should be an even slimmer overhead.

First, an inspection is performed on the execution time. An overview is available in Figure 1. This overview showcases an astonishing longer execution time when running inside Wasmtime. Most of this time is spent inside functions 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 execution time. This can be done vai `wasmtime compile` or the `Component::serialize` function inside Wasmtime. To load a precompiled file, the `Component::deserialize_file` function needs to be used. 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.

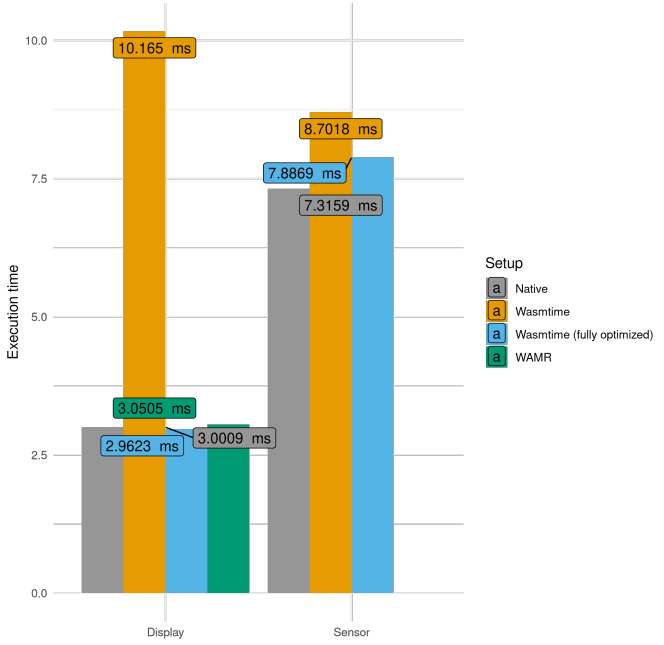

Figure 1: Overview of all the mean execution times

A further optimization can be done through no longer linking the command world. This world isn't necessary because of the custom host setup. Now, the fully optimized Wasmtime build is slightly faster when writing to the display than the native version. This speedup could be due to noise or kernel intrinsics.

Comparing WAMR with the fastest Wasmtime version for writing to the display, 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.

When considering the memory usage, Figure 2 and Figure 3, 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 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.

