FACULTY OF SCIENCES

Standardizing USB in the WebAssembly System Interface

Wouter Hennen
Student number: 01905957

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: Dr. ing. Merlijn Sebrechts, ing. Michiel Van Kenhove, Dr. ing. Tom Goethals

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science

Academic year 2023-2024

GHENT UNIVERSITY

Abstract

Updating software of Internet of Things (IoT) devices is currently difficult. Having a longer lifespan than the average gadget, and oftentimes having special hardware and compilers that requires a lot of maintenance, most IoT devices lose software support too soon [1]. WebAssembly (Wasm), once made to run compiled code in a lightweight virtual machine in the browser, is trying to tackle this problem by expanding its support outside the browser. Because of its near-native performance, security, compactness and language interoperability, Wasm is gaining popularity for its use on edge devices [1]. However, Wasm lacks a proper interface to use outside the browser. To tackle this issue the WebAssembly System Interface (WASI) was introduced, with the goal to provide a universal API for all supported platforms and languages. WASI is still in development and currently lacks an API for using USB devices. The goal of this thesis is to research and develop an USB API for WASI.

As part of the development is creating a proposal to extend the standard, where the interface of the API is laid out and other matter, such as platform support, is further discussed. As part of this thesis, an initial proposal has been created [2]. The proposal has advanced to Phase 1 of the proposal process [3] and progress is being made to progress to Phase 2. The proposal contains a WIT interface which supports enumerating devices, reading descriptors, connecting, reading and writing to devices and getting device connection updates. After doing research on existing similar solutions, such as WebUSB [4] and libusb [5], the proposal has taken inspiration from libusb. This makes it easier for developers to port existing applications to Wasm as both libusb and WASI USB have similar interfaces. Some parts have been made different deliberately, for example the way file handles work. This way, the API is easier to reason about. As Wasm is sandboxed by default, access control is provided so programs do not have access to all devices by default. It is up to the user to specify which devices a program can access.

A reference implementation for the API is provided [2] and extends the Wasmtime [6] runtime to provide support for the USB API.

The implementation has been evaluated in three ways. First, a functional evaluation was made. In this evaluation, a guest module will observe connection events for a specific game controller, connect to the game controller, read out its state and control its rumble motors, confirming that the most important parts of the API work.

Afterwards, the implementation has been evaluated for latency and memory usage. This is done by creating a USB mass storage device driver by using the WASI USB API. The file tree of a USB drive can be read out and the file contents can be read. Comparing this against a native implementation shows on average a 4.2% increase in latency, which is a good result. Memory usage using the WASI API is significantly higher, with the program using approx. 70% more memory during heavy workloads, compared to native. This is likely caused by the way the memory model works in the Component Model, requiring the use of copying of data between components.

The WASI USB API still has a long way to go before it is production ready, but the example programs in this thesis show the possibilities of the API and demonstrate how the API can be used in real-world use cases.

Samenvatting

Het bijwerken van software voor Internet of Things (IoT)-apparaten is momenteel moeilijk. Met een langere levensduur dan het gemiddelde gadget, en vaak speciale hardware en compilers die veel onderhoud vereisen, verliezen de meeste IoT-apparaten te snel softwareondersteuning [1]. WebAssembly (Wasm), oorspronkelijk gemaakt om gecompileerde code te laten draaien in een lichtgewicht virtuele machine in de browser, probeert dit probleem aan te pakken door zijn ondersteuning buiten de browser uit te breiden. Vanwege zijn quasi-native prestaties, veiligheid, compactheid en programmeertaal onafhankelijkheid, wint Wasm aan populariteit voor gebruik op edge-apparaten [1]. Echter, Wasm mist een goede interface voor gebruik buiten de browser. Om dit probleem aan te pakken werd de WebAssembly System Interface (WASI) geïntroduceerd, met als doel een universele API te bieden voor alle ondersteunde platforms en talen. WASI is nog in ontwikkeling en mist momenteel een API voor het gebruik van USB-apparaten. Het doel van deze thesis is om onderzoek te doen naar en een USB API te ontwikkelen voor WASI.

Een deel van de ontwikkeling omvat het maken van een *proposal* om de standaard uit te breiden, waarin de interface van de API wordt uiteengezet en andere zaken, zoals platformondersteuning, verder worden besproken. Als onderdeel van deze thesis is een initiëel *proposal* gemaakt [2]. Het *proposal* is gevorderd naar Fase 1 van het doorloopproces [3] en er worden stappen ondernomen om door te gaan naar Fase 2. Het voorstel bevat een WIT interface die het mogelijk maakt een lijst van verbonden apparaten op te vragen, *descriptors* uit te lezen, verbinding te maken, te lezen en te schrijven naar apparaten en updates over apparaatverbindingen te krijgen. Na onderzoek naar bestaande vergelijkbare oplossingen, zoals WebUSB [4] en libusb [5], heeft het voorstel inspiratie gehaald uit libusb. Dit maakt het voor ontwikkelaars gemakkelijker om bestaande applicaties naar Wasm te porteren aangezien zowel libusb als WASI USB vergelijkbare interfaces hebben. Sommige delen zijn opzettelijk anders gemaakt, bijvoorbeeld de manier waarop *device handles* werken. Op deze manier is de API gemakkelijker te begrijpen. Aangezien Wasm standaard in een sandbox draait, wordt *access control* gebruikt zodat programma's niet standaard toegang hebben tot alle apparaten. Het is aan de gebruiker om te specificeren welke apparaten een programma kan gebruiken.

Een referentie-implementatie voor de API wordt geleverd [2] en breidt de Wasmtime [6] runtime uit om ondersteuning te bieden voor de USB API.

De implementatie is op drie manieren geëvalueerd. Ten eerste werd een functionele evaluatie uitgevoerd. In deze evaluatie observeert een gastmodule verbindingsevenementen voor een specifieke gamecontroller, maakt verbinding met de gamecontroller, leest de status ervan uit en bestuurt de trilmotoren, waarmee wordt bevestigd dat de belangrijkste onderdelen van de API werken.

Vervolgens is de implementatie geëvalueerd op vertraging en geheugengebruik. Dit werd gedaan door een USB mass storage apparaatstuurprogramma te creëren met behulp van de WASI USB API. De bestandsstructuur van een USB-schijf kan worden uitgelezen en de inhoud van bestanden kan worden gelezen. Een vergelijking met een native implementatie toont gemiddeld een vertraging van 4,2%, wat een goed resultaat is. Het geheugengebruik van de WASI API is significant hoger, waarbij het programma ongeveer 70% meer geheugen gebruikt tijdens zware werkbelasting, vergeleken met een native programma. Dit wordt waarschijnlijk veroorzaakt door de manier waarop het geheugenmodel werkt in het Component Model, dat vereist dat gegevens worden gekopieerd tussen componenten.

De WASI USB API heeft nog een lange weg te gaan voordat het klaar is om gebruikt te worden, maar de voorbeeldprogramma's in deze thesis laten de mogelijkheden van de API zien en demonstreren hoe de API kan worden gebruikt in echte gebruikersscenario's.

Standardizing USB in the WebAssembly System Interface

Wouter Hennen
Supervisors: prof. dr. Bruno Volckaert, prof. dr. ing. Filip De Turck
Consellors: dr. ing. Merlijn Sebrechts, ing. Michiel Van Kenhove

Abstract—This thesis explores the addition of USB device support to the WebAssembly System Interface (WASI). The aim is to enable developers to communicate with USB devices in a platform-independent and language-independent manner, similar to the current capabilities provided by WebUSB and WebAssembly on browsers. By leveraging the built-in access control mechanisms of WASI, this can be achieved securely, allowing users to manage USB device access efficiently. This paper details the architecture, proposal, implementation, and evaluation of this integration, demonstrating its feasibility and performance.

Index Terms—WebAssembly, WebAssembly System Interface, Component Model, Wasm, WASI, USB, access control

I. Introduction

In Web browsers, Wasm is used to run compiled code in a lightweight virtual machine. Code gets compiled to Wasm, but can be run on any platform that supports Wasm, making it cross-platform. This model can also be applied outside the browser context and can solve some of the problems of native code. As Wasm is primarily targeted to web browsers it only contains APIs to interact with browsers. If Wasm code needs to run outside the browser, new APIs are needed. To solve this, a new specification was created: the WebAssembly System Interface (WASI). WASI is a collection of API specifications that can be used by Wasm applications outside the browser. Thanks to WIT and the canonical ABI, these APIs can provide a more developer-friendly interface compared to Wasm. These APIs can still be used by any programming language that offers support for WASI. Through the use of generated bindings, the API defined in WIT can be converted to a language-tailored variant, integrating well with features the language offers. For example, languages with a different representation for strings can still communicate together, because their shared representation, the canonical ABI, is the same. Because the APIs act as a layer between the native APIs and the application, access control can be applied to control what an application can access.

This thesis extends the WebAssembly System Interface (WASI) to include a standardized USB API, facilitating secure and efficient communication with USB devices across various platforms. The USB API leverages the access control features provided by WASI to secure the access to USB devices. Special attention will also be given to the portability and performance of the API, as these are key to the success of the API.

II. Proposal

The wasi-usb repository [1] contains the contents of the proposal to standardize the USB interface. The proposal contains a draft of the WIT interface of the USB API. This section will highlight the important parts of the API. The wasi-usb proposal is at the proposal phase 1 at the time of writing.

The proposal contains the following features:

- Enumerate USB devices.
- Read descriptors from devices.
- Open device handles to devices.
- Read data from devices.
- Write data to devices.
- Get notified when a device connects.
- Get notified when a device disconnects.

Figure 1 shows a general overview of the different parts of a program using the USB API. The implementation is contained in the 'WASI USB' block, this is what the proposal is about.

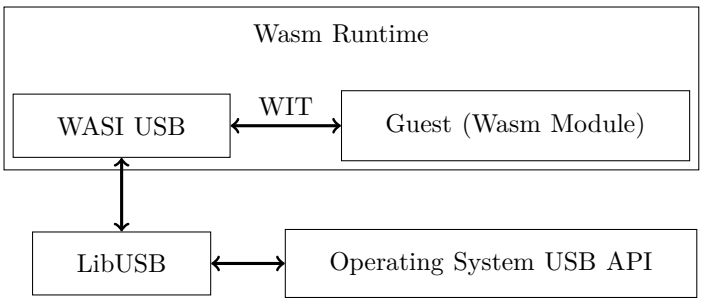

Fig. 1. Internals of the WASI USB API. The arrows represent communication between the different areas.

A. WIT interface

The proposal contains four WIT interfaces: usb, events, descriptors and types. The most important one, usb, will be discussed here. It is shown in Figure 2, some details are omitted for brevity.

The interface contains two resources: usb-device and device-handle. The usb-device resource contains a static

function enumerate, with which all available USB devices can be retrieved. These will be returned as instances of usb-device. The other three methods can be used on one of these instances. The device-descriptor and configurations functions will both return extra information about a device and are always available. That is, they do not need to have exclusive access to these devices to obtain that information. The open function is different in this regard. This function will establish a context with which you can communicate with the device. One important aspect is that opening a device can fail due to various reasons. For example, some operating systems might deny opening a device if the executable does not have enough permissions. These permissions are handled at a Wasm runtime level, and are not specific to Wasm modules. To create a robust and intuitive API, functions that require an 'open' device are scoped to the device-handle resource, which can only be obtained if a device can be opened successfully.

The device-handle resource will contain all the functions which require communicating with a device. For example, you are able to fetch the active configuration of a device. Functions exist to read and write using the four transfer types: read-interrupt, write-interrupt, etc. Most functions in device-handle are closely related to similar LibUSB functions [2].

B. Access Control

The USB proposal specifies access control at a device level. It does this by filtering out non-allowed devices for usb.{usb-device/enumerate}, which gives a list of all available devices, and events.update, which gives updates of device connection events. A guest module can only receive a usb-device resource from these two functions, so if they do not return that resource, the guest cannot use it.

Access control can also be specified at a configuration, interface and endpoint level, but these have been left out for now, as it is not known yet if they would provide any benefits.

C. Usage in multiple components

It is possible that multiple components interact with the USB API and try accessing a USB device at the same time. This should be avoided, as a component generally expects to have exclusive access to a device. To solve this issue, the runtime should keep track of which devices are used by which components, and deny opening a device when another component is already using it. This approach follows the same model as most operating systems, where only one program can have access to a USB device at a time.

III. Implementation

This section dives deeper into the internals of the implementation. First, the runtime implementation will be discussed. Next, using the API in a guest component will be explained.

```
package component:usb@0.2.0;

interface usb {
    use types.{usb-error};
    use descriptors.{configuration-descriptor,
        device-descriptor};

    type duration = u64;

    resource usb-device {
        configurations: func() -> ...;

        device-descriptor: func() -> device-
            descriptor;

        open: func() -> result<device-handle, ...>;

        enumerate: static func() ->list<usb-device>;
    }

    resource device-handle {
        reset: func() -> result<_, usb-error>;
        active-configuration: func() -> ...;

        select-configuration: func(configuration: u8
            ) -> ...;

        claim-interface: func(%interface: u8) ->
            ...;

        release-interface: func(%interface: u8);

        select-alternate-interface: func(%interface:
             u8, setting: u8) -> ...;

        read-interrupt: func(...) -> ...;
        write-interrupt: func(...) -> ...;

        read-bulk: func(...) -> ...;
        write-bulk: func(...) -> ...;

        read-isochronous: func(...) -> ...;
        write-isochronous: func(...) -> ...;

        read-control: func(...) -> ...;
        write-control: func(...) -> ...;
    }
}
```

Fig. 2. USB interface.

The wasi-usb [1] repository contains the implementation explained in this section, as well as a number of guest components utilizing it.

A. Runtime implementation

The implementation is written in Rust [3]. On the native side, LibUSB [4] is used to call OS-specific APIs. The Rusb [5] Rust package is used and adds a thin wrapper around the LibUSB API, making it easier to use in Rust. The implementation extends the Wasmtime [6] Wasm runtime and will embed Wasmtime in its compiled code. When running the program, three parameters can be passed in:

- Guest component path (required): The path to the guest component that will be executed. The component must be a command component which contains a run function.
- '–usb-devices': A list of devices that the guest component is allowed to access. The devices are specified in the vendor_id:product_id format.
- '–usb-use-denylist': Adding this flag will change the devices list from an allowlist to a denylist.

By default, the guest cannot access any devices. Table I shows the possible combinations for device access.

	Devices Specified	No devices specified
Allowlist	specified devices allowed	no device allowed
Denylist	all but specified allowed	all allowed

TABLE I
Device access control options.

B. Guest usage

In order to make use of the USB API, a Guest first needs to import the interfaces in its WIT interface, as shown in Figure 3. This component is a Command component, and therefore does not need to provide any exports. Based on this definition, a bindings generator can provide APIs that feel natural for the language in which the guest is written. The generated bindings will then convert the easy-to-use API to a representation that conforms to the canonical ABI [7].

```
package component:usb-component-wasi-stadia;

world root {
    import component:usb/types@0.2.0;
    import component:usb/usb@0.2.0;
    import component:usb/events@0.2.0;
    import component:usb/descriptors@0.2.0;
}
```

Fig. 3. The WIT world for the guest component.

IV. Evaluation

The API has been evaluated in three ways. First, a functional evaluation was performed, checking if the API behaves as expected. Next, a latency evaluation was done, measuring if there are any noticeable slowdowns while using the API compared to native code. Finally, a memory evaluation was constructed, measuring the impact on memory usage of running the code in Wasm compared to native.

The code of all the benchmarks can be found in the USB_WASI [8] repository.

A. Functional evaluation

The functional evaluation touches on all parts of the API: Getting device connection and disconnection events, reading descriptors from a device, opening a device handle, reading data from the device and writing data to the device.

```
dpad: ,
buttons: assistant_button|l2_button|r2_button,
left stick: x: 128 y: 128,
right stick: x: 128 y: 128,
l2: 255,
r2: 172
```

Fig. 4. Output after reading controller state.

The general idea of the program is to control a game controller. The program is started and observes the connected devices. Once a controller is connected that is recognized by the program, the program will connect to the controller. The state of all the controls of the device will be read, and input updates will be print out. To test sending data over the USB interface, the program will send commands to the controller to activate the rumble motors. If the program receives a disconnection event, it will stop reading data and become idle again, waiting for new connections.

This program has been tested with a Google Stadia controller. When pressing one or multiple buttons, the correct state is printed out. When pressing one or both of the shoulder buttons, the controller starts to vibrate. Figure 4 shows a sample output with the controller state.

B. Latency evaluation

Latency overhead is tested with two benchmarks: a synthetic benchmark, where an Arduino will constantly send data, and a real-world benchmark, where files will be read off of a usb mass storage device.

1) Reading data from Arduino: The benchmark will read 1 million packets of 64 bytes over the bulk interface from the arduino. In total, 64MB of data will be read. The data will be read sequentially, without delays. To start receiving the data, the program will first send the Device Terminal Ready signal to the Arduino. Once the Arduino receives this signal, it will start sending the data. The Arduino does as little processing as possible, sending a hardcoded array of bytes. The host receives the data and does but process it, but throws it away. This is to remove as many extra latency factors as possible.

Figure 5 shows a boxplot of the latencies of both programs. After a million measurements for each program, the median duration is exactly the same, 377us. The average of the native program is marginally lower, being 0.2% faster than the WASI program. The whiskers and IQR of the WASI program are slightly smaller, meaning a more consistent result. However, the differences in results are too small to conclude any

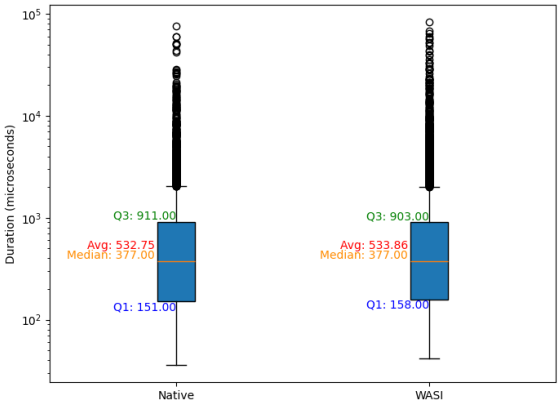

Fig. 5. Boxplot of latency of reading data from Arduino using read-bulk.

noticeable overhead.

Figure 5 also contains outliers, marked by the black circles. Some of the outliers are very large. Therefore, the results are shown on a logarithmic scale. These outliers are likely caused by transmission errors when receiving data from the Arduino. The Bulk interface is used, providing data integrity but no timing guarantees. Consequently, data loss will be prevented, but will introduce high latencies as observed here. Table II shows more information about the outliers. As the percentage of outliers is very small and occur in equal quantities in both cases, they have been excluded from further figures to improve the clarity of the graphs.

Figure 6 shows a Kernel Density Estimation for the measurements of the native and WASI implementation. Both graphs will show peaks around the 150us and 910us marks. This is an interesting result, as one would expect one uniform distribution instead of two. The first peak is trivial to explain: the Arduino is an USB 2.0 device, also known as a High-speed USB device. A High-speed USB device will send frames at a fixed interval of 125us. Therefore, we will receive new data approx. every 125us. An extra 25us are introduced because of processing delays.

The second peak is more nuanced. Table III shows a snippet of the measured latencies. For both Native and WASI code, the latencies will oscillate between both peaks. Based on these results, this peak is likely caused by the small buffer size in the Arduino. The buffer is 64 bytes large, which is the same size as one packet. When the buffer is full, the program halts until the buffer has space again. On the host this is perceived as an extra delay when receiving the data.

Based on the results, we can conclude that there are

	Native	WASI
Largest Outlier (us)	75618	83052
Amount of Outliers (>2000 us)	747	1058
Percentage of Total	0.07%	0.10%

TABLE II
Comparison of Latency Outliers between Native and WASI.

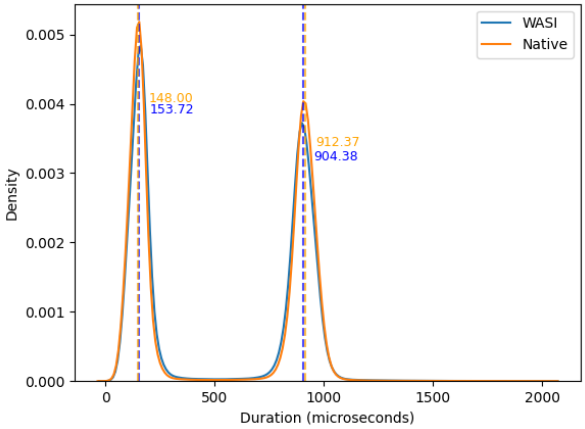

Fig. 6. KDE of latency of reading data from Arduino using read-bulk.

