Memorial University of Newfoundland
Faculty of Engineering and Applied Science

ENGI-981B
Computer Engineering Project
Progress Report

Safe Software for Embedded Systems

February 12, 2025

Supervisor: Dr. Jonathan Anderson

Trong Nguyen
Student ID: 202387819
tdnguyen@mun.ca

Yuxuan Ma
Student ID: 202384266
yuxuanm@mun.ca

Zhou Ziyang
Student ID: 202193950
zzhou21@mun.ca

I. INTRODUCTION

Embedded system is a combination of computer hardware and software designed for a specific function and embedded in a large system or a completed device. It is usually a microcontroller-based system and requires strict constraints about the resources, performance, reliability, security, cost, and so forth so that it does not downgrade the performance of larger systems where it is embedded. Embedded systems are widely used in various areas, from automobiles and industrial machines to mobile phones, IoT devices, medical devices, and personal fitness devices.

With the widespread use of embedded applications in IoT devices and industrial embedded systems, as well as the connection to the Internet 24/7, the cyber security concern has increased rapidly in recent years. Therefore, the change from native embedded software written in C programming language to safe software software solutions has gained significant attention in this field.

As proposed in part A of the project, our group is implementing safe software solutions for embedded systems, including WebAssembly (Wasm) and Lua approaches. The project takes place in three months of the Winter 2025 semester, and we are focusing on the following implementation and evaluation.

- Implement Wasm application for image processing to embedded systems.
- Implement Lua application for image processing to embedded systems.
- Compare footprints of Wasm, Lua, and native.
- Evaluate the performance of Wasm, Lua, and native.

In addition, we will investigate the running of a lite machine learning model as a Wasm application in an embedded system in the last phase of the project.

The progress report is organized into eight sections. Section II provides a literature survey about Wasm runtimes, Lua runtimes, and literature on their application to embedded systems. Section III introduces the methodology of the project. Sections IV and V present the hardware platforms and design details, respectively. The project plan and milestones are updated in section VI. Finally, sections VII and VIII summarize the questions, problems encountered, and future work in order.

II. LITERATURE SURVEY

In this section, we provide additional background information about Wasm and Lua runtimes and a comparison between them so that readers can understand and follow the technical details of the project. It is noted that the

A. Wasm Runtimes

Wasm [1] is platform-independent, safe, and fast because it is an assembly-like language executed in Wasm runtimes, a sandboxed environment. Wasm runtimes must adapt to the WebAssembly core specification published by the World Wide Web Consortium (W3C), and as per our team survey, Wasm runtimes are now available in almost all popular platforms and system architectures for embedded systems.

In our project, we considered six Wasm runtimes that support Wasm applications outside the Web, including wasmtime [2], wasmer [3], WAVM [4], WasmEdge [5], WAMR [6], and Wasm3 [7]. In this list, only WAMR and Wasm3 support embedded systems, so our options were reduced to two, and we selected WAMR for several reasons. At first, WAMR is an open-source project managed by the Bytecode Alliance, a nonprofit organization focused on Wasm and WASI with more than twenty members, including big-tech like Intel, ARM, Microsoft, Amazon, Mozilla, Anaconda, and Siemens, while Wasm3 is a project supported by Wasm community and currently under minimal maintenance. Therefore, when a new Wasm specification is published, WAMR will adapt to new specifications quicker than Wasm3. In addition, WAMR supports all running modes, including interpreter, Just-in-time (JIT), and Ahead-of-time (AoT), whereas wasm3 supports interpreter mode only.

WAMR (WebAssembly Micro Runtime) [6] is a lightweight standalone Wasm runtime with a small footprint (normally less than 60KB), high performance (near-native speed by AOT and JIT), and is applied widely in embedded, IoT, edge, Trusted Execution Environment (TEE), smart contract, cloud-native, and so on. WAMR uses the `Apache 2.0 license`, and its structure consists of three main parts: VMCore, the main part of this runtime including a set of runtime libraries for loading and running Wasm modules in the interpreter, JIT, and AoT modes; iwasm, the executable binary built with WAMR VMcore; and wamrc, the AoT compiler to compile Wasm file into AoT file. In addition, WAMR supports almost all popular architectures and platforms nowadays for embedded systems, including our project's hardware platforms: RISCV-32 architecture with ESP-IDF and bare-metal environment:

- Architectures: X86-64, X86-32, ARM, THUMB, AArch64, RISCV64, RISCV32, XTENSA, MIPS, ARC.
- Platforms: Linux, Linux SGX (Intel Software Guard Extension), MacOS, Android, Windows, Windows (MinGW, MSVC), Zephyr, AliOS-Things, VxWorks, NuttX,

RT-Thread, ESP-IDF(FreeRTOS), and be able to integrate to a new platform like the bare-metal environment.

