Dankwoord

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

Permission for Usage

Abstract - Dutch

Abstract - English

Contents

List of Figures

List of Tables

List of Acronyms

ABI application binary interface. , 10, 11

ACK Acknowledgement. , 2, 3

API Application Programming Interface. , 7, 13, 16

GPIO general-purpose input/output. , 5

HAL Hardware Abstraction Layer. , 5, 6, 16

HAT Hardware Attached on Top. , 15

I2C Inter-Integrated Circuit. , 2, 4, 5, 13–15, 17

IDL Interface Description Language. , 10, 11

MCU Microcontroller Unit. , 2, 6, 15, 16

NACK Negative-acknowledgement. , 3

OS Operating System. , 5, 6, 10

SCL Serial Clock Line. , 2, 4

SDA Serial Data Line. , 2, 3

SIG Special Interest Group. , 13

SMBus System Management Bus. , 4

SPI Serial Peripheral Interface. , 5

SVD System View Description. , 6

SWD Serial Wire Debug. , 6

UART universal asynchronous receiver-transmitter. , 5

VM Virtual Machine. , 7–9, 13

WAMR WebAssembly Micro Runtime. , 8, 16, 17

WASI WebAssembly System Interface. , 7, 8, 10, 13

Wasm WebAssembly. , 7, 8, 10, 12, 13

WIT Wasm Interface Type. , xii, 7, 8, 10, 11, 15–18

List of Code Fragments

1 Introduction

There are countless software solutions running on critical hardware, where sudden failure could result in the loss of people's live, e.g. programs guiding surgeons during operations or cars. Furthermore, in the case of cars, there's a clear trend towards more advanced infotainment systems and to update these remotely without the need to visit an auto mechanic. When such an update fails to perform, it should be possible to rollback unbeknownst to the driver.

In addition, with the advent of generational AI's like ChatGPT, worldwide GPU usage has exploded. Directly translating in an explosion of energy and water usage to run all and cool all this harware inside data-centres. Sadly, these GPUs have long-lasting periods of idly running, waiting for requests. It should be possible to start a GPU when running a request, but otherwise keep it off, without significant overhead in terms of latency.

2

Background

2.1 I2C fundamentals

As we will be leveraging the Inter-Integrated Circuit (I2C) protocol, it is worth looking into the inner workings.

I2C [1] 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.

2.1.1 Operations

A node on the bus can have one of two roles[1]:

- 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)

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

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.

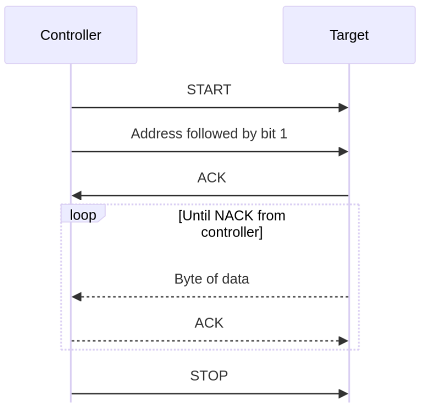

Figure 2.1: Example sequence of a read operation.

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 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.

2.1.2 SMBus

The System Management Bus (SMBus) [2] 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.

2.2 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 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 Operating System (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 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.

2.2.1 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 [3]. 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 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.

2.2.2 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 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.

2.2.3 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. This also unlocks the ability to print to STDOUT or even utilize the Debug Adapter Protocol.

2.3 WebAssembly

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 or Google Earth.

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. All possible through the usage of a system interface, called WebAssembly System Interface (WASI).

2.3.1 WebAssembly System Interface

WASI [4] is a modular collection of Application Programming Interface (API)'s defined with the WIT IDL, see section 2.4.1, that provide 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 WebAssembly Community Group.

In this subgroup, the following design principles are core:

- Capability-based security: All access to external resources is provided by capabilities, see chapter 2.4.
- 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 2.4.

Versions

Originally WASI launched under the name: `WebAssembly System Interface, snapshot 1`, nowadays called `0.1` or `Preview 1`. On the twenty-fifth of January 2024, `Preview 2` was launched. Following up will be a `Preview 3`, and then the `1.0` release.