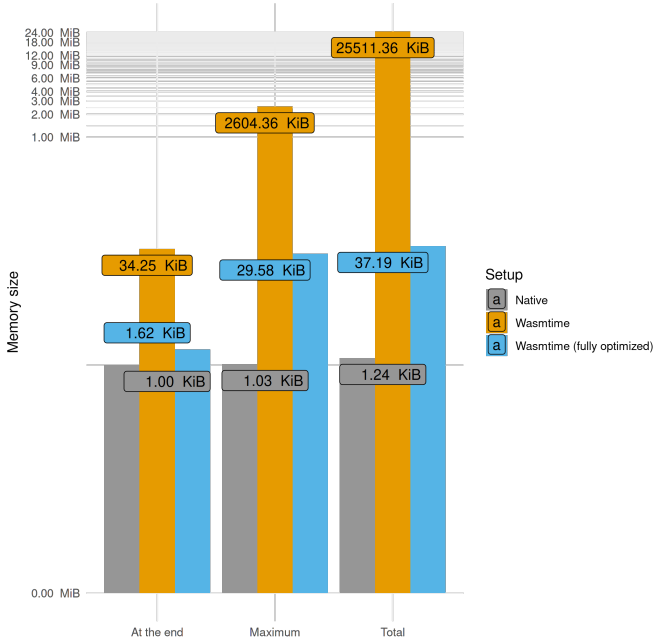

Figure 2: Used memory when reading from the sensor

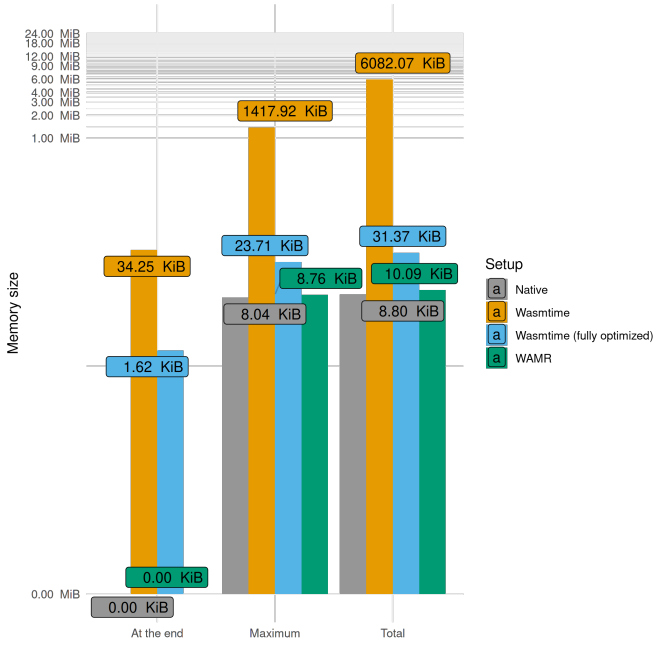

Figure 3: Used memory when writing to the display

V. Conclusion

Thanks to the provided interfaces inside `wasi-i2c`, it is possible for a Wasm component to control a device using I2C. Thus this answers the first research question. To minimize any migration costs from existing applications to WebAssembly, the `wasi-embedded-hal` crate is provided. With this crate, it is possible to easily interoperate with device crates inside the Rust `embedded-hal` ecosystem. Although the proposal is heavily inspired by this ecosystem, it is designed in such a way that no implementation should need to use this API to be successful.

Furthermore, to resolve the second research question, the aforementioned interfaces have been brought from a mere idea to fully-fledged interfaces ready for feedback from the WebAssembly Community Group. Thus, from phase zero to phase two. Two capabilities are provided, one for I2C and one for delays. More fine-grained control over these capabilities is something that's currently lacking, but certainly feasible to achieve.

To address the fourth question, the overhead is defined as a dichotomy of the mean execution time and the peak memory usage. With only an increase of 1.65% in mean execution time, and having an 8.93% higher peak memory usage over the complete lifespan of the application, the overhead of the WAMR runtime can be seen as negligble. For Wasmtime, some considerations apply. First, each WASI interface added into the linker incurs extra overhead. Second and last, before executing the component itself, the runtime first instantiates it. This instantiation can be done a priori but is not guaranteed to work on other machines. For this, the runtime engine should be configured in a peculiar way. This is seen as an open challenge. With these considerations, the mean execution time is −3.86% and peak memory usage is 15.87 times more when compared to native

This indicates that Wasmtime, and by extension the component model, incurs a hefty memory overhead. Efforts are underway to align this overhead more with WAMR's, but this in an early stage. Thus, to answer the fifth research question, in its current state the WASI component model is not suitable to run on devices with constrained resources.

References

[1] B. Chatain, "Cyber Resilience Act: MEPs adopt plans to boost security of digital products," *European Parliament*, Mar. 2024, [Online]. Available: https://www.europarl.europa.eu/news/en/press-room/20240308IPR18991/cyber-resilience-act-meps-adopt-plans-to-boost-security-of-digital-products

[2] E. Commission, "EU Cyber Resilience Act." Dec. 2023.

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

[4] F. Vandenberghe, L. Clark, and J. Schilling, "WASI I2C."

[5] F. Vandenberghe, "i2c-wasm-components."

[6] F. Vandenberghe, "hts221."

[7] E. R. W. Group, "PSA: Cortex-M Breakage (LLD as the default linker)." Aug. 2018.

[8] F. Vandenberghe, "Compiling wamr-sys for thumbv6m-none-eabi fails." Apr. 2024.