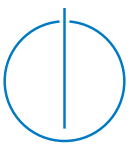
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of WebAssembly IoT Runtimes on a ESP32 Microcontroller

Lukas Heddendorp



DEPARTMENT OF INFORMATICS

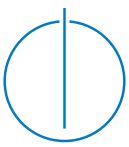
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluation of WebAssembly IoT Runtimes on a ESP32 Microcontroller

Evaluation von WebAssembly IoT Runtimes auf einem ESP32 Microcontroller

Author:	Lukas Heddendorp
Supervisor:	Teemu Kärkkäinen
Advisor:	Advisor
Submission Date:	16.03.2020



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.03.2020 Lukas Heddendorp

Acknowledgments

Abstract

Contents

1 Introduction

2 Background

2.1 Microcontrollers

Since this thesis is focused on the use of WebAssembly on microcontrollers, we would like to introduce the concept and limitations shortly. A microcontroller (**MCU** for microcontroller unit) is a small computer, meant to fulfill a very specific requirement without a complex operating system. They are designed for embedded applications from implantable medical devices to toys and very prominently in IoT devices. Devices will often have multiple microcontrollers, each responsible for a particular function. A car, for example, could include, amongst others, an MCU to control the mirror adjustments, one to handle fuel injection and another one for traction control.

Core elements of an MCU are the processor, memory, and I/O peripherals. The Processor (CPU) can be thought of as the brain of the MCU. It performs basic arithmetic, logic, and I/O operations. The memory is where any data is stored the processor needs to fulfill its tasks. Mainly there is program memory, which holds the MCUs instructions and Data memory, which servers as temporary storage while a program is executed. Lastly, the I/O peripherals are the controller's connection to the outside world; they allow the receiving and sending of information, such as receiving a signal from a switch and turning on a light in response.

2.1.1 ESP32

For this thesis, we want to specifically focus on the ESP32 system on a chip (SoC). A very popular low-cost, low power series of microcontrollers with integrated WiFi and Bluetooth. Developed by the Shanghai-based company Espressif Systems this successor to the ESP8266 offers a great platform for IoT and embedded projects. Compared to the MCU described before, the ESP32 has the additional processing power and I/O options that make it a great platform for developing secure IoT devices. It gained popularity fast after being released in September of 2016.

The ESP32 systems family provides an excellent base for many IoT applications. There are multiple versions available from ones very well suited for hobbyists to ones usable for industrial manufactures. With a low price point and area footprint, they still provide significant performance and many operational features.

To reach price and power consumption targets, the ESP32 has significant hardware limitations. This introduces some constraints when working with the platform. The Operating system for example can hardly be called that. Compared to popular operating systems like windows and linux, the used FreeRTOS can be thought of as a thread library. This specialty will be further explained in the following passage.

2.1.2 FreeRTOS

Many MCUs are used in applications where throughput is less important than a guaranteed performance. This is why the ESP32 uses FreeRTOS, a real-time operating system (RTOS) is specifically intended to be used in time-critical situations. A key characteristic of such an operating system is the predictable behavior of the scheduler, the part of the operating system that decides which task should be run by the CPU at any given time. Most schedulers allow the user to set priorities for tasks in order to decide which task should be run next.

FreeRTOS specifically is the leading RTOS amongst MCUs and is designed to be small enough to run on a microcontroller. Since most applications in which MCUs are used do not warrant the use of a full RTOS, FreeRTOS only provides the core scheduling functionality, timing, and synchronization primitives. It can, however, be extended by using addon components, for example, to make use of a specific networking stack. FreeRTOS also built a significant community and support for many platforms in its 15-year development.

2.2 WebAssebly

Beginning with static HTML pages, the web has since developed into a common application platform, accessible from many different devices running different operating systems. JavaScript is the only natively supported language on the web. However, even though it is universally used and made a lot of progression modern implementations, it still has some problems as a compilation target. WebAssembly addresses these issues and provides a compilation target for the web.

WebAssembly (WASM) was first announced in June 2015 and reached a cross-browser consensus in March 2017. Its goal was to provide near-native performance for browser-based applications, which could only be written in JavaScript for a long time. Since being published in March 2017, it is currently usable for about 90% of global internet users. More recently, the interest picked up around usage outside of the browser, which is also the primary concern of this thesis.

Being meant for the web, WASM has to achieve specific goals that give the platform new properties. It has to be safe since, on the web, code is loaded mainly from untrusted

sources. It has to be fast as the primary motivation to introduce WebAssembly was to provide a compile target on the web with reliable performance. Other than the usual low-level code such as regular assembly, WebAssembly has to be portable and work in all the different circumstances the web is currently used. Lastly, because the code is transmitted over the network, it has to be as small as possible to reduce bandwidth and improve latency.