Native (us)	WASI (us)
801	918
150	163
885	931
132	241
903	799
110	197
938	820
104	77
962	1003
108	97

TABLE III
Snippet of measured latencies. The latencies oscillate between a long latency and a short latency.

no measurable differences between the native and WASI program. However, this is mainly caused by the delay of the USB protocol. This delay vastly outweighs the delays caused by overhead of the Wasm runtime, making the Wasm overhead negligible.

However, it is possible that the delay of the USB protocol is smaller on devices more powerful than an Arduino or with a more modern USB version. With these configurations, it can be possible a small overhead for Wasm becomes visible.

2) Reading files from Mass Storage device: In order to confirm that WASI does not add noticeable latency, another benchmark is performed, which reads the contents of an USB mass storage device. The advantage of this benchmark compared to the Arduino benchmark is that it better represents real-world usage, instead of being a

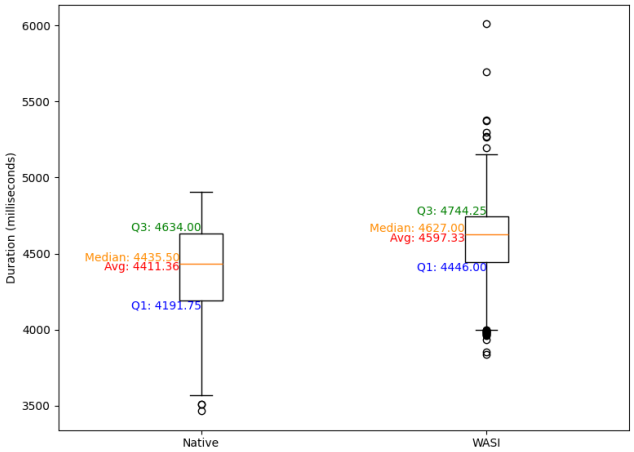

Fig. 7. Latency of reading the file tree and contents after changing the benchmark method. On average, the WASI USB implementation is approx. 4.2% slower.

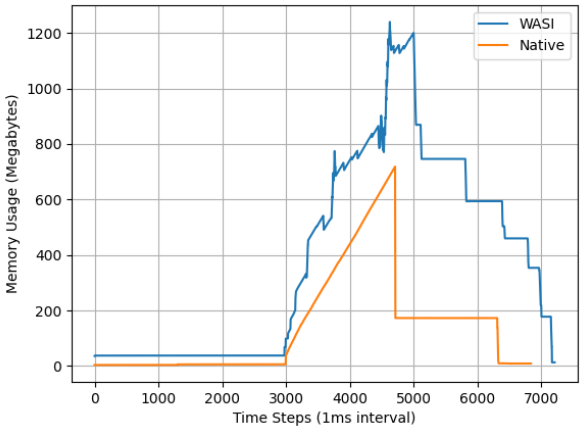

Fig. 8. Memory usage when traversing the file tree and reading the file contents. At the end of reading the largest file (678MB), the WASI program will consume 72.5% more memory than the native counterpart.

synthetic benchmark.

A program has been written which enumerates the file tree of a USB device and reads the contents of each file in the file tree. A Samsung T5 External SSD is used to perform these tests. The device is connected with a cable that supports a transfer speed up to 5Gbps and is formatted with the MBR partition map and the exFAT file system. There are 10 files stored on the device, most of which are a few KBs in size. One file has a size of 679MB. In total, 680MB is stored on the device. The benchmark is executed 1000 times.

Figure 7 shows a boxplot of the measured total times to read the file tree. The average WASI program execution time is 4.2% slower than the average native program, and the median 4.3% slower. The WASI program also has more outliers, indicating that the performance of the WASI program may be slightly less consistent.

Based on these results, we can conclude that running the code using the WASI USB API introduces a small amount of overhead.

C. Memory evaluation

In contrast to standard Wasm modules, WASI components have separate memory spaces. As a result, one component cannot read or write the memory of another component [9]. Therefore, it is interesting to see how much impact WASI has on the memory usage of a typical program.

1) Reading files from mass storage device: The same test methodology as Section IV-B2 will be used to benchmark memory usage. Also, instead of running the benchmark 1000 times, it is now run once. The program will also sleep three seconds before and after opening and closing the device, so the memory usage evolution can be better seen.

Figure 8 shows the differences in memory usage for both programs. As both programs consume a lot of memory, the initial Wasmtime memory usage is neglected, as it does not have a large influence on the results. The influence of initial Wasmtime memory is discussed in Section IV-C2.

After the initial three seconds of waiting time, both programs start traversing the file tree and reading the contents. It is clear that the WASI program uses more memory and with more spikes. As memory is not shared, values are copied over from host to guest, increasing the memory usage. At its peak, it will use 1239MB, while the native program uses 718MB. This is 72.5% more. This peak happens when reading in the largest file of 679MB. As the disk is mostly empty, data is stored contiguously on the disk. This improves reading performance and ensures a steady flow of data at the receiver's end. This is clearly visible in the graph for the native program, as its memory usage follows a linear function. The WASI implementation contains more spikes. These spikes are likely caused by the copying of values, which rapidly allocates and deallocates memory.

After the contents have been read, memory usage will quickly drop for both programs. However, memory usage for the WASI program will decrease slower than the native program. This mainly happens because of the memory-pool-like behavior of the Store [10], which will not deallocate memory. Memory used outside the Store will still get deallocated, which is why there is a memory decrease. Once the program ends the memory will get freed further.

2) Traversing file tree from mass storage device: A less memory-intensive benchmark was also performed to check how WASI memory usage would compare when using less memory. This benchmark uses the same test methodology

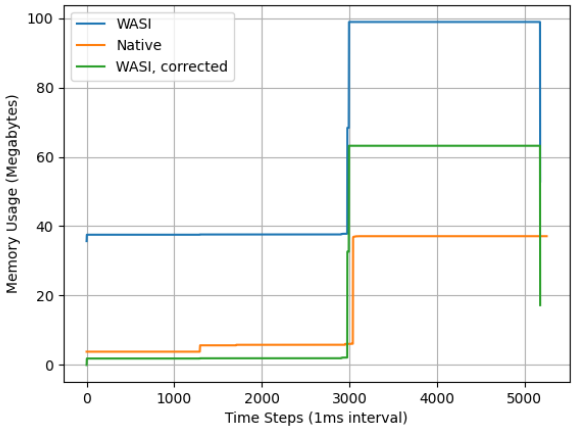

Fig. 9. Memory usage when traversing the file tree. The WASI program consumes 69.8% more memory than the native counterpart.

as Section IV-C1, except that it will not read out the file contents. The file tree has also changed and now contains 11947 files, with a total size of 3.25GB.

Figure 9 shows the results of the benchmark. The graph contains the results of the total memory usage of both programs, and also contains the WASI results without the initial memory occupied by Wasmtime. This is a useful metric, as a program utilizing the USB API can be part of a larger piece of code, where the initial memory overhead of Wasmtime gets distributed between all components and can be neglected.

The first 3000 measurements show the memory usage from when the program is sleeping for three seconds. The memory of both native and WASI programs are similar, not accounting for the initial Wasmtime memory usage. Furthermore, the WASI memory usage is slightly better at the start, but this is likely caused by applying the correction. Once the useful program code starts, memory spikes for both programs. However, the WASI program consumes approx. 63MB of memory, 69.8% more than the native's 37.1MB memory usage.

V. Conclusion

The proposal that was developed over the year, together with the initial working implementation in Wasmtime, has dug through most of the questions and issues that one would expect when developing such an API. While the proposal still has a long way to go, it has laid out the fundamentals of the API which can be expanded further upon.

One of the most prevalent questions was how access control, one of the prime features of WASI, could be integrated in the API. After trying out multiple possiblities, it was concluded that limiting access control on a device-level is deemed enough. The idea is also implemented in the implementation and works. The proposal also contains directions for adding access control to other parts of the interface, should this be needed.

Multiple API designs were tested out, one leaning more to what libusb offers, while the other leaning more into WebUSB. After trying out both APIs and getting feedback from the community, an API which closely follows libusb was chosen. This API is more aligned with how USB devices work internally. This also makes it easier for existing programs using libusb to port their code over to WASI USB.

Performance of the API has been thoroughly tested and results are overall positive. Wasm is fast enough to not have notable performance issues when using the USB API, especially when compared to the latency of communicating with an external device. However, due to the isolated memory of WASI components, data sent from and to a device needs to be copied, bringing in memory overhead. This can become a problem for applications that use a lot of memory.

VI. Future Work

Research done for this thesis has laid the groundwork for building a WASI USB API. However, the proposal is still at phase 1 and has a long way to go before it can become stable. Further work will need to happen on testing, providing multiple runtime implementations, creating fully functional programs that utilise the API and gaining further community feedback.

Also, the API currently has some shortcomings that will need to be addressed in the future:

- The proposal currently ignores language support, but USB devices can provide certain data, like device names, in multiple languages.
- Windows Hotplug support is currently missing, making the events interface of the proposal currently unavailable for Windows. However, recent activity shows that this might come soon [11].

References

[1] WebAssembly, "Webassembly/wasi-usb." [Online]. Available: https://github.com/WebAssembly/wasi-usb
[2] "libusb docs." [Online]. Available: https://libusb.sourceforge.io/api-1.0/group___libusb___dev.html
[3] "Rust." [Online]. Available: https://www.rust-lang.org/
[4] "libusb." [Online]. Available: https://libusb.info/
[5] "Rusb." [Online]. Available: https://docs.rs/rusb/latest/rusb/
[6] "Wasmtime." [Online]. Available: https://wasmtime.dev/
[7] "Canonical abi." [Online]. Available: https://component-model.bytecodealliance.org/advanced/canonical-abi.html
[8] Wouter01, "Wouter01/usb_wasi." [Online]. Available: https://github.com/Wouter01/USB_WASI
[9] "Why the component model?" [Online]. Available: https://component-model.bytecodealliance.org/design/why-component-model.html
[10] "wasmtime::store." [Online]. Available: https://docs.rs/wasmtime/20.0.2/wasmtime/struct.Store.html
[11] "windows: hotplug implementation by sonatique." [Online]. Available: https://github.com/libusb/libusb/pull/1406

Standaardiseren van USB in de WebAssembly System Interface

Wouter Hennen
Promotoren: prof. dr. Bruno Volckaert, prof. dr. ing. Filip De Turck
Begeleiders: dr. ing. Merlijn Sebrechts, ing. Michiel Van Kenhove

Abstract—Deze thesis onderzoekt de toevoeging van USB-ondersteuning aan de WebAssembly System Interface (WASI). Het doel is om ontwikkelaars in staat te stellen om op een platformonafhankelijke en programmeertaalonafhankelijke manier te communiceren met USB-apparaten, vergelijkbaar met WebUSB en WebAssembly in browsers. Door gebruik te maken van de ingebouwde access control van WASI, kan dit op een veilige manier worden bereikt, waardoor gebruikers efficiënt het beheer van de toegang tot USB-apparaten kunnen regelen. Dit document beschrijft de architectuur, het voorstel, de implementatie en de evaluatie van deze integratie. Aan de hand van benchmarks worden de prestaties van de implementatie geëvalueerd.

Index Terms—WebAssembly, WebAssembly System Interface, Component Model, Wasm, WASI, USB, wasi-usb, access control

I. Inleiding

In Web browsers wordt Wasm gebruikt om gecompileerde code uit te voeren in een lichte virtuele machine. Code wordt gecompileerd naar Wasm, maar kan worden uitgevoerd op elk platform dat Wasm ondersteunt, waardoor het platformonafhankelijk is. Dit model kan ook buiten de browsercontext worden toegepast en kan enkele problemen van native code oplossen. Omdat Wasm primair gericht is op webbrowsers, bevat het alleen API's om te communiceren met browsers. Als Wasm-code buiten de browser moet worden uitgevoerd, zijn er nieuwe API's nodig. Om dit op te lossen, is er een nieuwe specificatie gemaakt: de WebAssembly System Interface (WASI). WASI is een verzameling API-specificaties die door Wasm-toepassingen buiten de browser kunnen worden gebruikt. Dankzij WIT en de canonical ABI kunnen deze API's een meer gebruiksvriendelijke interface aanbieden in vergelijking met Wasm. Deze API's kunnen nog steeds worden gebruikt door elke programmeertaal die WASI ondersteunt. Door gebruik te maken van gegenereerde bindings, kan de in WIT gedefinieerde API worden omgezet naar variant die past bij de programmeertaal en compatibel is met de features van de taal. Zo kunnen bijvoorbeeld talen een verschillende stringrepresentatie hebben maar toch dezelfde API gebruiken. Omdat de API's fungeren als een laag tussen de native API's en de applicatie, kan access control worden toegepast om te bepalen waartoe een applicatie toegang heeft.

Deze thesis breidt de WebAssembly System Interface (WASI) uit met een gestandaardiseerde USB-API, die veilige en efficiënte communicatie met USB-apparaten op verschillende platforms mogelijk maakt. De USB-API maakt gebruik van de access control van WASI om de toegang tot USB-apparaten te beveiligen. Er zal ook aandacht worden besteed aan de draagbaarheid en prestaties van de API.

II. Voorstel

De wasi-usb repository [1] bevat de inhoud van het voorstel om de USB-interface te standaardiseren. Het voorstel bevat een eerste versie van de WIT-interface van de USB-API. In deze sectie worden de belangrijkste onderdelen van de API belicht. Het wasi-usb voorstel bevindt zich ten tijde van het schrijven in de proposal fase 1.

Het voorstel bevat de volgende functies:

- Lijst van USB-apparaten krijgen.
- Descriptors van apparaten lezen.
- Device handles openen.
- Gegevens van apparaten lezen.
- Gegevens naar apparaten schrijven.
- Melding ontvangen wanneer een apparaat aangesloten wordt.
- Melding ontvangen wanneer een apparaat verwijderd wordt.

Figuur 1 toont een algemeen overzicht van de verschillende onderdelen van een programma dat de USB-API gebruikt. De implementatie bevindt zich in het 'WASI USB'-blok; hier gaat het voorstel over.

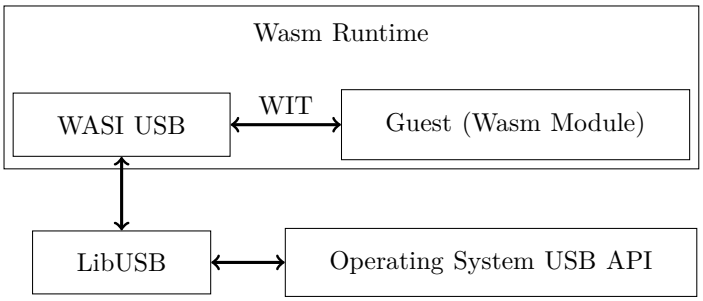

Fig. 1. Interne onderdelen van de WASI USB API. De pijlen vertegenwoordigen communicatie tussen de verschillende gebieden.

A. WIT interface

Het voorstel bevat vier WIT-interfaces: usb, events, descriptors en types. De belangrijkste, usb, wordt hier besproken. Deze wordt getoond in Figuur 2, enkele details zijn weggelaten voor het compact te houden.

De interface bevat twee resources: usb-device en device-handle. De usb-device resource bevat een statische functie enumerate, waarmee alle beschikbare USB-apparaten kunnen worden opgehaald. Deze worden geretourneerd als instanties van usb-device. De andere drie methoden kunnen op een van deze instanties worden gebruikt. De functies device-descriptor en configurations zullen beide extra informatie over een apparaat retourneren en zijn altijd beschikbaar. Dat wil zeggen, ze hoeven geen exclusieve toegang tot deze apparaten te hebben om die informatie te verkrijgen. De functie open is in dit opzicht anders. Deze functie zal een context opzetten waarmee u met het apparaat kunt communiceren. Een belangrijk aspect is dat het openen van een apparaat kan mislukken om verschillende redenen. Sommige besturingssystemen kunnen bijvoorbeeld weigeren een apparaat te openen als de uitvoerbare code niet voldoende rechten heeft. Deze rechten worden afgehandeld op Wasm-runtime niveau en zijn niet specifiek voor Wasm-modules. Om een robuuste en intuïtieve API te creëren, zijn functies die een 'open' apparaat vereisen, gesitueerd in de device-handle resource, die alleen kan worden verkregen als een apparaat succesvol kan worden geopend.

De device-handle resource zal alle functies bevatten die communicatie met een apparaat vereisen, bijvoorbeeld voor de de actieve configuratie van een apparaat op te halen. Er bestaan functies om te lezen en te schrijven met de vier transfer types: read-interrupt, write-interrupt, enz. De meeste functies in device-handle zijn nauw verwant aan vergelijkbare LibUSB-functies [2].

B. Access Control

Het voorstel specificeert access control op een apparaatniveau. Het doet dit door niet-toegestane apparaten te filteren voor usb.usb-device/enumerate, wat een lijst van alle beschikbare apparaten geeft, en events.update, wat updates geeft van apparaatverbindingen. Een gastmodule kan alleen een usb-device resource ontvangen van deze twee functies, dus als ze die resource niet retourneren, kan de gast deze niet gebruiken.

Toegangscontrole kan ook worden gespecificeerd op configuratie-, interface- en eindpuntniveau, maar deze zijn voorlopig niet toegevoegd, omdat nog niet bekend is of ze voordelen zouden bieden.

C. Gebruik in meerdere componenten

Het is mogelijk dat meerdere componenten tegelijkertijd communiceren met de USB-API en proberen toegang te krijgen tot een USB-apparaat. Dit moet worden vermeden, aangezien een component over het algemeen exclusieve toegang tot een apparaat verwacht. Om dit probleem op te lossen, moet de runtime bijhouden welke apparaten door welke componenten worden gebruikt en het openen van een apparaat weigeren wanneer een andere component het al gebruikt. Deze benadering volgt hetzelfde model als de meeste besturingssystemen, waarbij slechts één programma toegang kan hebben tot een USB-apparaat tegelijk.

```
package component:usb@0.2.0;

interface usb {
    use types.{usb-error};
    use descriptors.{configuration-descriptor,
        device-descriptor};

    type duration = u64;

    resource usb-device {
        configurations: func() -> ...;

        device-descriptor: func() -> device-
            descriptor;

        open: func() -> result<device-handle, usb-
            error>;

        enumerate: static func() -> list<usb-device
            >;
    }

    resource device-handle {
        reset: func() -> result<_, usb-error>;
        active-configuration: func() -> result<u8,
            usb-error>;

        select-configuration: func(configuration: u8
            ) -> result<_, usb-error>;

        claim-interface: func(%interface: u8) ->
            ...;

        release-interface: func(%interface: u8);

        select-alternate-interface: func(%interface:
             u8, setting: u8) -> ...;

        read-interrupt: func(...) -> ...;
        write-interrupt: func(...) -> ...;

        read-bulk: func(...) -> ...;
        write-bulk: func(...) -> ...;

        read-isochronous: func(...) -> ...;
        write-isochronous: func(...) -> ...;

        read-control: func(...) -> ...;
        write-control: func(...) -> ...;
    }
}
```

Fig. 2. USB interface.

III. Implementatie

Deze sectie gaat dieper in op de interne aspecten van de implementatie. Eerst wordt de runtime-implementatie

besproken. Vervolgens wordt uitgelegd hoe de API in een gastcomponent kan worden gebruikt.

De wasi-usb [1] repository bevat de implementatie die in deze sectie wordt uitgelegd, evenals een aantal gastcomponenten die hiervan gebruikmaken.

A. Runtime-implementatie

De implementatie is geschreven in Rust [3]. Voor de native implementatie wordt libusb [4] gebruikt om OS-specifieke API's aan te roepen. De Rusb [5] Rust package wordt gebruikt en voegt een thin wrapper rond de libusb-API toe, waardoor het eenvoudiger te gebruiken is in Rust. De implementatie breidt de Wasmtime [6] Wasm-runtime uit en zal Wasmtime in zijn gecompileerde code inbedden. Bij het uitvoeren van het programma kunnen drie parameters worden doorgegeven:

- Gastcomponentpad (vereist): Het pad naar de gastcomponent die zal worden uitgevoerd. De component moet een command component zijn die een run functie bevat.
- '–usb-devices': Een lijst van apparaten waartoe de gastcomponent toegang heeft. De apparaten worden gespecificeerd in het vendor_id:product_id formaat.
- '–usb-use-denylist': Het toevoegen van deze vlag zal de apparatenlijst veranderen van een toestemmingslijst naar een ontkenningslijst.

Standaard heeft de gast geen toegang tot apparaten. Tabel I toont de mogelijke combinaties voor apparaattoegang.

	Devices Specified	No devices specified
Allowlist	specified devices allowed	no device allowed
Denylist	all but specified allowed	all allowed

TABLE I
Opties voor toegangscontrole van apparaten.

