FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KIV/OPSWI
Dynamic Software Image Update on ESP32

Petr Kocián
kocian@students.zcu.cz
(A21N0032P)

February 23, 2024

Contents

1 Assignment **1**

2 Introduction **1**

3 Tools **2**
3.1 WebAssembly 2
3.2 ESP32 2
3.3 WebAssembly Micro Runtime 2
3.4 WebAssembly System Interface 2

4 Implementation **3**
4.1 C++ to WebAssembly compilation 3
4.2 Running WebAssembly on ESP32 3

5 Results **4**

6 Conclusion **4**

1 Assignment

- Familiarize yourself with ESP32
- Study the article: OTTA, M. Towards a Health Software Supporting Platform for Wearable Devices, Procedia Computer Science, Volume 210, pp. 112-115, 2022
- Explore different ways to compile C++17 and newer into WebAssembly
- Explore possible ways to run WebAssembly modules on ESP32, including dynamic uploading and freeing from memory of WebAssembly modules
- Experimentally verify the points mentioned above
- In case the suggested solution does not work, propose an alternative solution of updating the software image with regard to safety against failure

2 Introduction

The goal of this project is to evaluate the feasibility of the general concept proposed by Otta M. in [1]. Otta M. proposes an architecture for the ESP32 board based on the ESP-IDF framework [2] in which WebAssembly modules [3] can be remotely deployed, dynamically loaded, and executed. The architecture is intended for the SmartCGMS framework [4], which is written in C++17, therefore, this project focuses on C++17 to WebAssembly compilation.

The following sections will introduce the tools used for implementation, explain the details of the implementation, and discuss the achieved results.

3 Tools

This section briefly introduces technologies used in the project

3.1 WebAssembly

WebAssembly is a binary format that was originally developed for web browsers, but it is also supported by now-web runtime environments (e.g. WebAssembly Micro Runtime [5]). It allows the compilation of programs written in different languages (e.g. C++, Rust, C#) into WebAssembly binaries, called modules. These are then executed in virtual stack machines. WebAssembly's low overhead and portability were the reasons why it has been selected as the format in Otta's architecture proposal [1]. [6]

3.2 ESP32

ESP32 is a microcontroller unit with a focus on low-power consumption and connectivity (Wi-Fi and Bluetooth) [7]. The manufacturer of ESP32 (Espressif) provides a software development framework called Espressif IoT Development Framework (ESP-IDF) [2]. The ESP-IDF framework provides an easy way to compile and upload code to the ESP32 and it has also been selected by Otta M. for the architecture concept development [1]. Because of these reasons, it has also been used for the implementation and testing of this project.

3.3 WebAssembly Micro Runtime

WebAssembly Micro Runtime (WAMR) is Bytecode Alliance's open-source standalone WebAssembly runtime. It has a small footprint of around 85 kilobytes in interpreter mode. It provides support for WASI - WebAssembly System Interface [8]. WAMR's WASI support, its small footprint, and its support for ESP-IDF were the reasons it was chosen as a WebAssembly interpreter for this project. [5]

3.4 WebAssembly System Interface

WASI is a WebAssembly API designed to provide standardized access to system resources. It aims to make WebAssembly more independent of browsers and thus it helps bring WebAssembly closer to embedded devices. [8]

4 Implementation

This section describes the solution implementation and decisions as to why certain technologies were used.

4.1 C++ to WebAssembly compilation

There are multiple ways to compile C++ to WebAssembly. One option is to use Emscripten [9], but it is a toolchain focused mainly on the Web platform. As this project is focused on embedded systems, this option wasn't explored deeply.