WASM is a low-level binary format, designed to be a portable target for high-level languages like C
C++ or Rust. It is executed on a stack-based virtual machine on which it executes in near-native speed due to its low-level design. Still, it runs in a memory-safe environment inside the browser and is subject to the same security policies as JavaScipt code would be. WebAssembly modules are loaded with the application and provide bindings to JavaScript that make them usable in the browser.

Together with the binary format of WebAssembly, there is a text format that defines a programming language with syntax and structure. Every WASM binary is a self-contained module with functions, globals, tables, imports, and exports. This concept provides both encapsulation and sandboxing since modules can only interact with their environment using imports, and the client can only access the specified exports. Inside the module, the code ins organized in functions that can call each other even recursively.

2.2.1 WebAssembly for IoT

While WASM is developed for the web, it carefully avoids any dependencies on the web. It is meant to be an open standard that can be embedded in a variety of ways. The goals mentioned above, which WebAssembly achieves, make it an exciting format to explore on embedded devices. Due to it's aim to be universal it would allow the use of languages on MCUs that where not previously supported and since it is already meant to be transmitted over the network, also over the air updates of code running on the controller are possible. To achieve portability, the source level interface libraries would have to map the host environments' capabilities either at build time or runtime.

While WebAssebly in an IoT context is a very promising concept it is also a brand new development. The best support for WASM right now is in the browser of course but out of browser runtimes keep surfacing, implmented in various langugaes and providing an interesting execution environment. Runtimes meant to be used on MCUs are much more rare and not as mature yet. Given the big interest in the idea though and the working groups avoidance of web depnedencies it can be assumed that this situation will change in the future.

2.2.2 Interpreters

WIth our specific usecase in mind, the WASM3 engine was chosen since it's sepcific goal is to run WebAssembly on MCUs. Other than many other engines it does not follow a jsut in time compilation pattern though, but instead acts as an interpreter. Thus we'd like to shortly introduce the concept and advantages.

Interpreters are computer programs that execute a program. They pose a different concept to compiled execution, where a program would be translated to machine code before being run directly on the CPU. While offering multiple advantages, the main drawback is the execution speed compared to native code execution, which is often slower by order of magnitude and sometimes more. The overhead is generated by the interpreter having to analyze the program code before it can be executed.

Interpreters thus offer benefits in development speed since the code does not have to be recompiled to run and in portability because the same code could be run on multiple platform-specific interpreters without the need to compile it into the native machine code of multiple platforms. For our use-case, the interpreter executes WASM instructions, allowing the dynamic loading of modules and running them in the chose environment.

2.3 Microbenchmarking

Benchmarking is any form of measurement to qualify the behavior of a system. The most obvious examples would be measuring performance, energy or memory consumption, but also relibality and temeprature stability could contribute to a benchmark. Building good bechmarks is hard, because a program has to be created that yields repeatable and consinstent results. A mejor effort in the world of microcontroller benchmarking is the EEMBC.

The embedded microprocessor banchmark consortium is an industry association that has been designing benchmarks for over 20 years. The consortium offers multiple benchmarks, all meant to cover sepcific usecases od embedded controllers from ultra-low power IoT applications, over processor performance measurements to a recent benchmark designed to assess machine learning performance.

3 Methodology

3.1 Finding a Runtime

As WASM needs an environment to run in, a runtime had to be found for this project as well. Most WASM runtimes these days exist in web-browsers, but with interest in other use cases emerging, more possibilities to run WASM outside the browser are becoming available. For this project, though, it was important to find a small runtime that could be executed on the ESP32. While there are more and more WASM runtimes outside of browsers available, most of them are meant to run on full-sized computers, do not offer a way of embedding, or are not written in C//C++. These requirements make them unusable on Microcontrollers.

3.1.1 Selection of WASM3

WASM3 is a somewhat new effort to run WASM outside of the browser with the specific goal to allow execution on microcontroller platforms as well. It is also written entirely in C and allows embedding. According to benchmarks run by the creators, comparing them to other WASM runtimes, WASM3 performs about 4-5x slower than the best just in time engines and about 12x slower than native code execution. WASM3 also uses the interpreter model instead of a just in time compilation method, which would increase execution speed. However, there are tradeoffs of executable size memory usage and startup latency, which make the interpreter approach a better fit for our use case.