B. Gebruik door gastcomponent

Om gebruik te kunnen maken van de USB-API, moet een gast eerst de interfaces in zijn WIT-interface importeren, zoals weergegeven in Figuur 3. Deze component is een Command-component en hoeft daarom geen exports te hebben. Op basis van deze definitie kan een generator bindings aanmaken die natuurlijk aanvoelen voor de taal waarin de gast is geschreven. De gegenereerde bindings zullen de gebruiksvriendelijke API vervolgens omzetten naar een representatie die voldoet aan de canonical ABI [7].

IV. Evaluatie

De API is op drie manieren geëvalueerd. Eerst werd een functionele evaluatie uitgevoerd om te controleren of de API zich gedraagt zoals verwacht. Vervolgens werd een vertragingsevaluatie uitgevoerd om te meten of er merkbare vertragingen zijn bij het gebruik van de API vergeleken met native code. Ten slotte werd

```
package component:usb-component-wasi-stadia;

world root {
    import component:usb/types@0.2.0;
    import component:usb/usb@0.2.0;
    import component:usb/events@0.2.0;
    import component:usb/descriptors@0.2.0;
}
```

Fig. 3. De WIT world voor de gastcomponent.

```
dpad: ,
buttons: assistant_button|l2_button|r2_button,
left stick: x: 128 y: 128,
right stick: x: 128 y: 128,
l2: 255,
r2: 172
```

Fig. 4. Output na het lezen van de staat van de controller.

een geheugenevaluatie uitgevoerd om de impact op het geheugengebruik van het uitvoeren van de code in Wasm te meten in vergelijking met native code.

De code van alle benchmarks is te vinden in de USB_WASI [8] repository.

A. Functionele evaluatie

De functionele evaluatie raakt alle onderdelen van de API: het verkrijgen van verbindings- en ontkoppelingsgebeurtenissen van apparaten, het lezen van descriptors van een apparaat, het openen van een device handle, het lezen van gegevens van het apparaat en het schrijven van gegevens naar het apparaat.

Het algemene idee van het programma is om een gamecontroller te bedienen. Het programma wordt gestart en observeert de verbonden apparaten. Zodra een controller is verbonden die wordt herkend door het programma, zal het programma verbinding maken met de controller. De status van alle knoppen van het apparaat wordt gelezen en geprint. Om het verzenden van gegevens via de USB-interface te testen, stuurt het programma commando's naar de controller om de trilmotoren te activeren. Als het programma een ontkoppelingsgebeurtenis ontvangt, stopt het met het lezen van gegevens en wordt het weer inactief, in afwachting van nieuwe verbindingen.

Dit programma is getest met een Google Stadia-controller. Wanneer een of meerdere knoppen worden ingedrukt, wordt de juiste status afgedrukt. Wanneer een of beide schouderknoppen worden ingedrukt, begint de controller te trillen. Figuur 4 toont een voorbeeldoutput met de status van de controller.

B. Vertragingsevaluatie

Vertraging wordt getest met twee benchmarks: een synthetische benchmark, waarbij een Arduino constant

gegevens zal verzenden, en een benchmark in de echte wereld, waarbij bestanden van een USB mass storage apparaat worden gelezen.

1) Lezen van gegevens van Arduino: De benchmark zal 1 miljoen pakketten van 64 bytes over de bulkinterface van de Arduino lezen. In totaal wordt er 64 MB aan gegevens gelezen. De gegevens worden sequentieel gelezen, zonder vertragingen. Om de ontvangst van de gegevens te starten, stuurt het programma eerst het Device Terminal Ready-signaal naar de Arduino. Zodra de Arduino dit signaal ontvangt, begint hij de gegevens te verzenden. De Arduino voert zo min mogelijk verwerking uit en verzendt een hardcoded array van bytes. De host ontvangt de gegevens en verwerkt deze niet, maar gooit ze weg. Dit is om zoveel mogelijk extra vertragingsfactoren te vermijden.

Figuur 5 toont een boxplot van de vertragingen van beide programma's. Na een miljoen metingen voor elk programma is de mediaan exact hetzelfde, 377 µs. Het gemiddelde van het native programma is marginaal lager, 0,2% sneller dan het WASI-programma. De whiskers en IQR van het WASI-programma zijn iets kleiner, wat een consistenter resultaat betekent. De verschillen in resultaten zijn echter te klein om enige merkbare vertraging te concluderen.

Figuur 5 bevat ook uitschieters, aangeduid met de zwarte cirkels. Sommige van de uitschieters zijn erg groot. Daarom worden de resultaten weergegeven op een logaritmische schaal. Deze uitschieters worden waarschijnlijk veroorzaakt door transmissiefouten bij het ontvangen van gegevens van de Arduino. De Bulk interface wordt gebruikt, die gegevensintegriteit biedt maar geen timinggaranties. Hierdoor wordt dataverlies voorkomen, maar worden hoge latenties geïntroduceerd zoals hier waargenomen. Tabel II toont meer informatie over de uitschieters. Aangezien het percentage uitschieters zeer klein is en in gelijke hoeveelheden voorkomt in beide gevallen, zijn ze uitgesloten van verdere figuren om de duidelijkheid van de grafieken te verbeteren.

Figuur 6 toont een Kernel Density Estimation voor de metingen van de native en WASI-implementatie. Beide grafieken tonen pieken rond de 150 µs en 910 µs. Dit is een interessant resultaat, aangezien men één uniforme verdeling zou verwachten in plaats van twee. De eerste piek is eenvoudig te verklaren: de Arduino is een USB 2.0-apparaat, ook wel bekend als een High-speed USB-apparaat. Een High-speed USB-apparaat stuurt frames met een vaste interval van 125 µs. Daarom ontvangen we ongeveer elke 125 µs nieuwe gegevens. Een extra 25 µs wordt toegevoegd als gevolg van verwerking.

De tweede piek is subtieler. Tabel III toont een fragment van de gemeten vertragingen. Voor zowel de native als de WASI-code zullen de vertragingen tussen beide pieken

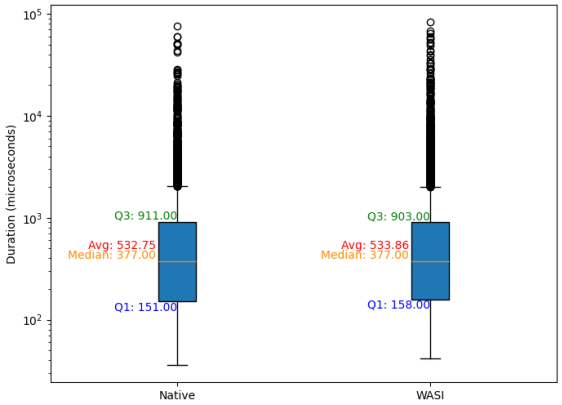

Fig. 5. Boxplot van vertragingen bij het lezen van data van Arduino met read-bulk.

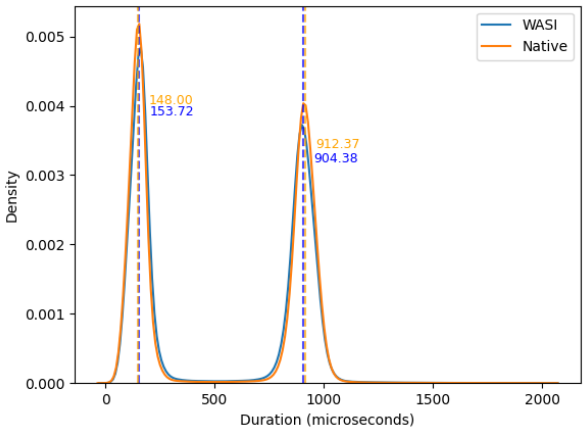

Fig. 6. KDE van vertragingen bij het lezen van data van Arduino met read-bulk.

oscilleren. Op basis van deze resultaten wordt vermoed dat deze piek wordt veroorzaakt door de kleine buffergrootte in de Arduino. De buffer is 64 bytes groot, wat overeenkomt met de grootte van één pakket. Wanneer de buffer vol is, stopt het programma totdat er weer ruimte beschikbaar is in de buffer. Op de host wordt dit waargenomen als een extra vertraging bij het ontvangen van de gegevens.

	Native	WASI
Largest Outlier (us)	75618	83052
Amount of Outliers (>2000 us)	747	1058
Percentage of Total	0.07%	0.10%

TABLE II
Vergelijking van uitschieters bij vertragingen tussen native en WASI.

Op basis van de resultaten kunnen we concluderen dat er geen meetbare verschillen zijn tussen het native en WASI-

Native (us)	WASI (us)
801	918
150	163
885	931
132	241
903	799
110	197
938	820
104	77
962	1003
108	97

TABLE III
Stuk van vertragingsmetingen. De vertragingen oscilleren tussen een lange en korte vertraging.

programma. Dit wordt echter voornamelijk veroorzaakt door de vertraging van het USB-protocol. Deze vertraging weegt ruimschoots op tegen de vertragingen veroorzaakt door de overhead van de Wasm-runtime, waardoor de overhead van Wasm verwaarloosbaar is.

Het is echter mogelijk dat de vertraging van het USB-protocol kleiner is op krachtigere apparaten dan een Arduino of met een modernere USB-versie. Met deze configuraties kan het mogelijk zijn dat een kleine overhead voor Wasm zichtbaar wordt.

2) Bestanden lezen vanaf een Mass Storage-apparaat: Om te bevestigen dat WASI geen merkbare vertraging toevoegt, wordt een andere benchmark uitgevoerd waarbij de inhoud van een USB-massastorageapparaat wordt gelezen. Het voordeel van deze benchmark ten opzichte van de Arduino-benchmark is dat deze beter de real-world-gebruik vertegenwoordigt, in plaats van een synthetische benchmark te zijn.

Er is een programma geschreven dat de bestandsstructuur van een USB-apparaat opsomt en de inhoud van elk bestand in de bestandsstructuur leest. Een Samsung T5 External SSD wordt gebruikt om deze tests uit te voeren. Het apparaat is verbonden met een kabel die een overdrachtssnelheid tot 5 Gbps ondersteunt en is geformatteerd met de MBR-partitiemap en het exFAT-bestandssysteem. Er staan 10 bestanden op het apparaat, waarvan de meeste een paar KB groot zijn. Één bestand heeft een grootte van 679 MB. In totaal is er 680 MB op het apparaat opgeslagen. De benchmark wordt 1000 keer uitgevoerd.

Figuur 7 toont een boxplot van de gemeten totale tijden om de bestandsstructuur te lezen. De gemiddelde uitvoeringstijd van het WASI-programma is 4,2% langzamer dan die van het native programma, en de mediaan 4,3% langzamer. Het WASI-programma heeft ook meer uitschieters, wat aangeeft dat de prestaties van het WASI-programma mogelijk iets minder consistent zijn.

Op basis van deze resultaten kunnen we concluderen dat het uitvoeren van de code met behulp van de WASI USB-API een kleine hoeveelheid overhead met zich meebrengt.

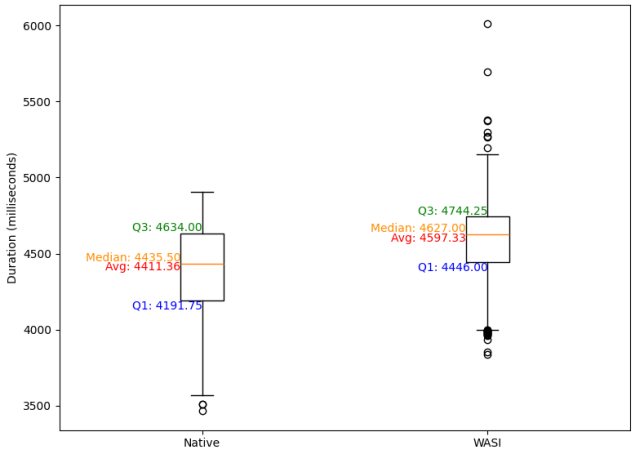

Fig. 7. De vertraging van het lezen van de bestandsstructuur en inhoud na het wijzigen van de benchmarkmethode. Gemiddeld genomen is de WASI USB-implementatie ongeveer 4,2% langzamer.

C. Geheugenevaluatie

In tegenstelling tot standaard Wasm-modules hebben WASI-componenten afzonderlijke geheugenruimtes. Als gevolg hiervan kan één component niet het geheugen van een andere component lezen of schrijven [9]. Het is daarom interessant om te zien welke invloed WASI heeft op het geheugengebruik van een typisch programma.

1) Bestanden lezen vanaf mass storage apparaat: Dezelfde testmethodologie als in Sectie IV-B2 wordt gebruikt om het geheugengebruik te benchmarken. In plaats van de benchmark 1000 keer uit te voeren, wordt deze nu één keer uitgevoerd. Het programma zal ook drie seconden wachten voordat en nadat het apparaat is geopend en gesloten, zodat de evolutie van het geheugengebruik beter zichtbaar is.

Figuur 8 toont de verschillen in geheugengebruik voor beide programma's. Aangezien beide programma's veel geheugen gebruiken, wordt het initiële Wasmtime-geheugengebruik verwaarloosd, omdat dit geen grote invloed heeft op de resultaten. De invloed van het initiële Wasmtime-geheugen wordt besproken in Sectie IV-C2.

Na de eerste drie seconden wachttijd beginnen beide programma's de bestandsstructuur te doorlopen en de inhoud te lezen. Het is duidelijk dat het WASI-programma meer geheugen gebruikt en meer pieken vertoont. Omdat het geheugen niet wordt gedeeld, worden waarden gekopieerd van host naar gast, wat het geheugengebruik verhoogt. Op zijn hoogtepunt zal het 1239 MB gebruiken, terwijl het native programma 718 MB gebruikt. Dit is 72,5% meer. Dit piekgebruik treedt op bij het lezen van het grootste bestand van 679 MB. Omdat de schijf grotendeels leeg is, worden gegevens aaneengesloten op de schijf opgeslagen. Dit verbetert de leesprestaties en zorgt voor een gelijkmatige stroom van gegevens aan het ontvangende einde. Dit is duidelijk zichtbaar in de grafiek voor

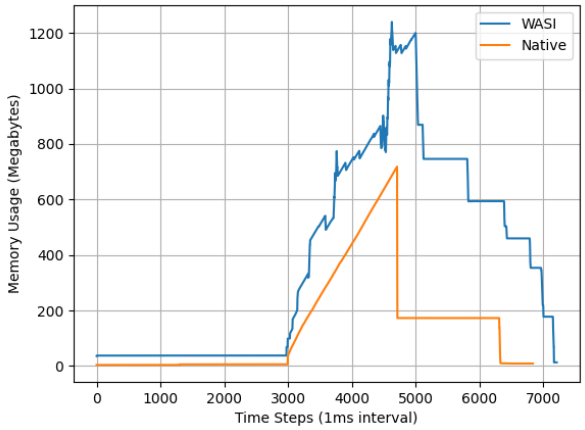

Fig. 8. Het geheugengebruik bij het doorlopen van de bestandsstructuur en het lezen van de bestandsinhoud. Aan het einde van het lezen van het grootste bestand (678 MB) zal het WASI-programma 72,5% meer geheugen gebruiken dan het equivalente native programma.

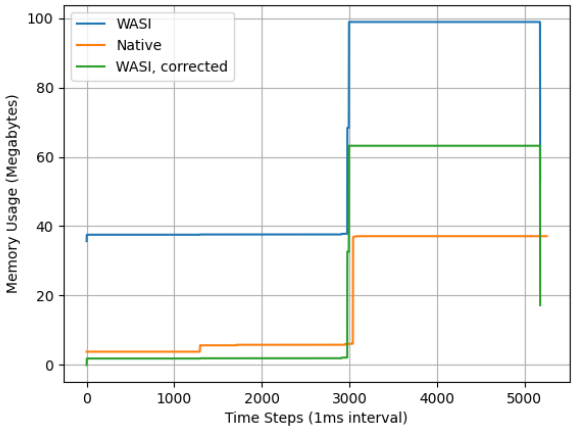

Fig. 9. Geheugengebruik bij het doorlopen van de bestandsstructuur. Het WASI-programma verbruikt 69,8% meer geheugen dan het equivalente native programma.

het native programma, omdat het geheugengebruik een lineaire functie volgt. De WASI-implementatie bevat meer pieken. Deze pieken worden waarschijnlijk veroorzaakt door het kopiëren van waarden, waardoor snel geheugen wordt toegewezen en vrijgegeven.

Nadat de inhoud is gelezen, zal het geheugengebruik voor beide programma's snel dalen. Het geheugengebruik voor het WASI-programma zal echter langzamer dalen dan voor het native programma. Dit gebeurt voornamelijk vanwege het memorypool-achtige gedrag van de Store [10], die het geheugen niet vrijgeeft. Geheugen dat buiten de Store wordt gebruikt, wordt nog steeds vrijgegeven, vandaar de afname van het geheugengebruik. Zodra het programma eindigt, zal het geheugen verder worden vrijgegeven.

2) Overlopen van file tree van mass storage apparaat: Een minder geheugenintensieve benchmark is ook uitgevoerd om te controleren hoe het WASI-geheugengebruik zou zijn bij gebruik van minder geheugen. Deze benchmark gebruikt dezelfde testmethodologie als Sectie IV-C1, behalve dat het de bestandsinhoud niet uitleest. De bestandsstructuur is ook gewijzigd en bevat nu 11947 bestanden, met een totale grootte van 3,25 GB.

Figuur 9 toont de resultaten van de benchmark. De grafiek bevat de resultaten van het totale geheugengebruik van beide programma's, en bevat ook de WASI-resultaten zonder het initiële geheugen dat wordt ingenomen door Wasmtime. Dit is een nuttige maatstaf, omdat een programma dat de USB-API gebruikt deel kan uitmaken van een groter stuk code, waarbij de initiële geheugenoverhead van Wasmtime wordt 'verdeeld' over alle componenten en kan worden verwaarloosd.

De eerste 3000 metingen tonen het geheugengebruik vanaf het moment dat het programma drie seconden slaapt. Het geheugen van zowel de native als de WASI-programma's is vergelijkbaar, als we het initiële Wasmtime-geheugengebruik niet meerekenen. Bovendien is het geheugengebruik van WASI aan het begin iets beter, maar dit wordt waarschijnlijk veroorzaakt door de correctie. Zodra de nuttige programmacode begint, zijn er geheugenpieken voor beide programma's. Het WASI-programma verbruikt echter ongeveer 63 MB geheugen, 69,8% meer dan het geheugengebruik van de native van 37,1 MB.

V. Conclusie

Het voorstel dat gedurende het jaar is ontwikkeld, samen met de initiële werkende implementatie in Wasmtime, heeft de meeste vragen en problemen doorgelicht die men zou verwachten bij het ontwikkelen van een dergelijke API. Hoewel het voorstel nog een lange weg te gaan heeft, heeft het de fundamenten van de API uiteengezet die verder kunnen worden uitgebreid.

Een van de meest voorkomende vragen was hoe access control, een van de belangrijkste kenmerken van WASI, geïntegreerd kon worden in de API. Na het uitproberen van meerdere mogelijkheden werd geconcludeerd dat het beperken van de access control op een apparaatniveau voldoende is geacht. Het idee is ook geïmplementeerd in de implementatie en werkt. Het voorstel bevat ook richtlijnen voor het toevoegen van access control aan andere delen van de interface, mocht dit nodig zijn.

Er zijn meerdere API-ontwerpen getest, waarbij de ene meer leunt op wat libusb biedt, terwijl de andere meer leunt op WebUSB. Na het uitproberen van beide API's en het krijgen van feedback van de gemeenschap, is gekozen voor een API die nauw aansluit bij libusb. Deze API is meer in lijn met hoe USB-apparaten intern werken. Dit maakt het ook gemakkelijker voor bestaande programma's

die libusb gebruiken om hun code over te zetten naar WASI USB.

De prestaties van de API zijn grondig getest en de resultaten zijn over het algemeen positief. Wasm is snel genoeg om geen merkbare prestatieproblemen te hebben bij het gebruik van de USB-API, vooral in vergelijking met de vertraging van de communicatie met een extern apparaat. Echter, vanwege het geïsoleerde geheugen van WASI-componenten, moet data die van en naar een apparaat wordt verzonden worden gekopieerd, wat extra geheugenoverhead met zich meebrengt. Dit kan een probleem worden voor toepassingen die veel geheugen gebruiken.

VI. Toekomstig Werk

Het onderzoek voor deze scriptie heeft de basis gelegd voor het bouwen van een WASI USB API. Echter, het voorstel bevindt zich nog steeds in fase 1 en heeft nog een lange weg te gaan voordat het stabiel kan worden. Verder werk zal moeten plaatsvinden op het gebied van testen, het leveren van meerdere runtime-implementaties, het maken van volledig functionele programma's die de API gebruiken en het verkrijgen van verdere feedback van de gemeenschap.