The flagship feature that `0.2` brought to the table is the component model, see chapter 2.4. Furthermore, two WIT worls, see section 2.4.1, 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.

2.3.2 Community

As mentioned in section 2.3.1, standardization is performed under the supervision of the WASI Subgroup. As a part of W3C's WebAssembly Community Group, it is the key player in the standardization process. With respect to the implementations, this is the Bytecode Alliance, a nonprofit organization of companies. Not every company wishes to part of these bodies, e.g. Wasmer, [zijn er nog goede voorbeelden?].

2.3.3 Runtimes

A runtime system is an infrastructure that participates in the creation and running of Wasm binaries. Typically, the components of this are the execution environment, or application VM, to provide a place for the program to run, the compiler front-end and/or the compiler back-end for the necessary analysis, transformations and optimization.

The Bytecode Alliance fosters two runtimes, Wasmtime and WAMR. The former can be seen as a more general-purpose runtime, focussing on server-side and non-web embeddings with components. Making it the de-facto runtime. While WebAssembly Micro Runtime (WAMR) specifically is 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. For example, Wasmtime has full component model support, while WAMR has it planned for end of 2024. With regards to the supported languages, WAMR only supports C/C++. A toolkit for Rust has been published in March 2024. On the other hand, Wasmtime has first-class support for eight languages, and community support for a further two. Furthermore, there's active effort in making this runtime lighter to run. One such effort is the inclusion of a Rust `no_std` option.

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, specialized for JavaScript, componentize.py, for Python, or Chicory, which runs on the Java VM.

2.4 The Component Model

The WebAssembly Component Model is an architecture for building interoperable Wasm liraries, applications and environments. These components can be seen as containers for modules, or other components, which express their interfaces and dependencies via WIT and the canonical application binary interface (ABI). 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.

It is important to note, though, that the Component Model is not WASI (`Preview 2`), nor a part of it. By way of comparison to 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.

It is easiest to think of the Component Model as a guest-host architecture, but it is certainly possible to compose multiple components together. For this, cargo-component can be used in Rust, or the language-agnostic wasm-tools.

2.4.1 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.

There are also some small intricacies worth pointing out. For types, two things stand 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.
Furthermore, functions can only be declared as part of an interface, or as an import or exprot 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.

To make managing dependencies inside your WIT definition easier, the wit-deps 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 2.3, 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.

2.4.2 Guest

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 [5].

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

2.4.3 Host

The job of the host is to load a component and execute it through the usage of a Wasm runtime. See section 2.3.3 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 2.4.1. When using `wit-bindgen`, this is done via the `with` option inside the the `bindgen` macro.

It is also possible to execute a component via the `wasmtime run` subcommand. This will compile the module to native code, instantiate it and optionally execute an export.

2.4.4 Problems

[Hier kan er misschien iets verteld worden over de argumenten om component model niet te gebruiken? Is dit wel nuttig en willen we dat wel?]

3

Architecture and standard

As stated previously in section 2.3.1, WASI is a collection of API's. In order for I2C support inside WASI, an API should thus be defined and standardized. The API is publicly available inside the wasi-i2c repository.

Software developed without any reviews, nor feedback, is software that is doomed to fail at some point. The same principle applies to the standardization of a proposal. For feedback, the input from the Wasm community, see section 2.3.2, is invaluable. Besides this community, 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.

3.1 The proposal process

Stated in section 2.3.1, WASI is under the goverance of the WASI Subgroup. This 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.

It is the convention that a proposal has the `wasi-` prefix. This is the reason the I2C proposal is called `wasi-i2c`.

3.1.1 Current `wasi:i2c` phase

Currently, the I2C proposal is in the first phase, with ongoing effort to fullfill the criteria to pass the vote to the second phase. Specifically, a broad enough of a consensus needs to be reached on the capability criteria. This effort is led under the guidance of certain champions. For `wasi:i2c` 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.

3.2 Alternatives to Wasm

4 Implementations

As stated in chapter 3, it is necessary to provide implementations to ascertain the soundness of the WIT interfaces. Ideally, these are as diverse as possible. Both in terms of operations on the I2C connection, and the target architectures.