3.2 Designing Tests

To assess the viability of running WebAssembly on an MCU, there had to be some comparison. For this, we decided to design multiple performance tests that measure common use-cases in both native and WASM execution. The cases we tested are matrix multiplication, random and linear memory performance, recursive calculation, switch statements, and native calls. The last test is interesting in this context since the runtime provides a way to call outside functionality from within the WASM module. This allows for actual communication of the WASM code with the outside environment.

3.3 Running tests

All tests where run on the ESP-WROOM-32 using the IDF provided by espressif. The same code was compiled to WebAssembly and also imported into the test program to allow for native execution. Then the test code was run multiple times to spot inconsistencies between the runs. The results from these tests will be explained in more detail in a specific section for each test.

3.3.1 Testing setup

All tests share a very similar main program to execute and time the tests, which we would like to explain now.

Listing 3.1: Main testing method

```
1  extern "C" void app_main(void) {
2      int64_t start_setup = esp_timer_get_time();
3      setup_wasm();
4      int64_t end_setup = esp_timer_get_time();
5      int64_t wasm_times[100];
6      int64_t native_times[100];
7
8      for (long long &wasm_time : wasm_times) {
9          int64_t start_time = esp_timer_get_time();
10         for (int j = 0; j < 10; ++j) {
11             run_wasm("20");
12         }
13         int64_t end_time = esp_timer_get_time();
14         wasm_time = (end_time - start_time) / 10;
15     }
16
17     for (long long &native_time : native_times) {
18         int64_t start_time = esp_timer_get_time();
19         for (int j = 0; j < 10; ++j) {
20             long value = run(20);
21         }
22         int64_t end_time = esp_timer_get_time();
23         native_time = (end_time - start_time) / 10;
24     }
25
```

```
26      printf("\nWasm3 v" M3_VERSION " on ESP32, build " __DATE__ " " __TIME__ "\n");
27      printf("Setup time: %lld\n", (end_setup - start_setup));
28      printf("|Run|WASM|NATIVE|\n|---|---|---|\n");
29      for (int i = 0; i < sizeof(wasm_times) / sizeof(wasm_times[0]); ++i) {
30          printf("|%d|%lld|%lld|\n", i + 1, wasm_times[i], native_times[i]);
31      }
32      sleep(100);
33      printf("Restarting...\n\n\n");
34      esp_restart();
35  }
```

The main testing method starts with setting up the WebAssembly runtime, which will be further explained with Listing 3.2. This process is timed to see how much overhead the runtime initialization introduces. Next, there are two arrays set up to hold the test results. Then the test is run for the WASM with the function described in listing 3.3, by taking the average time over ten runs and saving that into the previously declared array. Following, the same is done for the native version of the test code. Lastly, the results are printed to the console, and the controller is eventually restarted to rerun the tests.

Listing 3.2: Runtime setup

```
1  IM3Environment env;
2  IM3Runtime runtime;
3  IM3Module module;
4  IM3Function f;
5
6  static void setup_wasm() {
7          M3Result result = m3Err_none;
8
9          auto *wasm = (uint8_t *) wasm_test_cpp_wasm;
10         uint32_t fsize = wasm_test_cpp_wasm_len - 1;
11
12         env = m3_NewEnvironment();
13         if (!env) FATAL("m3_NewEnvironment failed");
14
15         runtime = m3_NewRuntime(env, 2048, NULL);
16         if (!runtime) FATAL("m3_NewRuntime failed");
17
18         result = m3_ParseModule(env, &module, wasm, fsize);
19         if (result) FATAL("m3_ParseModule: %s", result);
20
```

```
21          result = m3_LoadModule(runtime, module);
22          if (result) FATAL("m3_LoadModule: %s", result);
23
24      // result = LinkThesis(runtime);
25      // if (result) FATAL("LinkThesis: %s", result);
26
27          result = m3_FindFunction(&f, runtime, "run");
28          if (result) FATAL("m3_FindFunction: %s", result);
29      }
30  }
```

setting up the runtime is pretty straight forward, initially, the WASM module is imported from a header file, together with its length. Then the environment and runtime are created, followed by parsing the module. If the runtime has to provide functions that the WASM module relies upon, they are linked after loading the module. An example can be found in section 3.3.6 in which lines 24 and 25 of listing 3.2 are not commented out. Once the runtime is fully set up, the function itself is searched for. In our case, the functions name is always "run".