Bovendien heeft de API momenteel enkele tekortkomingen die in de toekomst moeten worden bekeken:

- Het voorstel negeert momenteel taalondersteuning, maar USB-apparaten kunnen bepaalde gegevens, zoals apparaatnamen, in meerdere talen verstrekken.
- Windows Hotplug-ondersteuning ontbreekt momenteel, waardoor de events-interface van het voorstel momenteel niet beschikbaar is voor Windows. Recentelijke activiteit wijst echter uit dat dit binnenkort mogelijk zal komen [11].

References

[1] WebAssembly, "Webassembly/wasi-usb." [Online]. Available: https://github.com/WebAssembly/wasi-usb

[2] "libusb docs." [Online]. Available: https://libusb.sourceforge.io/api-1.0/group___libusb___dev.html

[3] "Rust." [Online]. Available: https://www.rust-lang.org/

[4] "libusb." [Online]. Available: https://libusb.info/

[5] "Rusb." [Online]. Available: https://docs.rs/rusb/latest/rusb/

[6] "Wasmtime." [Online]. Available: https://wasmtime.dev/

[7] "Canonical abi." [Online]. Available: https://component-model.bytecodealliance.org/advanced/canonical-abi.html

[8] Wouter01, "Wouter01/usb_wasi." [Online]. Available: https://github.com/Wouter01/USB_WASI

[9] "Why the component model?" [Online]. Available: https://component-model.bytecodealliance.org/design/why-component-model.html

[10] "wasmtime::store." [Online]. Available: https://docs.rs/wasmtime/20.0.2/wasmtime/struct.Store.html

[11] "windows: hotplug implementation by sonatique." [Online]. Available: https://github.com/libusb/libusb/pull/1406

Acknowledgements

I would like to thank my counsellors Dr. ing. Merlijn Sebrechts and ing. Michiel Van Kenhove for their help, commitment, feedback and time to make this thesis possible.

I would also like to thank my supervisors, Prof. dr. Bruno Volckaert and Prof. dr. ir. Filip De Turck.

Additionally, I would also like to thank my colleague Warre Dujardin for the collaboration on the proposal.

I would like to thank the WASI community for the creation of WIT, the available tooling and support.

Toelating tot bruikleen

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Wouter Hennen, 25 mei 2024

Disclaimer regarding the master’s thesis

This master's thesis is part of an examination. Any comments made by the evaluation committee during the oral presentation of the master's thesis were not incorporated into this text.

Contents

List of Figures

List of Tables

List of Acronyms

ABI Application Binary Interface. , 11

API Application Programming Interface. ii, iii, xxi, xxiii, , 16, 17, 18, 20, 24, 25, 26, 27, 33, 34, 36, 45, 46, 48, 51, 53, 54

DTR Data Terminal Ready. , 41

exFAT extensible File Allocation Table. , 44, 45

FAT32 File Allocation Table 32. , 45

GPT GUID Partition Table. , 45

IDL Interface Description Language. , 15

IoT Internet of Things. ii, iii, , 1, 2, 7

KDE Kernel Density Estimation. , 42

MBR Master Boot Record. , 44, 45

NTFS New Technology File System. , 45

PoC Proof of Concept. , 26, 27

POSIX Portable Operating System Interface. , 15

USB Universal Serial Bus. ii, iii, xxi, xxiii, , 16, 18, 19, 23, 24, 26, 27, 30, 31, 33, 34, 40, 44, 45, 46, 48, 49, 51, 53, 54

WASI WebAssembly System Interface. ii, iii, xx, xxiii, , 1, 2, 8, 10, 11, 12, 15, 16, 17, 18, 19, 22, 23, 26, 27, 28, 29, 33, 34, 40, 41, 42, 43, 46, 47, 48, 49, 50, 51, 52, 53, 54

Wasm WebAssembly. ii, iii, xx, xxi, xxiii, , 1, 7, 8, 9, 10, 11, 15, 18, 20, 23, 25, 26, 27, 28, 34, 36, 40, 43, 48, 49, 53

WIT Wasm Interface Type. ii, iii, xx, xxi, , xxvi, 1, 11, 12, 15, 16, 18, 19, 28, 29, 30, 31, 33, 34, 35

List of Code Snippets

1

Introduction

Updating software of Internet of Things (IoT) devices is currently difficult. As the devices are home appliances, they are often used for longer periods of time, compared to other tech such as smartphones. As a consequence, these devices need longer software support. In reality, a lot of devices stop getting updates after a few years, making them vulnerable to cyberattacks. When an IoT device gets hacked, it can be used as part of a botnet, or worse, send sensitive info (such as a video feed) to the attacker. There are multiple causes to why this software support period is short: no standardized updating mechanisms, special hardware and compilers, etc. [1]

In Web browsers, WebAssembly (Wasm) is used to run compiled code in a lightweight virtual machine. Code gets compiled to Wasm, but can be run on any platform that supports Wasm, making it cross-platform. This model can also be applied outside the browser context and can solve some of the problems of native code. However, Wasm is primarily targeted towards browser applications, so most tooling is only available in the browser. Calling APIs to the outside world is also cumbersome in Wasm, as it has a limited featureset to call APIs. Also, no low-level libraries exist for controlling hardware devices, which would be required for IoT devices. To solve this, a new specification was created: the WebAssembly System Interface (WASI)

WASI is a collection of API specifications that can be used by Wasm applications outside the browser. Thanks to Wasm Interface Type (WIT) and the canonical ABI, these APIs can provide a more developer-friendly interface compared to Wasm. These APIs can still be used by any programming language that offers support for WASI. Through the use of generated bindings, the API defined in Wasm Interface Type (WIT) can be converted to a language-tailored variant, integrating well with features the language offers. Because the APIs act as a layer between the native APIs and the application, access control can be applied to control what an application can access.

With WASI, standardized APIs can be created that can be used to talk and control hardware. As the APIs are not device specific anymore and no compiling to specific architectures is needed anymore, updating software for IoT devices has become a lot easier.

Problem Statement

The WASI standard is still in development and currently lacks an API to interact with USB devices. Having an API for USB devices is crucial for IoT devices as a lot of hardware, such as cameras, will oftentimes communicate over USB.

Goal

The goal of this master's thesis is to research and propose a new USB API for WASI. Special attention will be given to access control, portability and performance.

Research Questions

- What aspects of the USB interface can benefit of access control?
- In what way can a WASI API be created that makes porting code easy?
- What is the performance impact of this API compared to native code?

2

USB Foundations

In the early days of computers, various connectors existed for computers to communicate with external devices. A few examples of such connectors are the PS/2 port (Primarily used for Mouse / Keyboard) or the Parallel port (Often used for printers). These connectors have varying sizes, shapes and limitations and cannot be used for all devices. Therefore, computers needed to have all these ports, requiring lots of space. To address these issues, the USB connector was introduced. Years later, the interface has become the de-facto standard for wired device communication. Over the years, USB has evolved with multiple revisions. These revisions focus on key aspects of the interface, such as transfer speeds, power delivery and added functionality. The USB connector has also undergone revisions, making it smaller and reversible, so it is usable on a larger variety of devices.

In this thesis, the focus is on the software side of the USB standard. Therefore, the hardware will not be considered further.

2.1 Transferring Data via USB

This section introduces the internals of the USB protocol. For more information, take a look at USB in a NutShell [1].

2.1.1 Pipes

Data is transferred through pipes. A pipe is a connection from the host controller to the endpoint. Not all pipes are the same: they differ in the bandwidth they support, which transfer types are supported, in which direction data can flow, and their packet and buffer size. Pipes can generally be split up into two kinds.

Streaming Pipes

A Streaming Pipe is a one-way communication channel for the host or guest device to send any kind of data to the other end. This pipe is controlled by either the host of or guest device, and data is sent in a sequential way.

[1] https://www.beyondlogic.org/usbnutshell/usb1.shtml

The isochronous, interrupt and bulk transfer types will use this pipe to send data.

Message Pipes

A Message Pipe is a bidirectional communication channel. This pipe allows both the host and guest device to send commands in either direction on the same pipe. All message pipes are controlled by the host device. Only one transfer type supports this pipe: the control transfer type.

2.1.2 Transfer Types

The USB standard defines four transfer types. Each transfer type serves a different purpose, being optimized for speed, latency, correctness or reliability.

Interrupt Transfer

Interrupt transfers are most used for devices that transfer small amounts of data frequently. They have a bounded latency and are therefore suited for devices that require low latency and low bandwidth. Interrupt transfers can be initiated by both the host and guest device. The sent data will be queued by the sender, until the receiver polls the device.

Examples of devices that often use interrupt transfers are mice and keyboards.

Isochronous Transfer

Isochronous transfers are used for real-time data streaming. They provide a guaranteed data rate, but do not guarantee a correct transfer of data, and data can be lost. This makes the transfer type not suitable for situations where data integrity is important.

An example use case where isochronous data transfer is often used is streaming audio or video.

Bulk Transfer

Bulk transfers are suited for transferring large amounts of data where timing is not an issue, but data integrity is. No guarantee on timing is made, but guaranteed correct delivery is.

An example use case for bulk transfers is sending files to devices.

Control Transfer

Control transfers are used for configuration, command and status operations between the host and guest device. Control transfers operate on the Message pipe and has setup, data and handshake stages. The handshakes

guarantee correct delivery, but will lead to a slower transfer speed. Therefore, control transfers are not used to transfer a lot of data, but rather for device initialization and control.

The Control transfer type can be seen as a TCP connection but for USB devices.

2.1.3 Descriptors

Descriptors are used numerous times throughout the USB specification. They define a standardized way to provide information about a part of the specification.

Details of each descriptor can be found in the appendix 8.

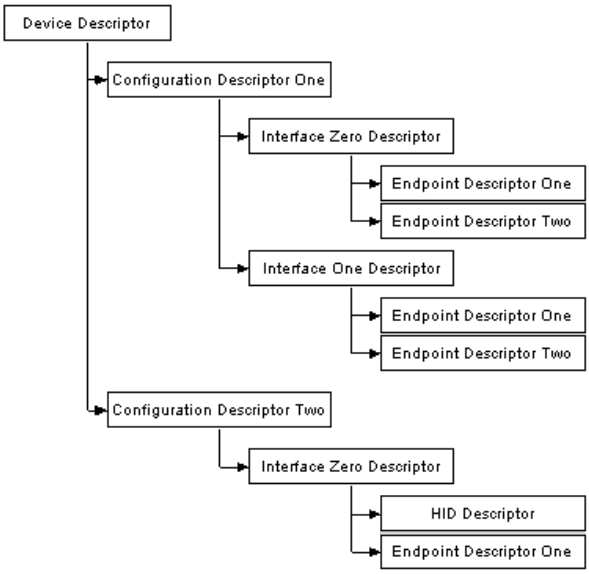

Figure 2.1: Descriptor Tree of a USB device. This tree contains all useful metadata to communicate with the device. [9]

Device Descriptors

Each USB device has a device descriptor which contains general information about the device, such as its device and vendor ID, which protocols it supports, its serial number, how much configurations it has, and more. This descriptor is essential, as it allows a program to find specific devices or kind of devices, so the right device can be chosen to communicate to.

Configuration Descriptors

The configuration descriptor provides information about a configuration of a USB device. A configuration describes the power usage of a device. For example, a device can have a configuration where it is self powered, and a

configuration where it requires extra power from the host. Each configuration can have different combinations of interfaces and endpoints.

When a USB device is connected to a host, the host will examine the configuration descriptors and select the one required by the program.

Interface Descriptors

The interface descriptor provides information such as the interface number, the alternate setting number (if multiple settings are available for the interface), the number of endpoints associated with the interface, etc.

Endpoint Descriptors

Endpoints are the communication channels through which data is transmitted between the host device and the USB device.

The endpoint descriptor provides information such as the endpoint address, transfer speed, transfer type (see Section 2.1.2), etc.

Each endpoint can have any transfer type, except for endpoint zero. The zero endpoint is assumed to be a control endpoint, and will have a control transfer type. Endpoint zero will be used to communicate with the device to get its descriptors.

2.2 Existing Solutions

2.2.1 libusb

libusb [5] is a versatile open-source library that offers platform-independent access to USB devices. Essentially, it is a thin wrapper around the system-provided USB APIs, providing a universal API. libusb is a C library and works on native hardware. It also works completely in user-mode, meaning no special privileges are required to communicate with USB devices. This makes it a very popular library to control USB devices, as it removes most issues that would occur when trying to support multiple platforms. It is also version-agnostic, so it will work with each USB protocol (1.0 - 4.0) [2].

As libusb makes creating cross-platform USB programs easy, it is used by most other technologies in this chapter, such as WebUSB (2.2.2) and our own implementation (See Chapter 5).

[2] https://github.com/libusb/libusb/wiki/FAQ#does-libusb-support-usb-3031324

2.2.2 WebUSB

WebUSB [4] is a library to work with USB devices in browsers.

The WebUSB specification allows webpages to control non-standard USB devices. Browsers already provide easy APIs for common USB devices, such as mice, keyboards, cameras and microphones. However, accessing devices that do not follow these common USB use cases were not supported in the browser. In 2017, the WebUSB specification was created to provide a new API that allows this.

WebUSB has been proven useful in a lot of cases. For example, it has been used to control Arduino devices and upload new programs to these devices. Another kind of use case is to easily upgrade devices through USB, without the requirement to install special software.

For example, after discontinuing its game streaming service Stadia, Google provided a firmware update for its Stadia controllers to enable bluetooth support [10] [11]. This update could be installed by connecting the controller to the computer and following the steps on the website. No additional software needed to be installed, making it very user-friendly.

WebUSB is not part of the web standards and is currently only supported in Chromium-based browsers. WebUSB only provides a limited Javascript API and is an abstraction over the raw USB interface, making it less useful for IoT devices.

WebUSB can currently be used by Wasm applications in the browser through the Javascript interface. However, this solution is far from ideal: there is additional overhead from using Javascript. Javascript is an interpreted language with dynamic and weak types. These capabilities make it harder to make a performant type-safe API. Furthermore, the API provides an abstraction over the USB specification and doesn't expose some parts of it.

3

WebAssembly & WebAssembly System Interface

This chapter gives insight in what WebAssembly (Wasm) is and how it works, alongside additional insight in how the WebAssembly System Interface (WASI) works and how it is progressing.

3.1 Wasm

When the web was born, the only supported programming language supported on the web was Javascript. As a relatively simple and portable language this did suffice. Over the years the web has become more complex and has become the center of the personal computer. However, its supported technologies did not grow at the same expansion pace. Javascript is still the only supported language and hampers the further expansion of the web. New technologies have been created to fix this issue, but none have succeeded. In order to be a viable Javascript alternative, a new model needs to be safe, fast, portable and compact. Wasm is the first new technology to check all these four boxes. [12]

3.1.1 Safe

Wasm is intended to run in various environments, with some executing Wasm code from unknown sources. The web is a prime example of such a case. In these cases, it is required that the code runs in a sandbox, so that it cannot access all resources of the device. This way, malicious code can only do harm in the sandbox, and not the entire device.

Additionally, a Wasm program is often build in multiple modules. A Wasm module can be seen as an island of code, which imports and exports functions. Each module has its own stack, memory, types and more. This means that each module is isolated and cannot affect other modules, increasing the security of the model. Additionally, Wasm makes use of a linear memory model (Figure 3.2). Each module gets its own range, and can only access that part of the memory.

3.1.2 Portable

WebAssembly (Wasm) works as a compilation target for programming languages. Languages can compile to Wasm, just like they would for e.g., x86. To compile to Wasm, the source code is first compiled to an intermediate representation, such as the LLVM IR. This is universal and also happens when compiling to other architectures. This part of the compilation process, called the frontend, stays the same. The backend of the compilation process, where the code gets translated into machine instructions, differs per instruction set. In the backend of the compiler the code will get compiled to the Wasm instruction set and output Wasm byte code. This binary can get executed by a Wasm runtime. (Figure 3.1) The byte code is platform-agnostic, so it can run on any hardware and operating system that has a Wasm runtime. This makes Wasm portable.

This general idea works great for compiled languages, such as C, Rust or Swift. However, for interpreted languages such as Python, the model will work different. On native instruction sets, interpreted languages are read at runtime by their interpreter, which is often written in C. To make this work in Wasm, the interpreter can be compiled to Wasm and can interpret the language source code, just like it would happen on a native machine.

3.1.3 Compact

To achieve a small program, which is often critical for quickly loading websites, Wasm acts as a stack machine instead of a register-based machine. In register-based machines, registers are used to make calculations. On stack machines, argument values and operands are pushed on the stack, and result values are popped from the stack (Figure 3.2). [12] By not having the need for registers, the binary size can be made smaller [13]. Another advantage of this method is that the amount of available registers can vary on different platforms. This is not an issue with stack machines.

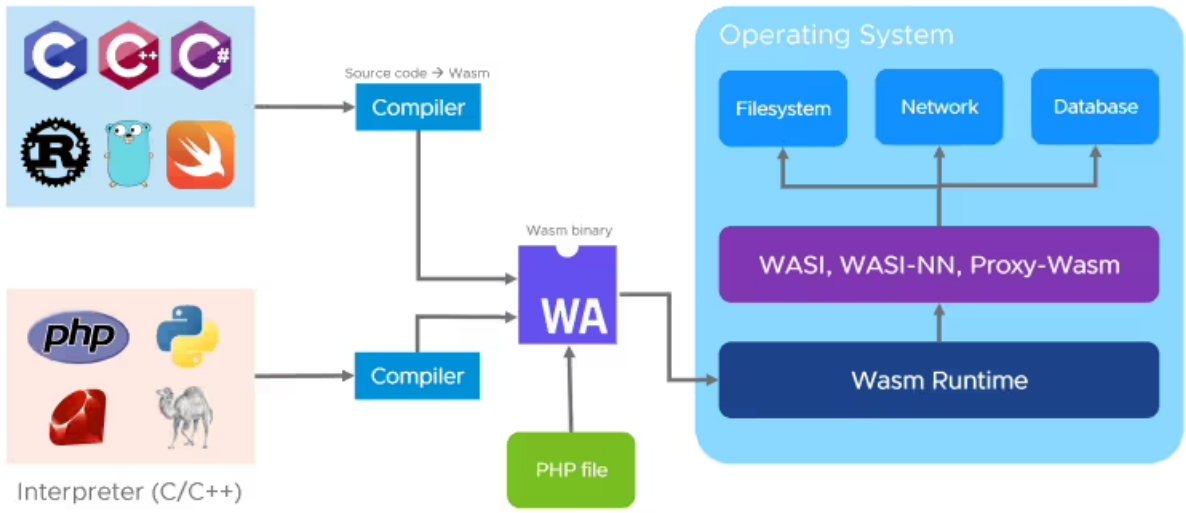

Figure 3.1: Source code gets compiled to the Wasm architecture. The Wasm runtime executes the Wasm byte code. [7]

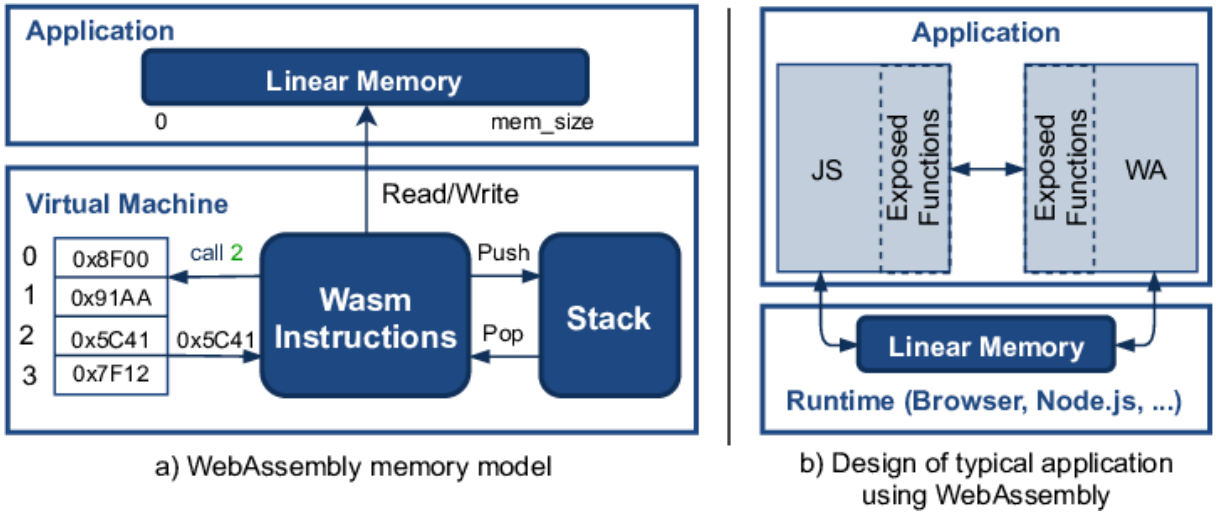

Figure 3.2: Wasm internals. [8]

3.2 WASI

When writing code that compiles to Wasm, there is a need for system interfaces. These are required to do useful stuff, such as getting access to files. In the browser, WebAssembly applications can use Javascript as a system interface. However, outside the browser, there is no such predefined system interface. To solve this problem the WebAssembly System Interface (WASI) was created. WASI is a set of portable APIs, similar to APIs you would find in an operating system.