B. Lua Runtimes

Lua [8] is a powerful, efficient, lightweight, embeddable scripting language that is successfully applied in various fields, including embedded systems, like the Ginga middleware for digital TV in Brazil, games, like Warcraft, and industrial applications, like Adobe's Photoshop Lightroom. In addition, it has been proved that writing applications in Lua are normally faster than the native code in C, and this can separate the application development from the underlying back end and avoid some vulnerabilities of writing code in C.

Unlike Wasm, there is no big organization that manages the development of Lua runtime for different platforms and architectures for embedded systems. We investigated some popular Lua runtime or frameworks, including LuaJIT [9], eLua [10], NodeMCU [11], LuaNode [12], LuatOS [13], and Lua RTOS [14]. We only select the runtime that supports ESP32C6 or its family since porting a runtime to a new target platform requires a lot of manual effort and LuaNode is the only runtime choice we currently have for this platform. Therefore, we selected LuaNode for implementation and LuatOS and Lua RTOS for reference.

- LuaNode: supports ESP32C6.
- LuatOS and Lua RTOS: do not support ESP32C6 but ESP32C3, the platform that use the same RISC-V architecture, so these frameworks have the potential for implementation on ESP32C6.

C. Application of Wasm and Lua in Embedded Systems

Wasm, initially designed for web browsers, has been successfully adapted for embedded systems through specialized runtimes such as WAMR and wasm3. These runtimes provide the necessary infrastructure for executing WebAssembly modules on bare metal hardware while maintaining the benefits of memory safety and sandboxing. Current implementations have demonstrated that Wasm can achieve near-native performance while providing a secure execution environment, which is particularly crucial for IoT and edge computing applications. Several major embedded platforms, including ESP32 and various RISC-V or ARM-based systems, now offer Wasm support through these runtimes.

Compared to wasm, Lua has established itself as a mature solution in the embedded space, with implementations like LuaJIT, eLua, NodeMCU, LuaNode, LuatOS, and Lua RTOS specifically optimized for microcontroller environments. Compared to WebAssembly, Lua offers excellent performance characteristics and a smaller runtime footprint, but its security model differs significantly. Traditional Lua implementations prioritize performance and flexibility over sandboxing, some modern

implementations have introduced optional security features, this remains a potential limitation in certain application scenarios.

III. METHODOLOGY

The main objective of this project is to implement safe software solutions, including the Wasm and Lua approaches, into embedded systems to answer the two questions below.

1) Can memory-safe software be implemented in an embedded system?
2) How is the performance of the memory-safe software solution execute in an embedded environment?

In addition, a comparison among Wasm, Lua, and native (C) will be performed regarding footprint and performance to provide a comprehensive view of safe software solutions. The details about hardware and software that are used in this project, as well as the design flow and method that our team used to divide the project into sub-tasks for three team members, are introduced in sub-sections A, B, and C, respectively.

A. Hardware

1. ESP32-C6-DevKitC-1

ESP32-C6-DevKitC-1 [15] is an entry-level development board from Espressif, based on the ESP32-C6-WROOM-1(U) module, which is built around the ESP32-C6 chip. Some main features of this hardware platform are summarized below. ESP32-C6-WROOM-1(U) module built on ESP32-C6 SoC, which is a 32-bit RISC-V single-core microprocessor with a maximum clock speed up to 160 MHz. The Memory of this module is 8MB of SPI Flash, ROM 320 KB, HP SRAM 512 KB and LP SRAM 16 KB. For Wireless connection, the module supports different ways including Wi-Fi IEEE 802.11ax/b/g/n 2.4GHz band, Bluetooth LE 5.3, Zigbee 3.0, and Thread 1.3.