Listing 3.3: WASM execution

```
1  static void run_wasm(const char *input1) {
2      M3Result result = m3Err_none;
3
4      const char *i_argv[3] = {input1, NULL};
5      result = m3_CallWithArgs(f, 1, i_argv);
6
7      if (result) FATAL("m3_CallWithArgs: %s", result);
8
9      long value = *(uint64_t *) (runtime->stack);
10 }
```

The execution of the WASM function is a matter of calling the previously found function with the runtimes `m3_CallWithArgs()` method and supplying it with the input arguments. The return value of the operation can be found on the virtual machines stack afterward.

3.3.2 Testing matrix multiplication

To test performance during matrix multiplication, then function takes one argument, the matrix size. It then creates two nxn matrices and multiplies them. In the end, it

returns one value of the resulting matrix. This is to prevent the compiler from deleting the actual calculation during compilation.

Listing 3.4: Matrix multiply test

```
uint32_t run(uint32_t n) {
    uint32_t a[n][n], b[n][n], mul[n][n];

    for (uint32_t i = 0; i < sizeof(a) / sizeof(a[0]); ++i) {
        for (uint32_t j = 0; j < sizeof(a[0]) / sizeof(a[0][0]); ++j) {
            a[i][j]=i+1;
            b[i][j]=i+2;
        }
    }

    for (uint32_t i = 0; i < sizeof(a) / sizeof(a[0]); ++i) {
        for (uint32_t j = 0; j < sizeof(a[0]) / sizeof(a[0][0]); ++j) {
            mul[i][j] = 0;
            for (uint32_t k = 0; k < sizeof(a[0]) / sizeof(a[0][0]); ++k) {
                mul[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return mul[n-1][n-1];
}
```

Running this function in the main program described in section 3.3.1 results in the Following measurements for the average execution times of 100 runs.

Run	WASM exectuion	native exectuion
1	26283	285
2	26277	281
3	26277	282
4	26277	281
5	26277	281
6	26277	282

As is evident from the measurements, the interpreter introduces a 90x slowdown of the WASM execution compared to the native one.

3.3.3 Testing memory performance

3.3.4 Testing recursive calls

The test of recursive calls is run by calculating Fibonacci numbers with the following code.

Listing 3.5: Recursive calling test

```
uint32_t run(uint32_t n) {
    if (n < 2) {
        return n;
    }
    return run(n - 1) + run(n - 2);
}
```

Listing 3.6: WASM code excerpt

```
(module
  (type $t1 (func (param i32) (result i32)))
  (func $run (export "run") (type $t1) (param $p0 i32) (result i32)
    (block $B0
      (br_if $B0
        (i32.lt_u
          (local.get $p0)
          (i32.const 2)))
      (return
        (i32.add
          (call $run
            (i32.add
              (local.get $p0)
              (i32.const -1)))
          (call $run
            (i32.add
              (local.get $p0)
              (i32.const -2))))))
    (local.get $p0)))
```

Listing 3.6 shows the WebAssembly text format generated for the Fibonacci function in listing 3.5 and we would like to take a closer look at what the WASM module looks like for this specific example. After the module opening, the type of our `uint32_t run(uint32_t n)` function is defined and reused in the function definition

in line 3, in this line the functions input and return types are also defined. The input is assigned to the $p0 variable for later use. In line 4 the block `$B0` is started, it contains the main function body. In line 5, we can see a `br_if` instruction; this is a conditional break that breaks the execution of the passed block if the condition is true. The condition, in this case, is the rest of the instructions included in the parentheses. Namely the comparison of the accepted parameter with 2 to see if it is smaller if that is the case the remaining block is skipped and code execution would continue in line 19 where the parameter is pushed on the stack, as the topmost value of the stack after the execution is the return value of a WASM function. Alternatively, the execution could continue in line 9 with the `return` instruction, which executes the instructions inside the parentheses and prevents any further code execution after that, mirroring the common return instruction in C. the Value return is the result of the two recursive calls of line 5 in the C listing. The `run` function is called again by using the `call` instruction in lines 11 and 15. It is important to note that this text format is not strictly WebAssembly, but one version to make it readable for humans. To make it more similar to the look of common programming languages for this listing, code folding was applied. To make the difference visible, listing 3.7 shows the WASM code without folding.