3.2.1 The Component Model

A Wasm program is often build up of multiple libraries called Wasm modules. These modules need a way to talk to each other. They do this by importing and exporting functions. Imported functions are definitions of functions implemented by another Wasm module. If a module calls an imported function, it will be executed by the other module. Exported functions are functions implemented in the module that are exposed to other modules. This form of communication is hard to use, as only functions with primitive types are allowed. Passing around more complex data structures, such as strings, lists or structs, is not possible. This is because different programming languages may have different data representations for these structures. For example, a String in Rust may be represented in another way than a String in Python.

To solve this issue, the Component Model was introduced. The Component Model allows users to build applications out of reusable components. These components can be written in different programming languages. Conceptually, a component is a wrapper around a Wasm module and adds an extra interface layer above the module. This interface is defined with the WIT language. It does not contain logic, but only describes what a component exports and imports. The WIT language offers a broader range of primitives compared to Wasm primitives. For example, strings, enums (ref. variant) and structs (ref. record) can be defined in a WIT. Section 3.2.2 provides further elaboration on WIT.

With the WIT, components can expose a standardized interface to other components. However, the WIT only contains the definition of that interface. For correct utilization of this interface, the components should conform to the WIT in a standardized way. The canonical ABI was created to account for this. The canonical ABI defines the binary representation of the types used in the WIT [14]. This way, when one module sends data to another module, the other module can identify what data has been sent. The canonical ABI also defines the format of the WIT types, such as how strings are represented in bytes. Each WASI component should thus make sure that the data it sends conforms to the ABI. This would be tedious for someone creating a component. To solve this issue, tools are created to generate bindings. These tools are tailored to specific programming languages, generating types and interfaces that seamlessly integrate with the idiomatic patterns of each language. The resulting bindings facilitate the conversion of language-specific data types to adhere to the canonical ABI standard when transmitting to other components. Conversely, upon receiving data from external sources, the bindings ensure the conversion back to the corresponding language-specific types.

Component kinds

The Component Model allows components to depend on other components. A distinction of what kinds of components exist is made. Two kinds of components exist.

Command Component A Command component is conceptually similar to a regular program on a machine. It has an entry point function labeled `_start`, which acts like a `main` function. This function is called when the component is ran. A command component can depend on reactor components, but not vice versa.

Reactor Component A Reactor component is conceptually similar to a library in other programming languages. Contrary to Command components, it does not contain a `_start` function. Instead, it exports functions and interfaces, which can be called by other Reactor or Command components.

3.2.2 Wasm Interface Type (WIT)

As briefly described in Section 3.2.1, the Wasm Interface Type (WIT) contains a definition of an interface a WASI component exposes. It defines this interface in worlds and interfaces. The WIT does not contain an implementation, but only defines the contract between components. This section will briefly discuss the basics of the WIT language.

World A world can be seen as the entry point of a component: it contains all the imports and exports of a component. Interfaces and functions can be imported and exported. Exports are what the component provides for other components. Imports are interfaces or functions the component requires. Imports may come from the component itself or from others. Finally, a world can also include another world, essentially being a superset of that world and using all its imports and exports.

A world can also reference interfaces of other packages. For example, the following code snippet imports and exports the interface `incoming-handler` of package `wasi:http` [15]:

```
world http-proxy {
    export wasi:http/incoming-handler;
    import wasi:http/outgoing-handler;
}
```

This example file uses the world `http-proxy`. It imports the `outgoing-handler` interface. This interface is declared in the `wasi:http` package and is imported from there. The world will also export the `incoming-handler` interface. This means that other components can interact with this component by using the types and functions in the `incoming-handler` interface.

Interface An interface is a group of types and functions. An interface can represent a certain feature in a library or semantically group a set of elements together. Interfaces declared in other packages can also be used in an interface. This can be done by using the `use` syntax:

```
interface test {
  use wasi:http/incoming-handler@0.2.0{handle};
}
```

In this example:

- **`test`**: An interface we define.
- **`wasi:http`**: The package that contains the interface `incoming-handler`.
- **`incoming-handler`**: An interface in the `wasi-http` package. Note that this interface must be imported in each world that contains the `test` interface.
- `0.2.0`: The version of the `wasi-http` package.
- **`handle`**: A function we want to use, contained in the `incoming-handler` interface.

The example file contains the `test` interface.

Types The WIT language offers a handful of types, which can be defined in two parts:

- Built-in primitives: For example, `u32`, `string`, `list<T>` or `option<T>`.
- User-defined types: These types are built on top of the primitive types provided by WIT. WIT provides different data structures to define custom types:
  - `enum`: An enum in WIT is a classic enum type, equivalent to C-style enums.

    ```
    enum color {
      blue
      green
      yellow
    }
    ```

  - `variant`: Variants are similar to enums, but can have data added to each case. It can be thought of as a combination of an enum and a union.

    ```
    variant file-tree-item {
      file(file),
      folder(string, list<file>),
      symlink(string)
    }
    ```

- `record`: Records are equivalent to C structs. They contain a set of fields, each having a name and a type. A record only contains data, and cannot have functions.

```
record file {
  id: string,
  name: string,
  contents: data,
  headers: list<header>
}
```

- `flags`: The `flags` type is an efficient representation of multiple boolean values. It will use a bitset to represent the values.

```
flags file-permissions {
  read,
  write,
  execute
}
```

Functions A WIT function is equivalent to functions in regular languages. It can contain parameters and a return value. In this example, the function `lookup` takes parameters `store` of type `kv-store` and `key` of type `string`. It will return an optional `string`.

```
lookup: func(store: kv-store, key: string) -> option<string>;
```

Resources A resource is a handle to an object in another component. The resource contains functions that are callable upon the associated object. Calling a function on the resource is equivalent to calling a function on an existential type, a concept seen in other languages like Rust or Haskell [16]. An existential type implements the resource requirements, but the type is not known. This makes resources different from other WIT types: instead of passing data around, they allow components to *access* objects from other components.

In the example below, the resource `file-handle` contains 5 functions, which can be put into two categories:

- static functions: functions `constructor` and `default` are static functions. These do not have a reference to an instance of the resource, but are equivalent to regular functions, except that they are namespaced under `file-handle`. The `constructor` function will create a new instance of the type and is just syntactic sugar for a regular static function:

```
equiv-to-constructor: static func(file: file) -> file-handle
```

- instance functions: `read`, `seek` and `write` are instance functions and are called on an already-existing instance which conforms to the resource. An instance function can be desugared into

  ```
  seek: func(self: borrow<file-handle>, offset: s32) -> u32;
  ```

```
resource file-handle {
    constructor(file: file);
    read(n: u32) -> result<list<u8>, handle-error>;
    seek(offset: s32) -> u32;
    write(data: list<u8>) -> result<_, handle-error>;

    default: static func() -> file-handle;
}
```

3.2.3 Milestones

WASI is still in development and far from complete. In order to track the development of WASI, its creation has been split up in multiple milestones [17].

Preview 1 WASI Preview 1 was launched in 2019 and marked the first significant milestone in improving WebAssembly interaction outside the browser. Preview 1 contains methods to interact with the file system and command line, resembling a subset of the Portable Operating System Interface (POSIX) interface. The API is offered to Wasm modules via the witx Interface Description Language (IDL).

Preview 2 WASI Preview 2 diverges from the classic POSIX interface and introduced the Component Model. The Component Model fixes design issues and brings multiple improvements compared to a POSIX interface, such as having to rely less on file descriptors and being able to use high-level value types instead of unstructured data [18]. Existing APIs have been rebased to work with the WIT language and the component model. Preview 2 is currently the latest preview.

Preview 3 WASI Preview 3 will bring smaller but important updates to the WIT language. It is expected to introduce the `future` and `stream` keywords in the WIT language, which adds the ability to express asynchronous functions in the interface. This will bring improvements to existing components which currently rely on polling.

WASI 1.0 The main goal of WASI 1.0 is to have a stable runtime which offers stable WIT interfaces to interact with various parts of the system, such as the filesystem. The most important proposals, such as `wasi-io` and `wasi-http`, must reach stage 5 of the standardization process [3] before WASI 1.0 can be launched.

4

Proposal

4.1 Standardization

The goal of this master's thesis is to extend the WASI standard with a new USB API. In order to do this, the API must go through a standardization process. This way, a consensus can be made about how the API should work, which platforms it will work on, and what its scope is. When gone through the proposal, one universal API can be offered that is supported by all runtimes which are WASI-compliant.

In order for the USB API to take part in this standardization process, a proposal must be created. This proposal contains all ideas and information required to create an implementation for the API. Throughout the standardization process, the proposal will go through five phases [3]:

Phase 0 (Pre-Proposal) When there is enough interest in a proposal and the idea seems viable, the proposal gets discussed in an online WASI meeting. The person who will create the proposal explains the idea, what it would solve, and how it can and cannot be used. This is often also the first moment to get feedback from the community. Each attendee of the meeting casts a vote about the idea. If the majority of the people approve, the idea can move to Phase 1, and a proposal can be created. Champions (people that will create the proposal) get assigned to the proposal.

At the time of writing, `wasi-usb` is in Phase 1.

Phase 1 To be able to move to phase 2, the requirements for phase 1 are worked out:

- The WIT interface gets created and documented by the champions. The WIT interface should be mostly finalized, but small changes can occur.
- Prototype implementations are required to evaluate the viability of the created API.
- The portability criteria are defined:

- Portability: On which platforms should the API run, what parts of the API are available on which platform.
- Practicality: The proposal should demonstrate that the API can be used in real-world scenarios.
- Testing: Define how the API should be tested and on which devices.
- Implementations: In Phase 3, each proposal should have at least two implementations in different WASI runtimes. The kinds of implementations are defined here.

These portability criteria are defined in this phase, but get worked out in the later phases. Once these requirements are met, the proposal can move to phase 2.

Phase 2 During phase 2, one host implementation is created to test the API. Dependencies of the implementation should have already reached at least stage two. Furthermore, a plan is developed for how the portability criteria will be met.

Phase 3 In Phase 3, extra implementations of the API are made in different WASI runtimes. All implementations should pass the testing defined in the portability criteria and be complete. In addition, the dependencies used in the implementation should have reached the proposal phase 3. To further test the API, Libraries and other tools that use the API are built.

Phase 4 & 5 No proposal has reached phase 4 yet, so the requirements of phase 4 & 5 is not final yet. These stages finalize the standardization of the API.

A full overview of phase requirements can be found here [1].

[1] https://github.com/WebAssembly/meetings/blob/main/process/phases.md

4.2 Timeline

Table 4.1 shows the timeline of events that happened during the year regarding the proposal. At the time of writing, the WASI USB proposal is in Phase 1.

Date	Event
29 November 2023	An issue [19] was created on the WASI repository, stating the interest in creating an USB proposal. Proposal is at phase 0.
5 March 2024	Initial WIT for proposal created
4 April 2024	Presentation [20] of proposal in WASI meeting. Proposal has moved to phase 1

Table 4.1: Timeline of events related to the USB proposal in the WASI repository.

4.3 Architecture

This section briefly explains how the different parts of the API work together. A visualisation of the parts can be seen in Figure 4.1.

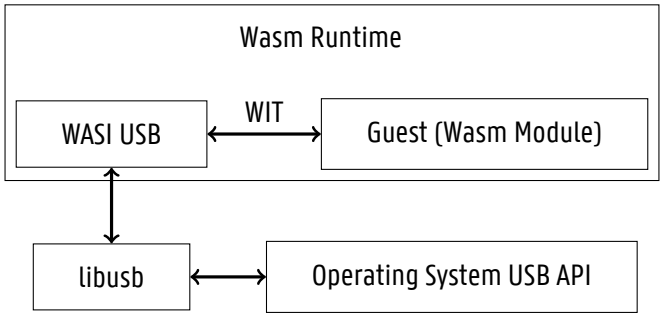

Figure 4.1: Internals of the WASI USB API. The arrows represent communication between the different areas.

Guest This is a Wasm module that is created by a user of the API. It contains code that utilizes the API to communicate with devices. The guest module is sandboxed in the Wasm runtime, meaning that it cannot access any system resources. By utilizing the WASI APIs, such as the WASI USB API, it can get access to system resources. It can communicate with the runtime implementation through the WIT interface.

WASI USB This is the host implementation of the WASI USB API. It acts as a bridge between native USB APIs and Wasm modules. This piece is part of the Wasm Runtime, but is not sandboxed, so it can access system resources. To do this, the code is not compiled to Wasm but is instead part of the native runtime binary. This code is platform-dependent and must ensure the exposed API works as similar as possible on each supported platform.

libusb To help the WASI USB runtime implementation having similar behavior on different platforms, libusb is used. libusb is a lightweight wrapper around native USB APIs [5], but abstracts them to a universal interface. By using libusb, the runtime implementation can be written against the libusb API instead of the need to provide separate implementations for each platform.

Operating System USB API Finally, libusb will call the native libusb APIs. These are provided by the Operating System kernel.

4.4 USB Proposal

The `wasi-usb` repository [2] contains the contents of the proposal to standardize the USB interface. The proposal contains a draft of the WIT interface of the USB API. This section will highlight the important parts of the API.

An important note is that in this chapter, 'WIT interface' and 'interface' will both be used, but have a different meaning. WIT interface refers to the whole WIT file. A WIT file itself also contains interfaces, which are also discussed here as interfaces. An interface can be thought of as an instance in the WebAssembly Component Model, for example a unit of functionality imported from the host or implemented by a component for consumption on a host. All functions and types belong to an interface [21].

`imports` World The `imports` world in code snippet 4.1 is the main entry point of the interface. It imports 4 interfaces: `types`, `descriptors`, `events` and `usb`. These are separate files with different parts of the WIT interface. The world imports them so they are included in the WIT interface. The world will also export these interfaces, making them available for components that use the `imports` world.

```
package component:usb@0.2.0;

world imports {
    import types;
    import descriptors;
    import events;
    import usb;

    export types;
    export descriptors;
    export events;
    export usb;
}
```

Source Code 4.1: `imports` world.

`usb` interface The `usb` interface seen in code snippet 4.2 contains two resources: `usb-device` and `device-handle`. Some details of the interface are omitted for brevity. The `usb-device` resource contains a static function `enumerate`, with which all available USB devices can be retrieved. These will be returned as instances of `usb-device`. The other three methods can be used on one of these instances. The `device-descriptor` and `configurations` functions will both return extra information about a device and are always available. That is, they do not need to have exclusive access to these devices to obtain that information. The `open` function is different in this regard. This function will establish a context with which you can communicate with the device. One important aspect is that opening a device can fail due to various reasons. For example, some operating systems might deny opening a device if the executable does not have enough permissions. These permissions are handled at a Wasm runtime level, and are not specific to Wasm modules. To create a robust and intuitive API, functions that require an 'open' device are scoped to the `device-handle` resource, which can only be obtained if a device can be opened successfully.

The `device-handle` resource will contain all the functions which require communicating with a device. For example, you are able to fetch the active configuration of a device. Functions exist to read and write using the four transfer types (see 2.1.2): `read-interrupt`, `write-interrupt`, etc. Most functions in `device-handle` are closely related to similar libusb functions [22].

```
package component:usb@0.2.0;

interface usb {
  use types.{usb-error};
  use descriptors.{configuration-descriptor, device-descriptor};
```

```
type duration = u64;

resource usb-device {
  // Get a list of all configurations of the USB device.
  configurations: func() -> result<list<configuration-descriptor>, usb-
      error>;

  device-descriptor: func() -> device-descriptor;

  // Open the device. If successfull, this will return a device handle
      which can be used to interact with the device.
  open: func() -> result<device-handle, usb-error>;

  // Get a list of all USB devices the guest is allowed to access.
  enumerate: static func() -> list<usb-device>;
}

resource device-handle {
  reset: func() -> result<_, usb-error>;
  active-configuration: func() -> result<u8, usb-error>;

  select-configuration: func(configuration: u8) -> result<_, usb-error>;

  // Claim an interface.
  // Claiming an interface can fail. For example, the operating system
      might not allow claiming the interface.
  claim-interface: func(%interface: u8) -> result<_, usb-error>;

  // Release an interface.
  release-interface: func(%interface: u8);

  // Select an alternate interface.
  select-alternate-interface: func(%interface: u8, setting: u8) -> result
      <_, usb-error>;

  read-interrupt: func(...) -> ...;
  write-interrupt: func(...) -> ...;

  read-bulk: func(...) -> ...;
  write-bulk: func(...) -> ...;

  read-isochronous: func(...) -> ...;
  write-isochronous: func(...) -> ...;
```

```
    read-control: func(...) -> ...;
    write-control: func(...) -> ...;
  }
}
```

Source Code 4.2: `usb` interface.

`events` interface Code snippet 4.3 shows the `events` interface. This interface allows a guest to poll for connection event updates. This way, a guest can know when a device gets connected or disconnected and act appropriately. It only contains a single function, `update`. As the name implies, this function will return a `device-connection-event`. When no update is available, the `pending` case will be returned.

When WASI Preview 3 brings async support, the `update` function can be adjusted so it can get updates via a stream instead of polling.

```
package component:usb@0.2.0;

interface events {
    use usb.{usb-device};

    variant device-connection-event {
        pending,
        connected(usb-device),
        disconnected(usb-device)
    }

    update: func() -> device-connection-event;
}
```

Source Code 4.3: `events` interface.

4.5 Capability-based security

One of the strenghts of WASI is the capability-based security mechanism it brings. This mechanism allows granular control of what resources an application is allowed to access. For example, the WASI filesystem APIs apply capability-based security, so applications only have access to files and directories they have been given access to [23].

The USB proposal aims to add a similar security model for accessing USB devices. This idea can be applied at various levels:

- Device-level: Only allow devices with specified vendor / product ID pairs.
- Configuration-level: Only allow devices with certain configuration types. For example, only allow self-powered devices.
- Interface-level: Only allow interfaces with certain kinds of properties. For example, only allow interfaces that have the mass storage device class code.
- Endpoint-level: Only allow endpoints with certain kinds of properties. For example, only allow endpoints with an *IN* direction, limiting the guest to only receive data.

The current proposal draft only applies this security model at the **device-level**. This option is the most obvious one, and is similar to the way WebUSB protects devices [24]. Applying the security model to the other three levels is possible, but it is not known yet if this would provide any benefits. Therefore, these levels have been left out of the security model for now. Community feedback will be essential to determine if these options are useful.

4.5.1 Implementation Considerations

When implementing the security model at a device-level, the Wasm runtime should take following measures:

- Extend runtime parameters so users can add allowed devices. These devices would likely be matched against a manufacturer and product ID pair. If proven useful, an extra parameter can also be provided to use a blacklist instead of a whitelist. This way, the user can provide which devices cannot be accessed.
- Change the `usb.{usb-device/enumerate}` function implementation, so that it filters out all non-allowed devices.
- Change the `events.{update}` function implementation, so it does not give update events about non-allowed devices.

If the security model would be applied to any of the three other levels, modification of the `device-handle` resource will be needed, as certain parameters will not be allowed anymore. For example, the `select-configuration`

function accepts any configuration number, while some configurations might not be allowed by the security model. To solve this issue, two solutions are possible:

- All permission-sensitive functions can throw errors when illegal numbers are passed.
- Resources are used instead of numbers to represent configurations, interfaces and endpoints.

The second option would create a more intuitive API, as less errors can be thrown. Instead of passing an arbitrary number, a resource can be passed. This resource can only be created by the host runtime implementation, so the guest cannot pass a resource it does not have access to. A downside of this approach would be that it differs more from the simplicity of the *classic* USB API.

4.6 Usage in multiple components

It is possible that multiple components interact with the USB API and try accessing a USB device at the same time. This should be avoided, as a component generally expects to have exclusive access to a device. To solve this issue, the runtime should keep track of which devices are used by which components, and deny opening a device when another component is already using it. This approach follows the same model as most operating systems, where only one program can have access to a USB device at a time.

5

Implementation

This chapter dives deeper into the internals of the implementation. First, an overview will be given of how a typical series of calls works under the hood. Next, the runtime implementation will be discussed. At last, using the API in a guest component will be explained.

The code of the implementation can be found in the *WASI_USB* repository [25].

5.1 Overview

Figure 5.1 shows the series of calls that happen in a typical, albeit small, use case of the API. The program functionality is the following:

1. Get all devices.
2. Find the device needed.
3. Open the device and select the right configuration / interface.
4. Read data from the device.
5. Close the device.