Operating voltage/Power supply: 3.0 ~ 3.6V.

- Support all popular peripherals like GPIO, SPI, UART, I2C, USB, JTAG,...
- Two USB ports: USB type-C to UART and USB type-C supporting JTAG debugging.
- Reset button, Boot button, and RGB LED.

2. R9A02G021 Fast Prototyping Board

R9A02G021 is a fast prototyping board from Renesas, based on the R9A02G0214CNE MCU, and its main characteristics can be summarized below.

B. R9A02G0214CNE MCU

a. RISC-V core, max 48 MHz
b. Memory: 128 KB ROM, 16 KB RAM, and 4 KB data flash memory
c. MCU internal clock
d. Providing 32.768 kHz reference clock

C. USB type-C, with built-in SEGGER J-Link OB debug probe (cJTAG: 2-wire compact JTAG)
D. Operating voltage/Power supply: 3.3V, MCU operation voltage range 1.6 V to 5.5 V,
E. Reset button, user button, and 4 LEDs.
F. Two Pmode connectors, Arduino connector, Seeed Grove connector

G. Software

Environment setup: This was done during the project's first week and is described below.

- Development environment: Ubuntu 24.04 LTS.
  The Linux-based operating system was selected to reduce complexity when building our project because we can run bash scripts and reuse a wide range of tools from open-source projects, which mainly support Linux-based environments.
- Integrated Development Environment (IDE):
  - Visual Studio Code, which is a popular IDE provided by Microsoft, is used to build the ESP-IDF project for the ESP32-C6-DevKit-1 board.
  - e² studio, which is an Eclipse-based IDE provided by Renesas, is used to build the Renesas RISC-V project for the Renesas R9A02G021 fast prototyping board.
- Framework/Toolchain:
  - Espressif IoT Development Framework (ESP-IDF) version 5.4.
    This framework includes the GCC-based toolchain for RISC-V.
  - LLVM for Renesas RISCV MCU, integrated with e² studio.
- Debugger:
  - OpenOCD to debug the ESP32-C6-DevKit-1.
  - The integrated debugger in e² studio and the SEGGER tools in the SEGGER™ J-Link™ Software and Documentation Package to debug on the Renesas board.

For Wasm/Lua integration:

- Wasm compiler: clang in WASI SDK, Rustc, Cargo, TinyGo, Ppci-mirror.
- Wasm runtime: WebAssembly Micro Runtime (WAMR).
- Lua runtime: LuaNode, LuatOS, and Lua RTOS.

For performance evaluation and optimization:

- Our team will investigate and select popular benchmarking tools to perform a performance analysis. Refer to section V for more information about planning this task.
- Our team also refers to the optimization methods provided by experts in WAMR and Lua runtime projects to reduce the footprints of Wasm and Lua applications.

H. Design Flow and Task Division

To ensure a systematic and efficient approach, we established a structured workflow that guides the development process from start to finish. In the development phase of this project, we followed this workflow to implement the integration, testing, and performance evaluation of WebAssembly applications on different embedded platforms. Figure 1 introduces the workflow and task assignment for implementing Wasm into the two hardware platforms. A similar approach is applied to the Lua part of the project.

1. Wasm /Lua/Native applications development
   Ziyang Zhou is in charge of developing image-processing applications in different programming languages, like C/C++, Rust, Lua, Go, and Python. For Wasm applications, it is necessary to compile the applications to Wasm format (.wasm) and then dump them into binary format (.h) for running in the Wasm runtime environment.

2. Platform integration and construction (Trong and Yuxuan). Because the same Wasm runtime and Wasm applications are applied to two different hardware platforms, Trong and Yuxuan need to work together on the ESP32-C6-DevKit-1 project first to make sure it can work as expected. This can prevent Ziyang's task from being blocked, and more than that, we can avoid wasting time fixing the same error on the two hardware platforms.
   a. ESP32-C6-DevKitC-1:
      In this project, the WAMR runtime, which can be downloaded from GitHub of the Bytecode Alliance, and the Wasm application in binary code format provided by Ziyang are integrated into the ESP-IDF project before being built and flashed to the ESP32-C6-DevKitC-1. The real-time operating system freeRTOS is used in this platform.

   b. Renesas R9A02G021 Fast Prototyping Board
      Similar works as ESP32-C6-DevKitC-1 are applied in this flow, however, the bare-metal environment is used instead of freeRTOS. It is noticed that many new functions need to be developed in the bare-metal environment to provide the Wasm runtime.