Listing 3.7: WASM code without folding

```
1  (func $run (export "run") (type $t1) (param $p0 i32) (result i32)
2    block $B0
3      local.get $p0
4      i32.const 2
5      i32.lt_u
6      br_if $B0
7      local.get $p0
8      i32.const -1
9      i32.add
10     call $run
11     local.get $p0
12     i32.const -2
13     i32.add
14     call $run
15     i32.add
16     return
17   end
18   local.get $p0)
```

3.3.5 Testing switch statements

Listing 3.8: Switch statement test code

```
uint32_t run(uint32_t n) {
    uint32_t array1[20] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    uint32_t result = 200;
    for (int i = 0; i < n; ++i) {
        uint32_t compare = i % 20;
        switch (compare) {
            case 0:
                result = array1[0];
                break;
            case 1:
                result = array1[1];
                break;
            case 2:
                result = array1[2];
                break;
            case 3:
                result = array1[3];
                break;
            case 4:
                result = array1[4];
                break;
            case 5:
                result = array1[5];
                break;
            case 6:
                result = array1[6];
                break;
            case 7:
                result = array1[7];
                break;
            case 8:
                result = array1[8];
                break;
            case 9:
                result = array1[9];
```

```
36                 break;
37             case 10:
38                 result = array1[10];
39                 break;
40             case 11:
41                 result = array1[11];
42                 break;
43             case 12:
44                 result = array1[12];
45                 break;
46             case 13:
47                 result = array1[13];
48                 break;
49             case 14:
50                 result = array1[14];
51                 break;
52             case 15:
53                 result = array1[15];
54                 break;
55             case 16:
56                 result = array1[16];
57                 break;
58             case 17:
59                 result = array1[17];
60                 break;
61             case 18:
62                 result = array1[18];
63                 break;
64             case 19:
65                 result = array1[19];
66                 break;
67             default:
68                 result = 100;
69                 break;
70         }
71     }
72     return result;
73 }
```

3.3.6 Testing native calls

A very important function of the runtime is to expose outside functions to the WASM module and allow the interaction with other libraries from within the WASM code. Fpr this we designed two failry simple tests that call functions not defined in the WASM code.

Listing 3.9: Outside call test code

```
1 #include "test_api.h"
2
3 WASM_EXPORT
4 void run(uint32_t n) {
5     mark();
6 }
```

As is obvious from listing 3.9, the test code just calls the outside `mark()` function. There is also an import of the test api header in which the external function is defined to make the testcode compile.

Listing 3.10: Test api defintiion

```
1  #ifndef WASM3_TEST_API_H
2  #define WASM3_TEST_API_H
3
4  #include <stdint.h>
5
6  #define WASM_EXPORT extern "C" __attribute__((used)) __attribute__((visibility ("default")
7  #define WASM_EXPORT_AS(NAME) WASM_EXPORT __attribute__((export_name(NAME)))
8  #define WASM_IMPORT(MODULE, NAME) __attribute__((import_module(MODULE))) __attribute__((im
9
10 extern "C" {
11
12 WASM_IMPORT("thesis", "sendValue") uint32_t sendValue (void);
13 WASM_IMPORT("thesis", "mark") void mark (void);
14
15 }
16
17 #endif //WASM3_TEST_API_H
```

The resulting WASM code does not include the `mark` method but inmstead imports it from the `thesis` module that is expected to be avalialbe at runtime.

Listing 3.11: WASM code for the outside call

```
1 (module
2   (type $t0 (func))
3   (type $t1 (func (param i32)))
4   (import "thesis" "mark" (func $thesis.mark (type $t0)))
5   (func $run (export "run") (type $t1) (param $p0 i32)
6     (call $thesis.mark)))
```

As displayed in listing 3.11 line 4 the mark function from the thesis module is imported as defined in listing 3.10. The run function then just calls the imported function in line 6.

In order to provide this importet functionat runtime the setup for opur tests has to be changed slightly, namely it has to be linked druing the runtime setup in listing 3.2.

Listing 3.12: Function linking

```
1  int64_t native_timestamp;
2
3  m3ApiRawFunction(m3_thesis_mark) {
4      native_timestamp = esp_timer_get_time();
5      m3ApiSuccess();
6  }
7
8  M3Result LinkThesis(IM3Runtime runtime) {
9      IM3Module module = runtime->modules;
10     const char *thesis = "thesis";
11
12     m3_LinkRawFunction(module, thesis, "mark", "i()", &m3_thesis_mark);
13     return m3Err_none;
14 }
15
16 void mark() {
17     native_timestamp = esp_timer_get_time();
18 }
```

In listing 3.12 line 1 we introduce a varibel to hold a timestamp after the `mark` function was called. In line 3 we define the function, which just saves the current timestamp and ends with success. This is then linked into the runtime in line 12. To compare netive exectuion this time we can not call the exact same function, since it was not compiled to WASM at all, so we implement a similar function in line 16 that is called during the test of native exectuion.

4 Benchmarking

5 Conclusion

List of Figures

List of Tables