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.

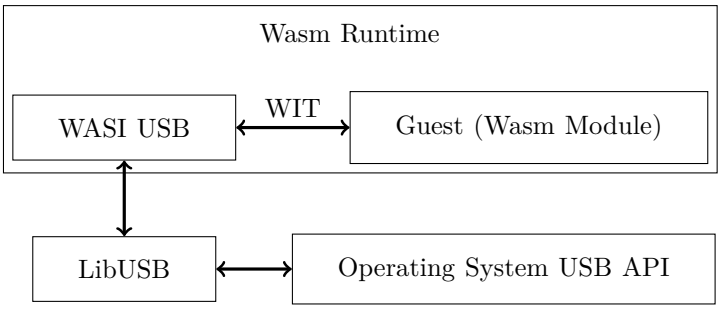

Fig. 1. Internals of the WASI USB API. The arrows represent communication between the different areas.

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

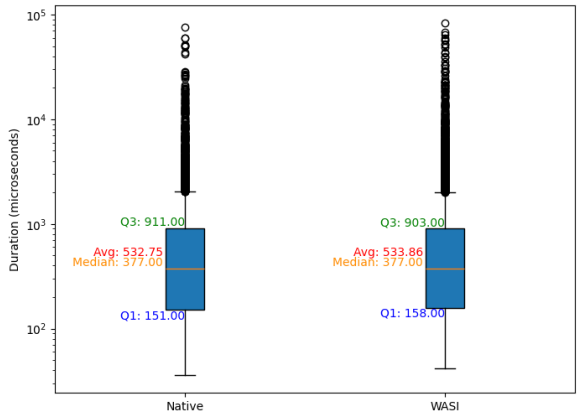

Fig. 5. Boxplot of latency of reading data from Arduino using read-bulk.

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.

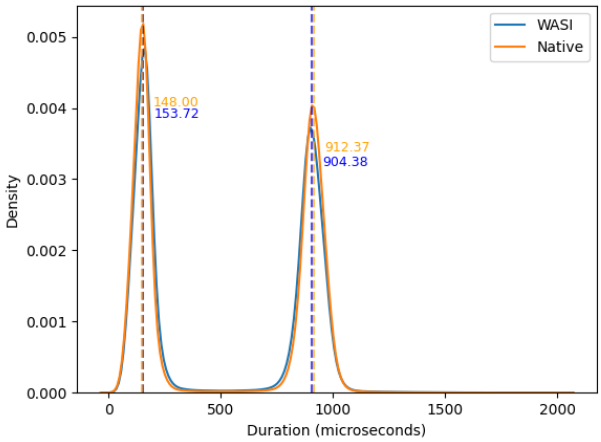

Fig. 6. KDE of latency of reading data from Arduino using read-bulk.

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

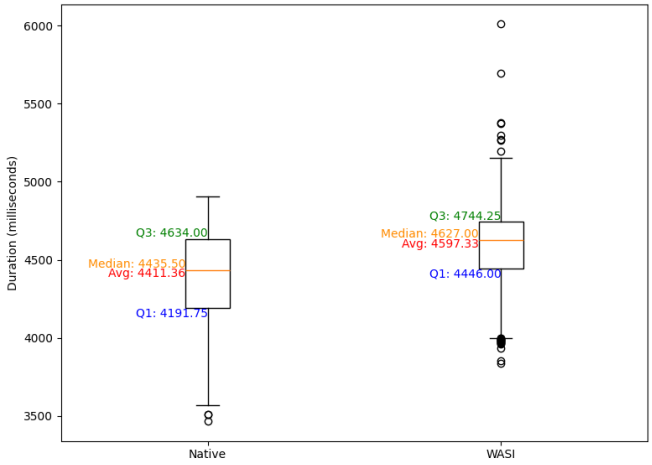

Fig. 7. Latency of reading the file tree and contents after changing the benchmark method. On average, the WASI USB implementation is approx. 4.2% slower.

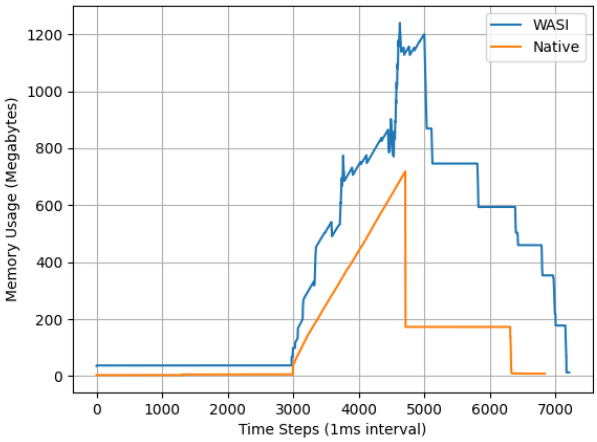

Fig. 8. Memory usage when traversing the file tree and reading the file contents. At the end of reading the largest file (678MB), the WASI program will consume 72.5% more memory than the native counterpart.

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

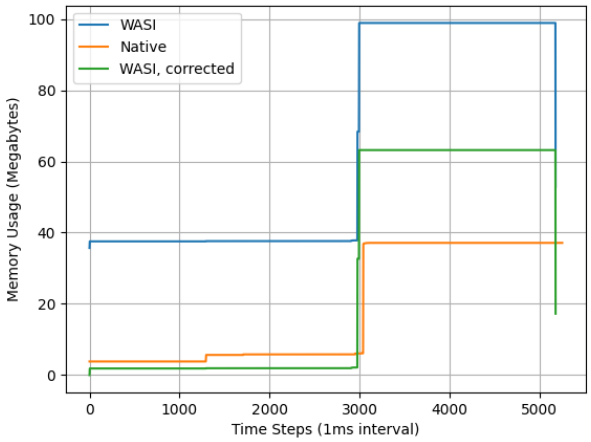

Fig. 9. Memory usage when traversing the file tree. The WASI program consumes 69.8% more memory than the native counterpart.

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