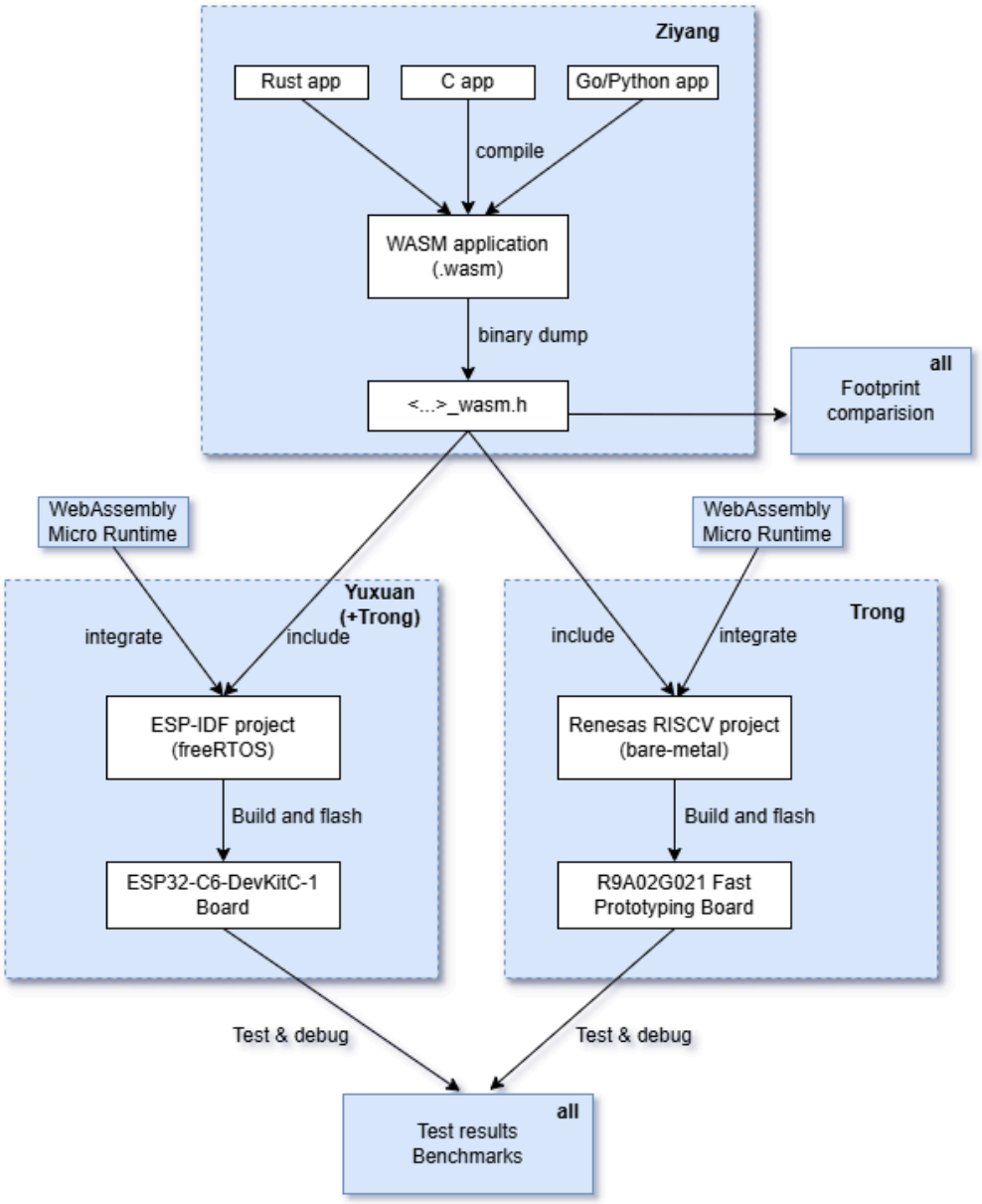

Figure 1. Design Flow and Task Division for Wasm.