Three implementations are provided: one that performs an I2C read and two that write to an I2C connection, developed for three devices targeting two architectures. On the one hand, we have a Raspberry Pi 3 and 4 targeting ARM64 Linux, and on the other hand we have a Raspberry Pi Pico MCU targeting RP2040 processor.

On the Pi 3, a Hardware Attached on Top (HAT) is mounted that contains a HTS221 sensor, from which the current temperature and humidity is read. The Pi 4 is either connected with a HD44780 LCD character display or a 4-digit 7-segment display. Although the Pi 3 could also be linked up with these displays, it's more of a hassle thanks to the HAT. The pico is solely linked with the 4-digit display. Because of the MCU constraints, controlling the HD44780 is out of scope.

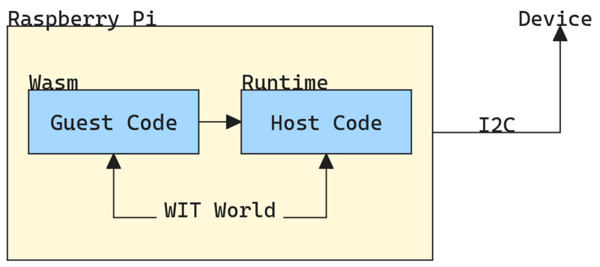

Figure 4.1: Schematic overview of an implementation.

Conceptually, each implementation comes down to the schematic defined in figure 4.1. Only the guest code differs for each device. See chapter 2.3 for an in-depth explanation.

4.1 Embedded driver development in Rust

Besides the implementation itself, two things are of significance to design a device driver in Rust. Namely, following the Embedded HAL API, see chapter 2.2, and supporting the desired target architecture.

Fortunately, Rust provides support for a great deal of platforms. Thus, building for ARM64 is as simple as providing the `--target aarch64-unknown-linux-gnu` flag. However, the RP2040 architecture needs special attention. For this, besides setting the `--target` flag with `thumbv6m-none-eabi`, we also need to provide a file called `memory.x`. This file is a linker script which specifies the memory layout of the target device, see section 2.2.3. Furthermore, the MCU has only support for a `no_std`.

4.2 Embedded driver development in WebAssembly

Besides the considerations from the previous section, we now also need to keep the used runtime target platform support in mind. See section 2.3.3 for an overview. Furthermore, in this section, the different features of each runtime are also highlighted. These led to a vastly different implementation for both the host and the guest code.

4.2.1 Wasmtime

The guest is made into a component via the procedure specified in section 2.4.2. The configured WIT interface is the one specified in codefragment 4.1. Herein `wasi:i2c` is the proposal, as explained in chapter 3. Both displays are 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 generated bindings have no way of knowing that they actually should follow the `embedded-hal` traits, for this the wasi-embedded-hal crate is used.

On the side of the host, binding is done as described in section 2.4.3. Here, `wasi:i2c` is seen as a missing import, thus the `with` option is used with a data structure that implements the necessary traits.

4.2.2 WAMR

As explained in section 2.3.3, WAMR is a lightweight runtime that, currently, has no support for preview 2. To sustain a pure Rust codebase, WAMR Rust SDK is used. This SDK provides Rust language bindings and support for passing integers and floats between the host and the guest. Note that strings can be passed via a conversion to a vector of the string code points.

As we can no longer use WIT, nor pass a connection to the guest, we are enforced to greatly differ from the Wasmtime implementation. Therefore, we will focus us on the simpler 4-digit 7-segment display with the following conceptuel differences for WAMR: The I2C connection is kept global inside the host and there are now 4 arguments for the `write` function, one for each digit.

```
package sketch:implementation;

interface hts {
    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>;
}

interface lcd {
    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);
}

world sensor {
    import wasi:i2c/i2c@0.2.0-draft;

    export hts;
}

world screen {
    include wasi:i2c/imports@0.2.0-draft;

    export lcd;
}
```

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

5

Evaluation

Conclusie

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

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

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

[4] 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,” Dec. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4323447

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

Appendices

Bijlage A

Toelichting bijlage.

Bijlage B

Toelichting bijlage.