Under the hood a lot more is happening, which can be seen Figure 5.1. Most calls can be forwarded to libusb, which will handle OS-specific behavior. Some steps, like opening a device, require some extra work in the runtime.

On the left side of the figure are the function calls to libusb shown. On the right side are the calls shown from and to the Wasm guest. The function names are of the format `wit_interface.{function_name_in_interface}`.

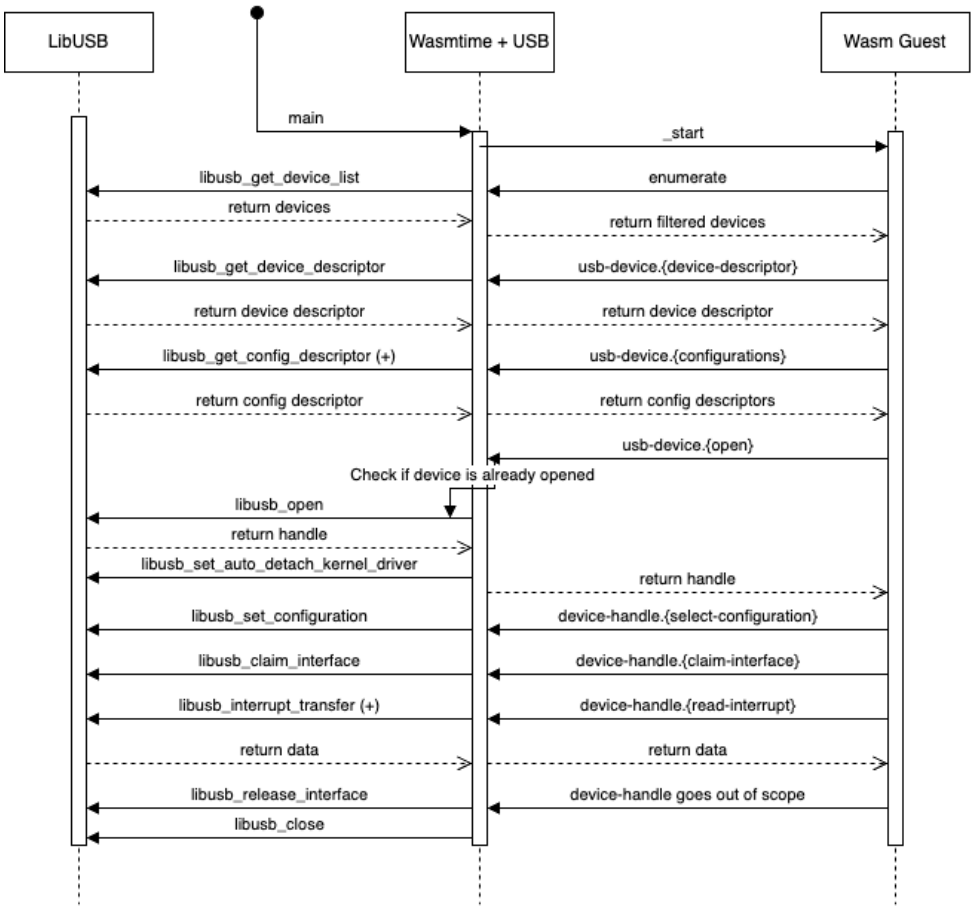

Figure 5.1: Typical series of events sent between Wasm module - Wasmtime - libusb.

5.2 Runtime Implementation

One of the requirements when proposing a new WASI API is to have multiple runtime implementations of that API. This is required in order to advance to phase 3 of a proposal. The USB WASI proposal is still at phase 1. However, having a working implementation of the developing API makes it easier to test out ideas and spot issues quickly. It also is useful to determine the viability of the API. Therefore, a runtime implementation has been created before the proposal process had started.

When creating the PoC implementation, a Wasm runtime must be chosen which will be extended by the API. There are numerous Wasm runtimes available, such as Wasmtime [6], Wasmer [26] or WasmKit [27]. Wasmtime was chosen for implementing the PoC, as it has support for the Component Model and it is developed by the Bytecode Alliance [28], the organization behind WASI.

5.2.1 Choosing the programming language

The implementation is written in Rust [29], a system-level programming language which focuses on performance and safety. Rust's performance is similar to that of C, but its design eliminates entire classes of memory related bugs. Rust was an easy choice, as it has great support for Wasm and WASI, and Wasmtime is also written in it [6].

5.2.2 Interfacing with libusb

As told in Section 2.2.1, libusb will be used to communicate with operating-specific APIs to communicate with USB devices. libusb is written in C [5]. It is possible to interface with C in Rust. To make things easier, the Rusb [30] package is used, which acts as a thin Rust wrapper around the libusb C API.

5.2.3 Interfacing with Wasmtime

Wasmtime can be used in two ways:

- Using the default Wasmtime runtime through the command line. A compiled Wasm module can be passed as an argument:

  ```
  $ wasmtime hello.wasm
  Hello, world!
  ```

- Embedding the Wasmtime runtime in another app. The Bytecode Alliance provides packages in multiple languages, such as Rust or Go, to do this [6]. This is the method used for creating the PoC and is discussed further below.

The PoC runtime embeds the Wasmtime runtime. The method of running is also similar: a compiled Wasm module is passed as a command line argument.

Code snippet 5.1 shows the main function of the runtime. First, the arguments are parsed, which contain the path to the component to run and identifiers of USB devices which are visible to components using the USB API. Next, the component gets instantiated (see code snippet 5.2) and an allowlist of devices is created. Finally, the component gets started.

The main function is annotated with `#[tokio::main]`, meaning that the asynchronous Tokio runtime [31] will be used. Tokio is an asynchronous runtime for Rust, and is not related to the Wasmtime runtime. The use of an asynchronous runtime is done for two reasons:

- The Wasmtime configuration in the runtime has enabled async support. At the time of writing, this isn't a really useful addition [32]. However, this will become more useful once WASI preview 3 lands, which

will add proper async support to the WIT language. Enabling async support now partially prepares for this upcoming feature.

- The runtime receives updates when USB devices are connected and disconnected. This feature runs blocking functions which run on a separate thread. Tokio makes this easy to do.

```
#[tokio::main]
async fn main() -> Result<()> {
    let parsed = UsbDemoAppParser::parse();
    let mut app = UsbDemoApp::new(parsed.component_path)?;

    let allowed_devices = if parsed.usb_use_denylist {
        AllowedUSBDevices::Denied(parsed.usb_devices)
    } else {
        AllowedUSBDevices::Allowed(parsed.usb_devices)
    };

    app.start(allowed_devices).await?
        .map_err(|_| anyhow!("Failed to run component."))
}
```

Source Code 5.1: The main function will start running the guest component.

Code snippet 5.2 shows how the passed in component will get instantiated. The `main` function first calls the `new` function. This function creates some necessary objects:

- `Config`: The configuration used when creating a new `Engine`.
- `Engine`: An engine manages and compiles Wasm modules.
- `Linker`: A linker is used to link components. Each component defines imports and exports. The linker is used to resolve these imports and exports, and throw errors if the required imports cannot be resolved.
- `Component`: Used to represent a WASI component.
- `Store`: A Store is a collection of WebAssembly instances and host-defined state. All WebAssembly instances and items will be attached to and refer to a Store. For example instances, functions, globals, and tables are all attached to a Store. Instances are created by instantiating a Module within a Store. [33]

Next, the built-in components are added to the linker. `wasmtime_wasi::add_to_linker_async` adds all the standard Wasmtime components to the linker, such as `wasi:cli/stdout`. `Imports::add_to_linker`

links the USB WIT interface. Finally, the command component is loaded in and compiled.

The `start` function creates a new `Store` object. This object is associated with the earlier created engine, and contains the memory for an instance of `USBHostWasiView`. Finally, an instance of the guest component is created. We assume that the component is a `Command` (3.2.1) and therefore exposes a `main` function. The `main` function, in WASI terms also known as the `_start` function, gets called.

```
struct UsbDemoApp {
    engine: Engine,
    linker: Linker<USBHostWasiView>,
    component: Component
}

impl UsbDemoApp {
    fn new(component: PathBuf) -> Result<Self> {
        let mut config = Config::default();
        config.wasm_component_model(true);
        config.async_support(true);

        let engine = Engine::new(&config)?;
        let mut linker = Linker::new(&engine);

        wasmtime_wasi::add_to_linker_async(&mut linker)?;
        Imports::add_to_linker(&mut linker, |view| view)?;

        let component = Component::from_file(&engine, component)?;

        Ok(Self {
            engine,
            linker,
            component
        })
    }

    async fn start(&mut self, allowed_devices: AllowedUSBDevices) ->
    ↪  anyhow::Result<Result<(), ()>> {
        let data = USBHostWasiView::new(allowed_devices)?;
```

```
        let mut store = Store::new(&self.engine, data);

        let (command, _) = Command::instantiate_async(&mut store,
        ↪  &self.component, &self.linker).await?;

        command.wasi_cli_run().call_run(store).await
    }
}
```

Source Code 5.2: Code for extending the Wasmtime runtime.

5.2.4 Conforming to the WIT interface

The runtime must conform to the WIT interface described in Section 4.4. This part of the code *receives* requests from guest components using the interface. The communication between the host and a component happens through the canonical ABI [14]. As discussed in Section 3.2.1, bindings can be used to ease the translation from and to the canonical ABI format. Wasmtime offers the `bindgen` macro [34] to generate such bindings in Rust.

The usage of the `bindgen` macro is shown in code snippet 5.3. This macro will generate Rust types and traits that represent their WIT counterparts. The host must conform to all these traits. Otherwise, adding the USB WIT interface to the linker will be disallowed.

The `with` map shown in code snippet 5.3 is used to point the `bindgen` tool to the Rust types we want to use to represent the `usb-device` and `device-handle` resources. The `bindgen` macro will use these types in the generated traits.

```
pub mod bindings {
    wasmtime::component::bindgen!({
        world: "component:usb/imports",
        async: true,
        with: {
            "component:usb/usb/usb-device":
            ↪ crate::device::usbdevice::USBDevice,
            "component:usb/usb/device-handle":
            ↪ crate::device::devicehandle::DeviceHandle,
        },
        path: "../WIT/wit"
    });
}
```

Source Code 5.3: Bindings are generated by using the Wasmtime `bindgen` macro.

Code snippet 5.4 shows an example usage of the generated bindings:

- **`USBHostWasiView`** is the type that implements the USB WIT interface. An instance of this type is passed to the linker to represent the USB interface. The type checker verifies that `USBHostWasiView` conforms to all the required traits.
- **`HostDeviceHandle`** is the trait generated by the bindings. It contains functions which `USBHostWasiView` needs to conform to.
- **`Resource<DeviceHandle>`** is a reference to an `DeviceHandle` instance.

`DeviceHandle` has a handle that is used to call libusb functions. In this example, the `claim_interface` function will call the equivalent libusb function and return its result.

```rust
#[async_trait]
impl HostDeviceHandle for USBHostWasiView {
    async fn claim_interface(&mut self, handle: Resource<DeviceHandle>,
    ↪  interface: u8) -> Result<Result<(), DeviceHandleError>> {
        let result = self.table()
            .get_mut(&handle)?
            .handle
            .claim_interface(interface)
            .map_err(|e| e.into());

        Ok(result)
    }
}

pub struct DeviceHandle {
    pub handle: rusb::DeviceHandle<rusb::Context>
}
```

Source Code 5.4: An example of using generated bindings. An implementation for `claim-interface` is provided.

5.2.5 Conforming to constraints

Section 4.5 and 4.6 mention important constraints regarding the correct implementation of the API. The implementation conforms to the following constraints.

Capability-based security As seen in code snippet 5.1, an allowlist gets created, containing a series of device IDs. A device is identified by the pair `vendor_id:product_id`. These can be passed in as arguments when running the program. An optional flag `--usb-use-denylist` can be given so the program uses a denylist instead of an allowlist.

```
cargo run -- --usb-devices 18d1:9400,18d1:9401 guest-component.wasm
```

Table 5.1 shows the four configurations that are possible. The functions `usb-device.{enumerate}` and `events.{update}` will use this configuration to filter out devices that should not be visible.

	Allowlist	Denylist
Devices Specified	specified devices allowed	all but specified allowed
No devices specified	no device allowed	all allowed

Table 5.1: Device access control options.

Usage in multiple components Exclusive access to devices is guaranteed by keeping track of the device handles. Only one `device-handle` instance may exist for each USB device. `USBHostWastView` stores a set of device addresses for devices with an active `device-handle`. It does not keep track of which component uses which device. When a `device-handle` gets dropped, the device address is removed from the set and becomes available again. When a component tries opening an already-opened device, an error will be thrown.

5.2.6 Platform limitations

The current host implementation works on Linux, macOS and Windows. However, some features work slightly different on some platforms:

- macOS:
  - Using devices for which the kernel provides drivers is disallowed, unless the program has elevated privileges, for example using `sudo` [35].
- Windows:
  - libusb hotplug support is currently missing for Windows. This feature is used in the implementation to get device connection events. Recent activity shows that this might come soon [36].
  - HID keyboards and mice cannot be accessed using the native HID driver as Windows reserves exclusive access to them [37].

5.3 Usage in guest components

This section demonstrates how a WASI component can use the USB API. Parts of the code used for the program in Section 6.1 will be explained.

5.3.1 Defining a WIT interface

The guest component is required to also provide a WIT interface, shown in code snippet 5.5. The interface contains a single `root` world, describing the imports and exports required by the component. The guest component

explained here is a Command component (3.2.1), and therefore does not need to provide any exports. In order to use the USB API, it will import all the interfaces of the USB component.

```
package component:usb-component-wasi-stadia;

world root {
    import component:usb/types@0.2.0;
    import component:usb/usb@0.2.0;
    import component:usb/events@0.2.0;
    import component:usb/descriptors@0.2.0;
}
```

Source Code 5.5: The WIT world for the guest component.

5.3.2 Creating a Wasm module

The code written for the guest component must be compiled to a .wasm file. Doing this can be time consuming, as multiple steps are required:

1. Create bindings.
2. Build a Wasm module.
3. Create a new WASI component and apply the adapter for WASI preview 2.

In order to ease this process, the Cargo-component tool [38] was created. By running

```
cargo component build
```

the tool will interpret the WIT interface, generate bindings for Rust code, and produce a compiled Wasm module. The guest component makes use of this tool to produce a Wasm module with ease.

5.3.3 Calling the USB API

Now that the WIT interface is defined and bindings are created, calling the USB API is straightforward. Code snippet 5.6 shows the `main` function of the guest component. This function will start listening to connection events, and calls the `component:usb/eventsupdate` function to get these events. Next, it will match the event using the `component:usb/eventsdevice-connection-event` variant. As can be seen in the code snippet, these WASI types are translated to Rust types and can be imported into the file. Then, they can be used like regular Rust code.

```rust
use crate::bindings::component::usb::{
    usb::UsbDevice,
    events::DeviceConnectionEvent
};

#[tokio::main(flavor = "current_thread")]
async fn main() -> anyhow::Result<()> {
    loop {
        match update() {
            DeviceConnectionEvent::Pending =>
            ↪ sleep(Duration::from_secs(1)).await,
            DeviceConnectionEvent::Connected(device) if
            ↪ device.is_stadia_device() => {
                // Found stadia controller
                ... // Handle
            },
            DeviceConnectionEvent::Disconnected(device) if
            ↪ device.is_stadia_device() => {
                // Stadia controller is removed
                ... // Handle
            },
            _ => continue
        }
    }
}
```

Source Code 5.6: The WIT world for the guest component.

6

Evaluation

The API has been evaluated in three ways. First, a functional evaluation was performed, checking if the API behaves as expected. Next, a latency evaluation was done, measuring if there are any noticeable slowdowns while using the API compared to native code. Finally, a memory evaluation was constructed, measuring the impact on memory usage of running the code in Wasm compared to native.

The code of all the benchmarks can be found in the *USB_WASI* [25] repository.

6.0.1 Test Setup

This section will go briefly over the test setups used to perform the evaluation. Table 6.1 shows the hardware configurations of the devices used to do these tests. Table 6.2 shows the software environment in which the tests were performed.

Device	Name	SoC/Microcontroller	RAM	Storage	USB Version
Host	Apple Macbook Air (2020)	Apple M1 (8-core CPU, 7-core GPU)	8GB	256GB SSD	4.0
Guest	Arduino Micro	Atmel ATmega32U4	2,5KB	32KB Flash	2.1
Guest	Samsung T5 Portable SSD	Unknown	Unknown	500GB SSD	3.1 Gen 2
Guest	Google Stadia Game Controller	NXP MIMXRT1061	1024KB	16MB	2.0

Table 6.1: The hardware used for testing the performance of the API.

Operating System	macOS 14.5
Wasm Runtime	Wasmtime 20.0.2
USB Library	libusb 1.0.27, Rusb 0.9.4
Rust Version	rustc 1.78.0

Table 6.2: Software Environment.

Figure 6.1: Setup for testing with the Google Stadia controller. The controller is directly connected to the host device with an USB 2.0 C to C cable.

Figure 6.2: Setup for testing with the Arduino board. The Arduino is connected via a USB 3.0 A to C adapter, and a USB 2.0 B to A cable.

Figure 6.3: Setup for testing the Samsung T5 SSD. The SSD is directly connected to the host device with a USB 3.1 Gen 1 (5Gbps) C to C cable.

6.1 Functional Evaluation

When developing the API, it is useful to already have guest code utilizing the API to quickly iterate. The following proof of concept is one of the guest programs created to test the API. It touches on all parts of the API: Getting device connection and disconnection events, reading descriptors from a device, opening a device handle, reading data from the device and writing data to the device. As it is a functional evaluation, it does not test the performance of the API.

The general idea of the program is to control a game controller. The program is started and observes the connected devices. Once a controller is connected that is recognized by the program, the program will connect to the controller. The state of all the controls of the device will be read, and input updates will be print out. To test sending data over the USB interface, the program will send commands to the controller to activate the rumble motors [1]. If the program receives a disconnection event, it will stop reading data and become idle again, waiting for new connections.

Test Methodology

The code will call the event-related API to start watching for USB devices. It will get events when devices are connected or disconnected. As WASI does not support asynchronous code yet [39], a form of polling is used to get device connection events. The guest code uses a single-threaded asynchronous runtime, which will often yield to get new device connection events. This way, a multi-threaded real app can be simulated.

1. Each time a device connection event happens, the device product and vendor ID are checked to see if they match the predefined controller product and vendor ID. A Google Stadia controller is used to test this code, so the IDs of this controller type are used.
2. Make a connection and open a device handle if the device IDs match.
3. Select the correct configuration and interface. Input devices, like controllers or mice, send their data over the Interrupt transfer type, because the data sent is small and time-sensitive. Knowing this information, the interfaces can be requested and filtered to select the correct interface. macOS is used to run this example which offers default drivers for controllers, so elevated privileges are needed to detach the kernel drivers. Otherwise, the program cannot claim the interrupt interface.
4. Read the controller state from the interrupt interface, such as which buttons are pressed, what the position of the joysticks is, etc. An array of bytes is received, and by observing changes to this array when pressing a button, the byte layout can be decoded.
5. Print out the controller state. A sample output is showed in Code snippet 6.1.

[1] Rumble motors are used to vibrate the controller

6. Make the controller vibrate. This happens when the program state reports that one of the shoulder buttons has registered pressure. The state of the shoulder buttons is mapped to intensity of the rumble motors. This data is sent over the interrupt interface. When the controller receives the data it starts vibrating.

7. When a device disconnection event with matching IDs happens, the program will stop reading the controller state, close the device handle, and wait for a new connection.

```
dpad: ,
buttons: assistant_button|l2_button|r2_button,
left stick: x: 128 y: 128,
right stick: x: 128 y: 128,
l2: 255,
r2: 172
```

Source Code 6.1: Output after reading controller state.

Results

This program has been tested with a Google Stadia controller. When pressing one or multiple buttons, the correct state is printed out. When pressing one or both of the shoulder buttons, the controller starts to vibrate.

6.2 Latency Evaluation

One of the advantages of Wasm should be that it offers near-native speed. Therefore, it is interesting to test if running a program in Wasm brings any performance overhead compared to a native program. For an USB API, this can best be tested by measuring the latency when sending or receiving data. It can be problematic if the overhead is large, as data exchange through USB happens by receiving or sending data in small chunks. Each of these chunks has to pass through the Wasm bridge and has its own overhead.