Instead, WASI-SDK [10] with Clang was used. The WASI-SDK repository contains clear instructions on how to install WASI-SDK and how to use it. The official repository contains a CMake toolchain file to integrate WASI-SDK with a CMake build system or a Docker image with WASI-SDK and all its requirements preinstalled. There are limitations to C++ compilation as WASI and WASI-SDK are in constant development, for example, thread support is only experimental, networking is not yet supported and neither are C++ exceptions. Consequently, C++ has to be compiled with -fno-exceptions flag. [10]

4.2 Running WebAssembly on ESP32

Again, there are more interpreters with official support for ESP32. The two most popular are WAMR [5] and Wasm3 [11]. Wasm3 is currently in a state of minimal maintenance, therefore, WAMR was used for this project implementation.

As this project's focus was on WebAssembly compilation and interpreters, UART was used for dynamically uploading WebAssembly modules to the ESP32. The decision to use UART was based on its ease of implementation and usage.

The final execution model involves two threads: a UART listener and a WebAssembly interpreter thread. The interpreter thread manages the WebAssembly resources, it loads and instantiates the WebAssembly runtime and then executes it. The WebAssembly module is provided by the UART thread that stores the module in a shared byte array. Whenever the UART thread receives a new module, it signals it to the interpreter thread, which then destroys the currently running WebAssembly module and unloads it from memory. After the old WebAssembly module is destroyed it loads, instantiates, and runs the newly received WebAssembly module.

5 Results

The ESP32 implementation was verified by compiling two simple C++ programs that each print a different message to the standard output. A simple program to upload data to UART was written and then both WebAssemly modules were uploaded after each other to the ESP32 during one run of the program.

The program successfully executed both WebAssembly modules. Currently, the implementation only supports WebAssembly modules, that have single-pass execution. No signaling has been implemented between the interpreter and the Webassembly module. WAMR offers a way to share a buffer between the interpreter and the WebAssembly module. This could be used to communicate between a running WebAssembly module and the interpreter to stop the module's execution since the WebAssembly module cannot be destroyed and unloaded from memory until it finishes its execution.

A big limitation of the current implementation is memory usage. The ESP32 DRAM size is 320 kilobytes. 160 kilobytes can be only statically allocated. There remains 160 kilobytes for runtime heap allocation. This limits the size of the programs that can be compiled to WebAssembly for ESP32.

6 Conclusion

While there are still limitations to compiling C++ to WebAssembly, there is ongoing development aimed at both the toolchains and the system interface. Even in the current state, it is possible to compile and run C++ programs as WebAssembly modules on ESP32 microcontrollers. As WebAssembly continues to evolve, there might be support for more platforms, enabling the execution of more memory-intensive programs as WebAssembly modules on embedded devices.

References

[1] M. Otta. "Towards a health software supporting platform for wearable devices". In: *Procedia Computer Science* 210 (2022), pp. 112–115. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2022.10.126. URL: https://www.sciencedirect.com/science/article/pii/S1877050922015836.

[2] E. Systems. *ESP-IDF: Espressif IoT Development Framework*. Espressif Systems. URL: https://github.com/espressif/esp-idf.

[3] *WebAssembly Core Specification*. Version 2.0. W3C, Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/.

[4] T. Koutny and M. Ubl. "Parallel software architecture for the next generation of glucose monitoring". In: *Procedia Computer Science* 141 (2018), pp. 279–286. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.10.197. URL: https://www.sciencedirect.com/science/article/pii/S1877050918318507.

[5] B. Alliance. "WebAssembly micro runtime". In: *GitHub, accessed: February* 15 (2020).

[6] 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". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

[7] Espressif Systems. *ESP32 Overviewl*. Accessed: 24/1/2024. 2024. URL: https://www.espressif.com/en/products/socs/esp32.

[8] *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/. (Accessed: 2024-02-22).

[9] *emscripten*. URL: https://emscripten.org/. (Accessed: 2024-02-22).

[10] *WASI-SDK*. URL: https://github.com/WebAssembly/wasi-sdk. (Accessed: 2024-02-22).

[11] *Wasm3*. URL: https://github.com/wasm3/wasm3. (Accessed: 2024-02-22).