6.2.1 Receiving data from Arduino

Test Methodology

A program has been written with libusb (Native) and WASI USB (Wasm). Each program will do the following steps:

1. Enumerate the USB devices.

2. Find the Arduino board, based on its product and vendor id.

3. Open the device and claim the Bulk interface.

4. Send the Data Terminal Ready (DTR) signal to the Arduino. This signal is sent to the Arduino to acknowledge that the host device is ready to receive data. Once the Arduino has received this signal, its setup phase is completed and it will start sending data.
5. *Warm up* the interface by throwing away the first batch of readings. By doing this, initial latency is removed from the measurements.
6. Read data from the Bulk interface, while measuring how long this operation takes. The Arduino sends data in batches in 64 bytes, and the program will read them a million times. The program will throw away the read results, so only the transfer of the data is measured. In total, 64MB is received by the host.
7. Write durations to file.

Results

Figure 6.4 shows a boxplot of the latencies of both programs. After a million measurements for each program, the median duration is exactly the same, 377µs. The average of the native program is marginally lower, being 0.2% faster than the WASI program. The whiskers and IQR of the WASI program are slightly smaller, meaning a more consistent result. However, the differences in results are too small to conclude any noticeable overhead.

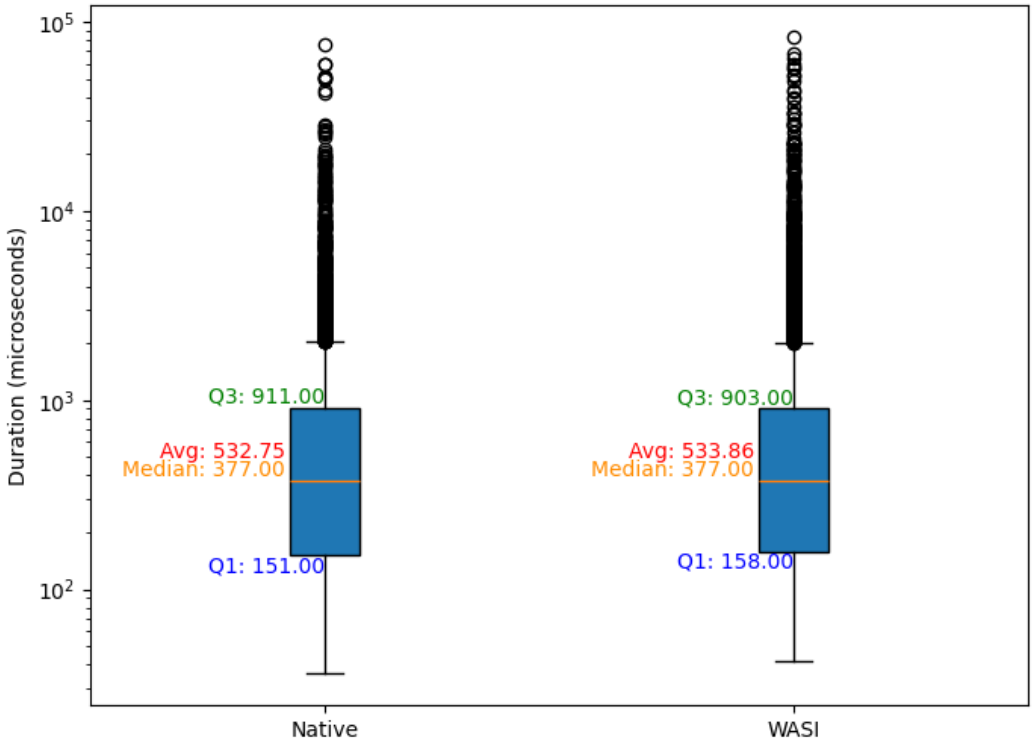

Figure 6.4: Latency of reading data from Arduino using read-bulk.

Figure 6.4 also contains outliers, marked by the black circles. Some of the outliers are very large. Therefore, the results are shown on a logarithmic scale. These outliers are likely caused by transmission errors when receiving data from the Arduino. The *Bulk* interface is used, providing data integrity but no timing guarantees. Consequently, data loss will be prevented, but will introduce high latencies as observed here. Table 6.3 shows more information about the outliers. As the percentage of outliers is very small and occur in equal quantities in both cases, they have been excluded from further figures to improve the clarity of the graphs.

	Native	**WASI**
Largest Outlier (µs)	75618	83052
Amount of Outliers (>2000 µs)	747	1058
Percentage of Total	0.07%	0.10%

Table 6.3: Comparison of Latency Outliers between Native and WASI.

Figure 6.5 shows a Kernel Density Estimation (KDE) for the measurements of the native and WASI implementation.

Both graphs will show peaks around the 150µs and 910µs marks. This is an interesting result, as one would expect one uniform distribution instead of two. The first peak is trivial to explain: the Arduino is an USB 2.0 device, also known as a High-speed USB device. A High-speed USB device will send frames at a fixed interval of 125µs. Therefore, we will receive new data approx. every 125µs. An extra 25µs are introduced because of processing delays.

The second peak is more nuanced. Table 6.4 shows a snippet of the measured latencies. For both Native and WASI code, the latencies will oscillate between both peaks. Based on these results, this peak is likely caused by the small buffer size in the Arduino. The buffer is 64 bytes large, which is the same size as one packet. When the buffer is full, the program halts until the buffer has space again. On the host this is perceived as an extra delay when receiving the data.

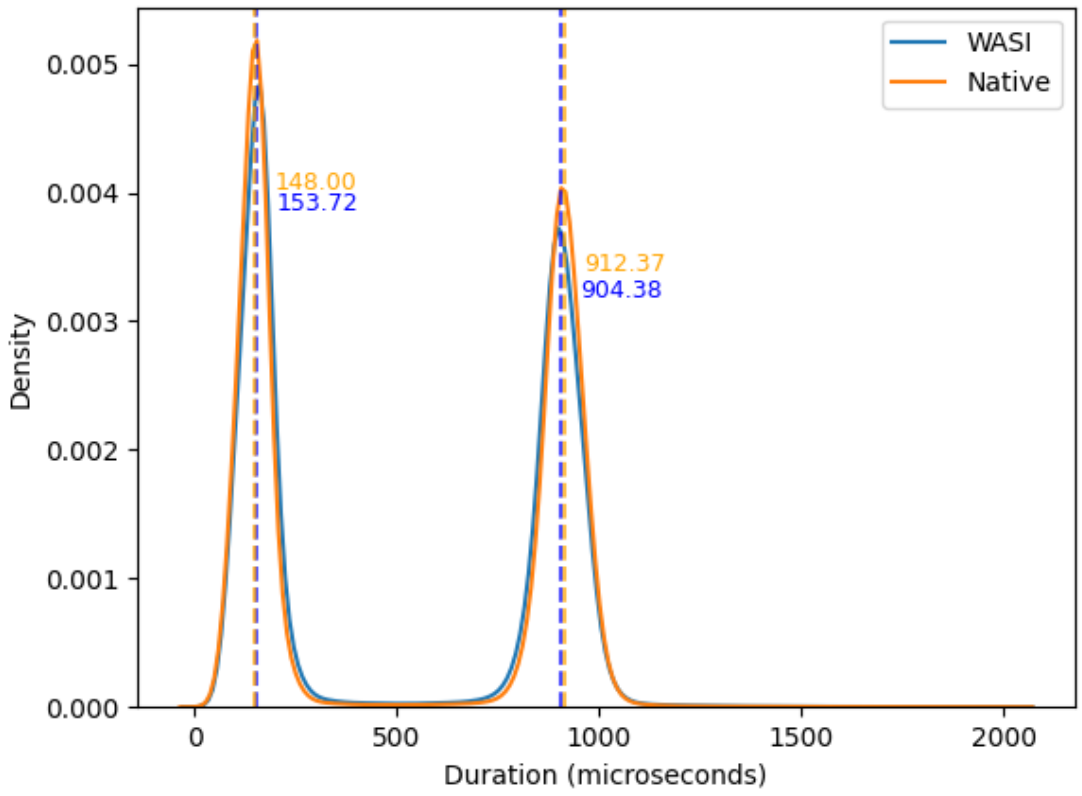

Figure 6.5: Latency of reading data from Arduino using read-bulk.

Based on the results, we can conclude that there are no measurable differences between the native and WASI program. However, this is mainly caused by the delay of the USB protocol. This delay vastly outweighs the delays caused by overhead of the Wasm runtime, making the Wasm overhead negligible.

Native (µs)	WASI (µs)
801	918
150	163
885	931
132	241
903	799
110	197
938	820
104	77
962	1003
108	97

Table 6.4: Snippet of measured latencies. The latencies oscillate between a long latency and a short latency.

However, it is possible that the delay of the USB protocol is smaller on devices more powerful than an Arduino or with a more modern USB version. With these configurations, it can be possible a small overhead for Wasm becomes visible.

6.2.2 Reading files from Mass Storage device

In order to confirm that WASI does not add noticeable latency, another benchmark is performed, which reads the contents of an USB mass storage device. The advantage of this benchmark compared to the Arduino benchmark is that it better represents real-world usage, instead of being a synthetic benchmark.

Test Methodology

A program has been written which enumerates the file tree of a USB device and reads the contents of each file in the file tree. A Samsung T5 External SSD is used to perform these tests. The device is connected with a cable that supports a transfer speed up to 5Gb/s and is formatted with the MBR partition map and the exFAT file system. There are 10 files stored on the device, most of which are a few KBs in size. One file has a size of 679MB. In total, 680MB is stored on the device.

The program contains a working but incomplete implementation for the USB mass storage Bulk-Only transport specification. Further information about the specification and implementation details can be found in the specification documents [40] [41]. The code exposes an interface where one can *seek* to a specific address and *read* the contents from there up until a specified length.

USB API Usage As the name of the specification suggests, communication with the device happens on the interfaces with the *Bulk* transfer type. Some exceptions, such as the `reset` command, will use the *Control* transfer type on interface zero.

Reading drive contents The USB mass storage implementation acts as the base layer to communicate with the device. However, further abstraction is required. USB mass storage devices can still differ in numerous ways:

- Partition Map: A disk is usually split up in multiple partitions. A partition map contains information about those partitions. It is located in the first sector (sector 0) of the device. Popular partition maps are the Master Boot Record (MBR) and the GUID Partition Table (GPT). The test device has a Master Boot Record (MBR) partition map. The Rust package *mbrman* [42] is used to read the MBR and obtain the start and end addresses of the partition which contains the file system.
- File System: The file system organizes files and their metadata in a defined format. There exist a lot of file systems, but the most popular for mass storage devices are File Allocation Table 32 (FAT32), extensible File Allocation Table (exFAT) and New Technology File System (NTFS). The test device uses the exFAT file system. The test program uses the *exfat* [43] Rust package to interpret the exFAT partition.

Both *mbrman* and *exfat* use the mass storage implementation to read data at addresses. Code snippet 6.2 shows how the mass storage implementation is used to read the files.

```rust
fn main() {
    // Open the mass storage device.
    let mut device = MassStorageDevice::new()?;
    let block_length = device.capacity.block_length;

    // Read the MBR to get information about the device partitions.
    let mbr = mbrman::MBR::read_from(device, block_length)?;

    // Select the first used partition.
    let data_partition = mbr.iter().find(|p| p.1.is_used())
        .ok_or(anyhow!("No used partition found"))?.1;

    // Apply a slice to the device stream, so only the selected
    ↪ partition is considered when reading.
    let slice_start = ...
    let slice_end = ...
    let slice = IoSlice::new(device, slice_start, slice_end)?;
```

```
    // Apply buffering to the stream to increase performance.
    let buffered_stream = BufReader::new(slice);

    // Open the exFAT file system.
    let filetree = ExFat::open(buffered_stream)?;

    // Recursively read the device tree.
    for item in filetree {
        read_item(item)?;
    }
}
```

Source Code 6.2: Code to read files and directories from the mass storage device.

With a working implementation to read out the file tree, a benchmark can be performed. The benchmark enumerates the file tree and reads out the contents of each file, and reports each file size and name. The duration of this operation is logged. This is repeated 1000 times, so the file tree will be read out 1000 times. This is done for the program running in Wasmtime using the USB API and a native program using Rusb [30].

Results

Figure 6.8 shows a boxplot with the results of reading the file tree and file contents of the device. With a lower average and median, the WASI program seems to outperform the native program. The average execution time of the native program is 12% slower than the WASI program, the median execution time is 18% slower. This is odd, as one would expect the native version to be faster.

However, there is an explanation for this behaviour. The program will read out the entire contents of the disk, which is in total around 3GB. Each file will be loaded into memory and immediately discarded. The disk contains a few files with a size of over 500MB. Loading these files into memory becomes expensive due to the amount of `malloc`s that need to happen. Profiling the code shows more insights.

The code is compiled in debug mode so debug symbols are added to both programs, so we can interpret the system calls easier. This will lead to a significantly slower running program, especially for the WASI program. This is not a problem as the profiling is only used to measure memory usage and not speed. The sample size is also lowered to 10 samples instead of 1000.

Figure 6.6 shows the total time spent on `malloc` for the native program. 25% of the execution time was spent on allocating memory, which is a significant amount. Figure 6.7 shows that the WASI program only spends 0.6% of its execution time on `malloc` and therefore does not have this issue. This happens because of the way a `Store`

works in Wasmtime: “A Store is intended to be a short-lived object in a program. No form of GC is implemented at this time so once an instance is created within a Store it will not be deallocated until the Store itself is dropped.” [33]. Because no memory is freed, the already-allocated memory can be reused, avoiding extra `malloc` and `free` calls. A Wasmtime `Store` therefore can act as a memory pool, which can be significantly faster than using `malloc` [44].

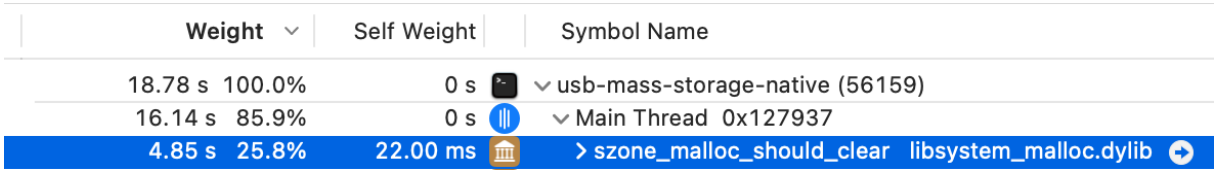

Figure 6.6: Approx. 25% of the native program execution time is spent on `malloc`.

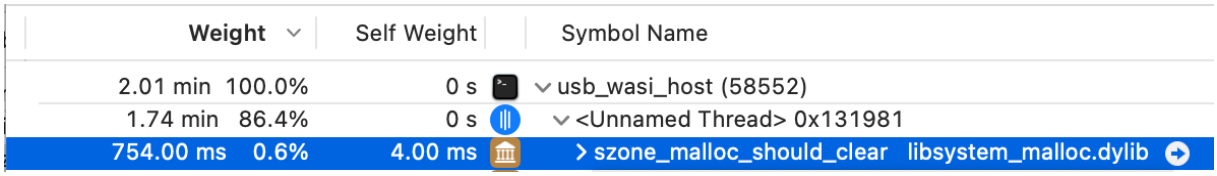

Figure 6.7: Approx. 0.6% of the WASI program execution time is spent on `malloc`.

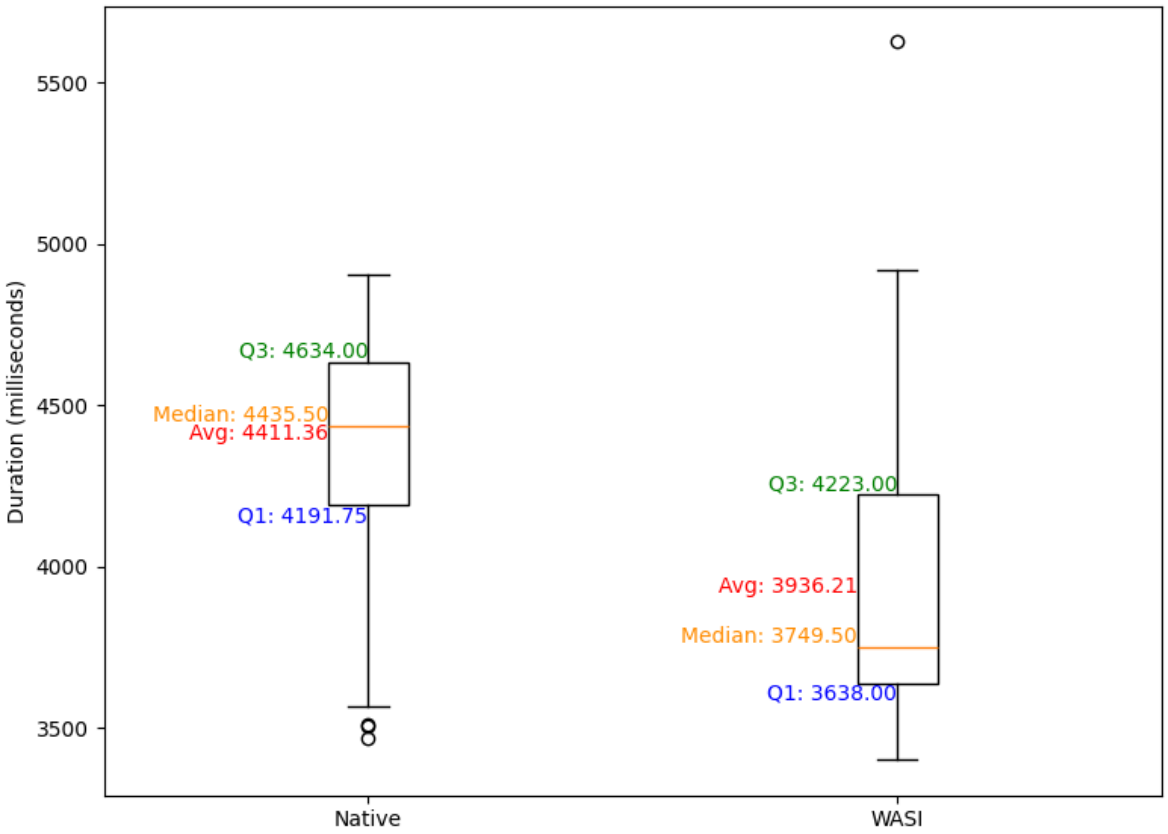

Figure 6.8: Latency of reading the file tree and contents. The WASI results incorrectly report a better result due to differences in memory allocation between both programs. On average, the native program is 12% slower.

Improving results

To make a better measurement at the latency of the USB API, the test has been modified. Instead of the test running multiple times in the guest code, the guest component runs multiple times. Each time, a new `Store` will be created. This will let the memory allocation behave similar to the native program.

Figure 6.9 shows a boxplot with the new measurements. The measurements of the native program are the same as in Figure 6.8. The average WASI program execution time is 4.2% slower than the average native program, and the median 4.3% slower. The WASI program also has more outliers, indicating that the performance of the WASI program may be less consistent.

Conclusion

Based on these results, we can conclude that running the code using the WASI USB API introduces a small amount of overhead. Caution must be exercised when benchmarking memory-intensive Wasm programs to obtain accurate

results.

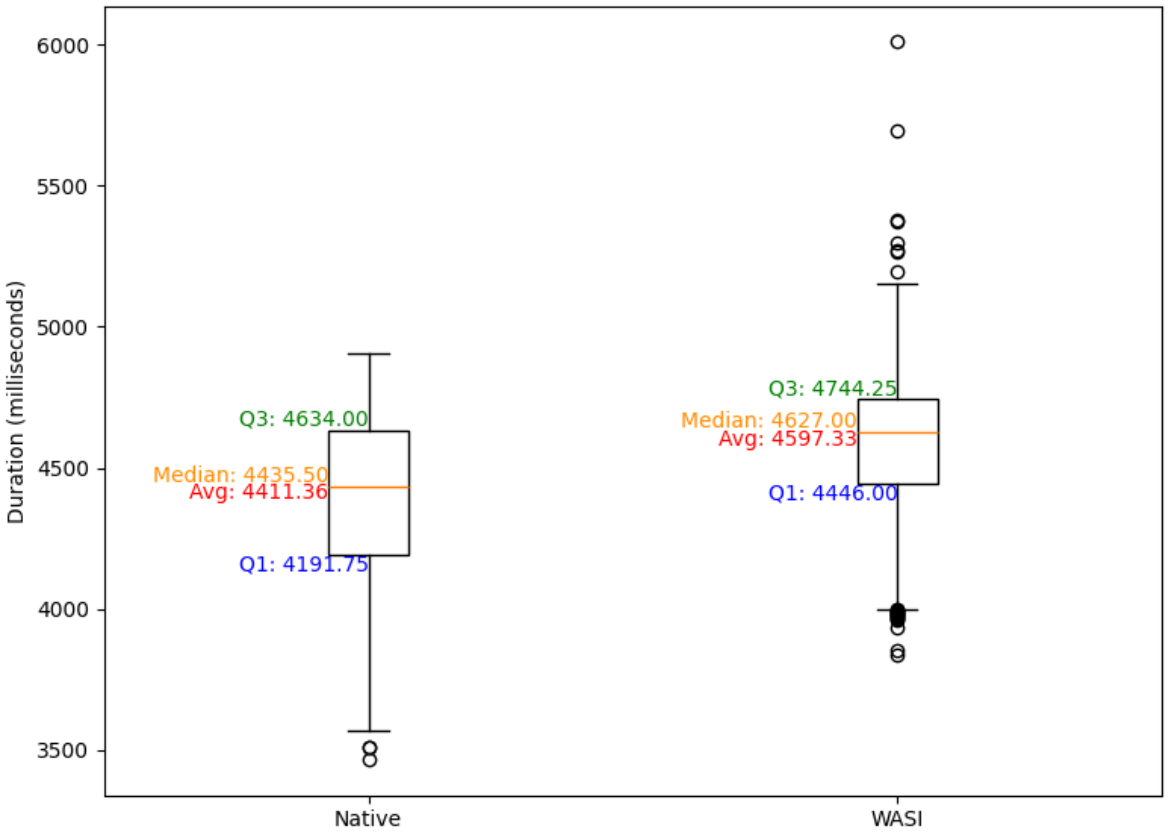

Figure 6.9: Latency of reading the file tree and contents after changing the benchmark method. Both programs now use a similar memory allocation method leading to more accurate results. On average, the WASI USB implementation is approx. 4.2% slower.

6.3 Memory Usage Evaluation

In contrast to standard Wasm modules, WASI components have separate memory spaces. As a result, one component cannot read or write the memory of another component [45]. This improves the security of the Component Model and the interoperability between components, but can also add additional memory usage due to the need of copying memory. Therefore, it is interesting to see how much impact WASI has on the memory usage of a typical program.

6.3.1 Reading files from mass storage device

The same test methodology as Section 6.2.2 will be used to benchmark memory usage. Section 6.2.2 already discussed a performance issue caused by memory allocation. As this section is about the *amount* of memory used, and not about the *performance* of utilizing it, this problem won't be further discussed here.

Test Methodology

Some changes are made to the test methodology described in 6.2.2. Instead of logging the duration of reading the file tree, the amount of memory will be logged. Also, instead of running the benchmark 1000 times, it is now run once.

The method to measure the memory will differ slightly in both implementations. Using a profiler to measure the memory has given incorrect results when benchmarking the WASI implementation. Therefore, no profiler is used for this benchmark. Instead, the memory size is checked numerous times during the execution of the program. This is done getting the *real* memory size of the program every millisecond. For the WASI program this is done in Wasmtime. A baseline memory usage is also measured for the WASI program which gets loaded but does not do anything.

The program will also sleep three seconds before and after opening and closing the device, so the memory usage evolution can be better seen.

Results

Figure 6.10 shows the differences in memory usage for both programs. As both programs consume a lot of memory, the initial Wasmtime memory usage is neglected, as it does not have a large influence on the results. The influence of initial Wasmtime memory is discussed in Section 6.3.2.

After the initial three seconds of waiting time, both programs start traversing the file tree and reading the contents. It is clear that the WASI program uses more memory and with more spikes. As memory is not shared, values are copied over from host to guest, increasing the memory usage. At its peak, it will use 1239MB, while the native program uses 718MB. This is 72.5% more. This peak happens when reading in the largest file of 679MB. As the disk is mostly empty, data is stored contiguously on the disk. This improves reading performance and ensures a steady flow of data at the receiver's end. This is clearly visible in the graph for the native program, as its memory usage follows a linear function. The WASI implementation contains more spikes. These spikes are likely caused by the copying of values, which rapidly allocates and deallocates memory.

After the contents have been read, memory usage will quickly drop for both programs. However, memory usage for the WASI program will decrease slower than the native program. This mainly happens because of the memory-pool-like behavior of the `Store` [33], which will not deallocate memory. Memory used outside the `Store` will

still get deallocated, which is why there is a memory decrease. Once the program ends the memory will get freed further.

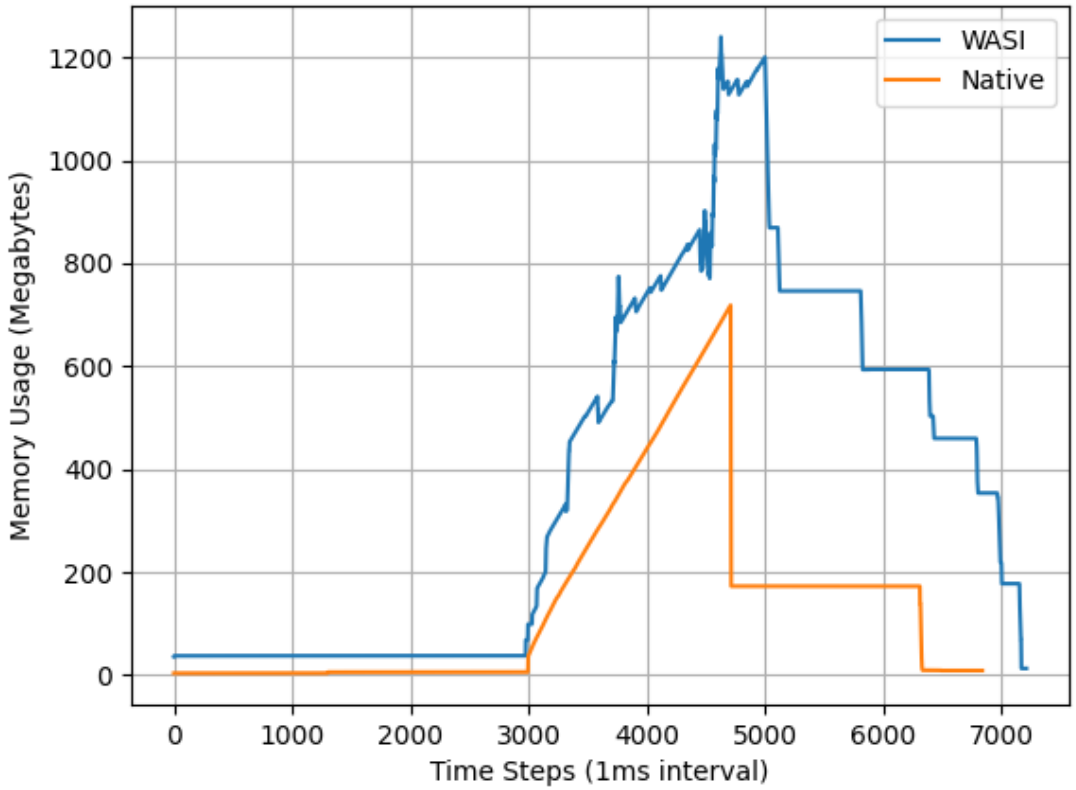

Figure 6.10: Memory usage when traversing the file tree and reading the file contents. At the end of reading the largest file (678MB), the WASI program will consume 72.5% more memory than the native counterpart.

6.3.2 Traversing file tree from mass storage device

A less memory-intensive benchmark was also performed to check how WASI memory usage would compare when using less memory. This benchmark uses the same test methodology as Section 6.3.1, except that it will not read out the file contents. The file tree has also changed and now contains 11947 files, with a total size of 3.25GB.

Results

Figure 6.11 shows the results of the benchmark. The graph contains the results of the total memory usage of both programs, and also contains the WASI results without the initial memory occupied by Wasmtime. This is a useful metric, as a program utilizing the USB API can be part of a larger piece of code, where the initial memory overhead of Wasmtime gets *distributed* between all components and can be neglected.

The first 3000 measurements show the memory usage from when the program is sleeping for three seconds, as described in Section 6.3.1. The memory of both native and WASI programs are similar, not accounting for the initial Wasmtime memory usage. Furthermore, the WASI memory usage is slightly better at the start, but this is likely caused by applying the correction. Once the useful program code starts, memory spikes for both programs. However, the WASI (corrected) program consumes approx. 63MB of memory, 69.8% more than the native's 37.1 MB memory usage.

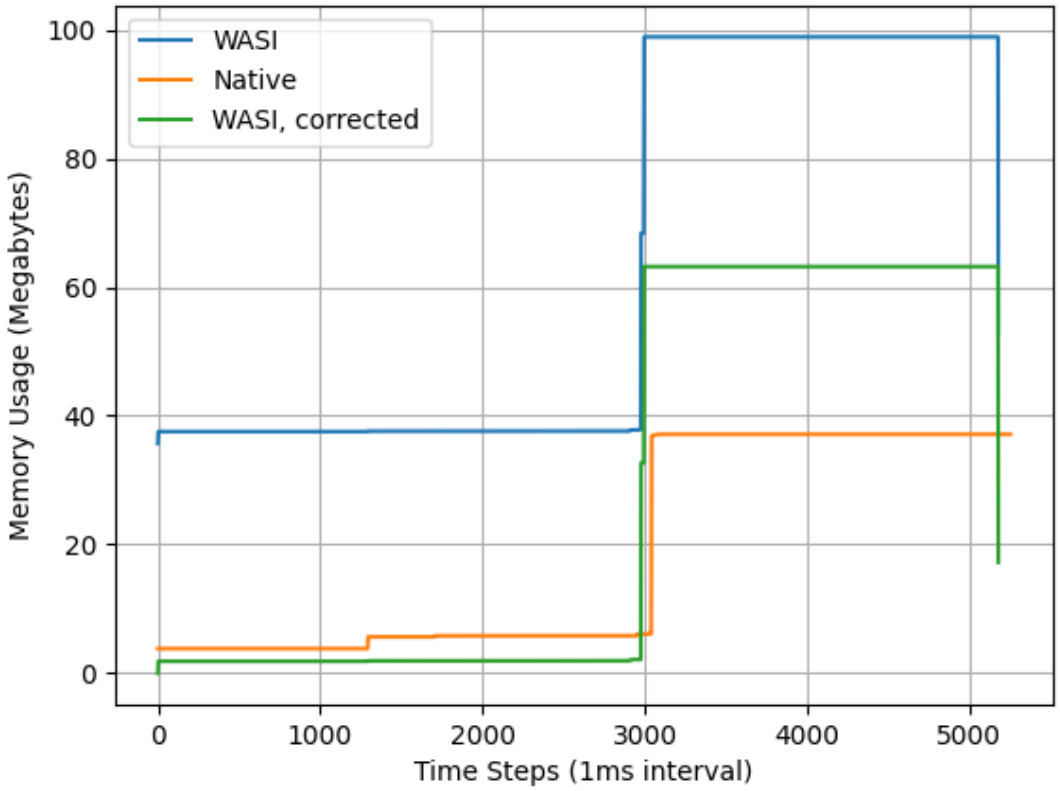

Figure 6.11: Memory usage when traversing the file tree. The WASI program consumes 69.8% more memory than the native counterpart.

7

Conclusion

The WebAssembly System Interface (WASI) is rapidly progressing. While still in development, the initial vision of the people behind the Bytecode Alliance is coming to life and will hopefully get to a stable state in the coming years. This masters thesis has done research on the possibilities of utilizing USB devices in WASI and will hopefully make a positive impact on the eventual support of it.

The proposal that was developed over the year, together with the initial working implementation in Wasmtime, has dug through most of the questions and issues that one would expect when developing such an API. While the proposal still has a long way to go, it has laid out the fundamentals of the API which can be expanded further upon.

One of the most prevalent questions was how access control, one of the prime features of WASI, could be integrated in the API. After trying out multiple possibilities, it was concluded that limiting access control on a device-level is deemed enough. The idea is also worked out in the implementation and functions as expected. The proposal also contains directions for adding access control to other parts of the interface, should this be needed.

Multiple API designs were tested out, one leaning more to what libusb offers, while the other leaning more into WebUSB. After trying out both APIs and getting feedback from the community, an API which closely follows libusb was chosen. This API is more aligned with how USB devices work internally. This also makes it easier for existing programs using libusb to port their code over to WASI USB.

Performance of the API has been thoroughly tested and results are overall positive. WebAssembly (Wasm) is fast enough to not have notable performance issues when using the USB API, especially when compared to the latency of communicating with an external device. However, due to the isolated memory of WASI components, data sent from and to a device needs to be copied, bringing in memory overhead. This can become a problem for applications that use a lot of memory.

8

Future Work

Research done for this thesis has laid the groundwork for building a WASI USB API. However, the proposal is still at phase 1 and has a long way to go before it can become stable. Further work will need to happen on testing, providing multiple runtime implementations, creating fully functional programs that utilize the API and gaining further community feedback.

Also, the API currently has some shortcomings that will need to be addressed in the future:

- The proposal currently ignores language support, but USB devices can provide certain data, like device names, in multiple languages.
- Windows hotplug support is currently missing, making the `events` interface of the proposal currently unavailable for Windows. As discussed in Section 5.2.6 this could be solved soon.

References

[1] P. Ray, "An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions," *Future Internet*, vol. 15, p. 275, 08 2023.

[2] WebAssembly, "Webassembly/wasi-usb." [Online]. Available: https://github.com/WebAssembly/wasi-usb

[3] "Wasi: Proposals." [Online]. Available: https://github.com/WebAssembly/WASI/blob/main/Proposals.md

[4] "Webusb," Nov 2023. [Online]. Available: https://wicg.github.io/webusb/

[5] "libusb." [Online]. Available: https://libusb.info/

[6] "Wasmtime." [Online]. Available: https://wasmtime.dev/

[7] A. Alexandrov, A. the authors Asen Alexandrov Staff Engineer at OCTO Daniel Lopez Sr. Director at OCTO, A. A. S. E. a. OCTO, and D. L. S. D. a. OCTO. [Online]. Available: https://wasmlabs.dev/articles/docker-without-containers/

[8] T. Brito, P. Lopes, N. Santos, and J. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," 04 2022.

[9] "Usb in a nutshell." [Online]. Available: https://www.beyondlogic.org/usbnutshell/usb1.shtml

[10] "Google stadia." [Online]. Available: https://stadia.google.com/controller/

[11] M. Posch, "Reverse-engineering the stadia controller bluetooth switching procedure," Dec 2023. [Online]. Available: https://hackaday.com/2023/12/19/reverse-engineering-the-stadia-controller-bluetooth-switching-procedure/

[12] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[13] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 4, jan 2008. [Online]. Available: https://doi.org/10.1145/1328195.1328197

[14] "Canonical abi." [Online]. Available: https://component-model.bytecodealliance.org/advanced/canonical-abi.html

[15] bytecodealliance, "The component model docs." [Online]. Available: https://github.com/bytecodealliance/component-docs/blob/74beecf57c9012f6db48ba4f9b18865a4f2798c6/component-model/src/design/wit.md

[16] "Existential types." [Online]. Available: https://wiki.haskell.org/Existential_type

8 References

[17] D. Gohman, "Wasi 0.2 launched," Jan 2024. [Online]. Available: https://bytecodealliance.org/articles/WASI-0.2

[18] L. Wagner, "The path to components," Oct 2022. [Online]. Available: https://www.youtube.com/watch?v=phodPLY8zNE

[19] W. Hennen, "Proposal: Wasi usb api · issue #570 · webassembly/wasi." [Online]. Available: https://github.com/WebAssembly/WASI/issues/570

[20] B. Hayes, "meetings/wasi/2024/wasi-04-18.md at main · webassembly/meetings." [Online]. Available: https://github.com/WebAssembly/meetings/blob/main/wasi/2024/WASI-04-18.md

[21] WebAssembly, "Wit specification." [Online]. Available: https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md#wit-interfaces

[22] "libusb docs." [Online]. Available: https://libusb.sourceforge.io/api-1.0/group__libusb__dev.html

[23] "Filesystem access security." [Online]. Available: https://docs.wasmtime.dev/security.html#filesystem-access

[24] R. Grant, K. Rockot, and O. Ruiz-Henríquez, "Webusb api." [Online]. Available: https://wicg.github.io/webusb/#abusing-a-device

[25] Wouter01, "Wouter01/usb_wasi." [Online]. Available: https://github.com/Wouter01/USB_WASI

[26] "Wasmer." [Online]. Available: https://wasmer.io/

[27] "Wasmkit." [Online]. Available: https://github.com/swiftwasm/WasmKit

[28] "Bytecode alliance." [Online]. Available: https://bytecodealliance.org/

[29] "Rust." [Online]. Available: https://www.rust-lang.org/

[30] "Rusb." [Online]. Available: https://docs.rs/rusb/latest/rusb/

[31] "Tokio." [Online]. Available: https://tokio.rs/

[32] [Online]. Available: https://docs.rs/wasmtime/20.0.2/wasmtime/struct.Config.html#method.async_support

[33] "wasmtime::store." [Online]. Available: https://docs.rs/wasmtime/20.0.2/wasmtime/struct.Store.html

[34] "wasmtime::component::bindgen." [Online]. Available: https://docs.wasmtime.dev/api/wasmtime/component/macro.bindgen.html

[35] libusb, "Faq." [Online]. Available: https://github.com/libusb/libusb/wiki/FAQ#how-can-i-run-libusb-applications-under-mac-os-x-if-there-is-already-a-kernel-extension-installed-for-the-device-and-claim-exclusive-access

8 References

[36] "windows: hotplug implementation by sonatique." [Online]. Available: https://github.com/libusb/libusb/pull/1406

[37] "Windows." [Online]. Available: https://github.com/libusb/libusb/wiki/Windows#known-restrictions

[38] bytecodealliance, "Github - bytecodealliance/cargo-component: A cargo subcommand for creating webassembly components based on the component model proposal." [Online]. Available: https://github.com/bytecodealliance/cargo-component?tab=readme-ov-file

[39] B. Hayes, "Webassembly: An updated roadmap for developers," Jul 2023. [Online]. Available: https://bytecodealliance.org/articles/webassembly-the-updated-roadmap-for-developers

[40] USB-IF, "Universal serial bus mass storage specification for bootability," Oct 2004. [Online]. Available: https://usb.org/sites/default/files/usb_msc_boot_1.0.pdf

[41] "Universal serial bus mass storage class, bulk-only transport." [Online]. Available: https://www.usb.org/sites/default/files/usbmassbulk_10.pdf

[42] "mbrman package, rust." [Online]. Available: https://docs.rs/mbrman/latest/mbrman/

[43] "exfat package, rust." [Online]. Available: https://docs.rs/exfat/latest/exfat/

[44] C. to Wikimedia projects, "Memory pool," Aug 2023. [Online]. Available: https://en.wikipedia.org/wiki/Memory_pool

[45] "Why the component model?" [Online]. Available: https://component-model.bytecodealliance.org/design/why-component-model.html

Appendices

USB Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of the Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	USB Specification Number which device complies to
4	bDeviceClass	1	Class	Class Code. If equal to Zero, each interface specifies its own class code. If equal to 0xFF, the class code is vendor specified. Otherwise, the field is a valid Class Code.
5	bDeviceSubClass	1	SubClass	Subclass Code
6	bDeviceProtocol	1	Protocol	Protocol Code
7	bMaxPacketSize	1	Number	Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64
8	idVendor	2	ID	Vendor ID
10	idProduct	2	ID	Product ID
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manufacturer String Descriptor
15	iProduct	1	Index	Index of Product String Descriptor
16	iSerialNumber	1	Index	Index of Serial Number String Descriptor
17	bNumConfigurations	1	Integer	Number of Possible Configurations

Table 1: Device Descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor describing this configuration
7	bmAttributes	1	Bitmap	D7 Reserved, set to 1. (USB 1.0 Bus Powered) D6 Self Powered D5 Remote Wakeup D4..0 Reserved, set to 0.
8	bMaxPower	1	mA	Maximum Power Consumption in 2mA units

Table 2: Configuration Descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (9 Bytes)
1	bDescriptorType	1	Constant	Interface Descriptor (0x04)
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Value used to select alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used for this interface
5	bInterfaceClass	1	Class	Class Code
6	bInterfaceSubClass	1	SubClass	Subclass Code
7	bInterfaceProtocol	1	Protocol	Protocol Code
8	iInterface	1	Index	Index of String Descriptor Describing this interface

Table 3: Interface Descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	Endpoint Address. Bits 0..3b Endpoint Number. Bits 4..6b Reserved. Set to Zero. Bits 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoints)
3	bmAttributes	1	Bitmap	Bits 0..1 Transfer Type. 00 = Control, 01 = Isochronous, 10 = Bulk, 11 = Interrupt. Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronisation Type (Iso Mode). 00 = No Synchronisation, 01 = Asynchronous, 10 = Adaptive, 11 = Synchronous. Bits 5..4 = Usage Type (Iso Mode). 00 = Data Endpoint, 01 = Feedback Endpoint, 10 = Explicit Feedback Data Endpoint, 11 = Reserved.
4	wMaxPacketSize	2	Number	Maximum Packet Size this endpoint is capable of sending or receiving
6	bInterval	1	Number	Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Isochronous must equal 1 and field may range from 1 to 255 for interrupt endpoints.

Table 4: Endpoint Descriptor.