IV. DESIGN DETAILS

In this section, the design of the Wasm application on an embedded system is introduced. The design for the Lua application will be provided in the final report.

A. Wasm Design Overview

Figure 2 illustrates the design overview for the implementation of of Wasm application to the two hardware platforms. Firstly, an image processing application written in C, Rust, or other languages is compiled into Wasm format (.wasm). This file can be tested in Linux, MacOS, or Windows using WAMR, but it needs to be dumped into binary format (.h) to run in an embedded system.

After that, the freeRTOS or bare-metal environment integrates WAMR into it, as well as includes the Wasm application (.h) to the project, then builds and flashes to the target hardware platform. Once the hardware platform starts, the freeRTOS or bare-metal runs the WAMR runtime, loads the application into the runtime, and then executes it.

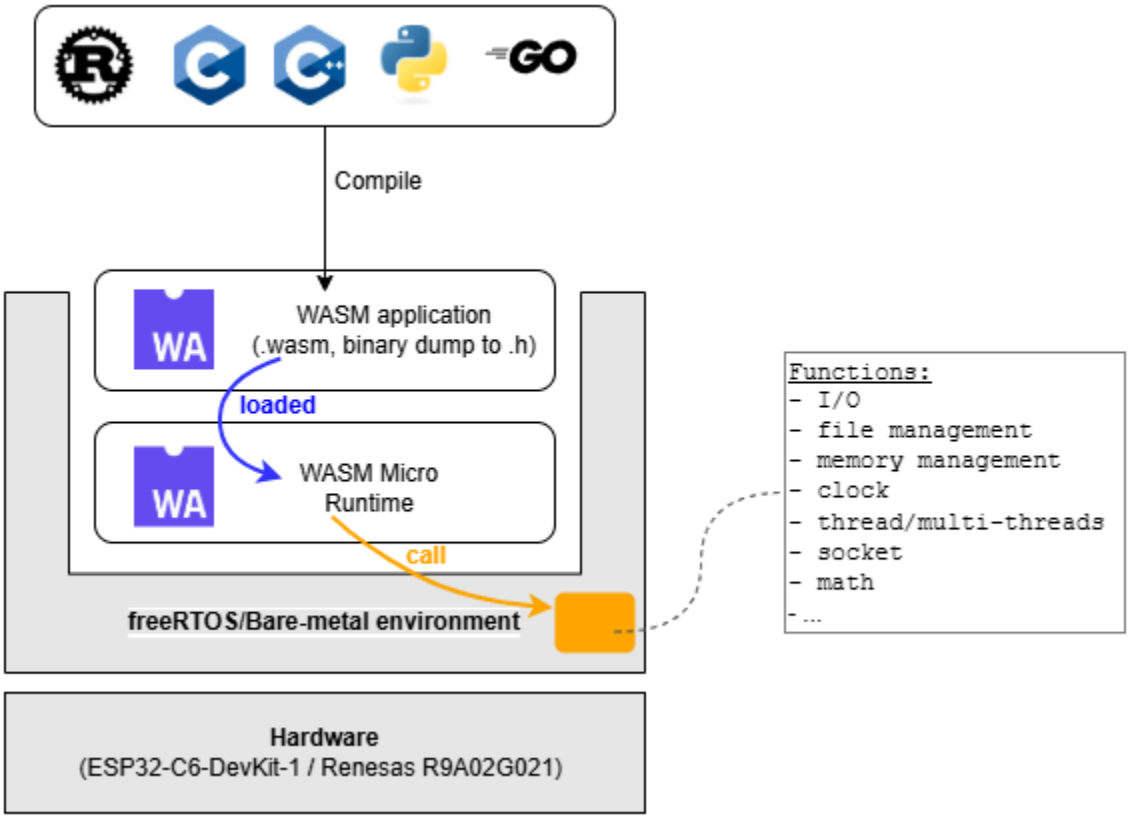

Figure 2. Wasm Design Overview

B. Image Processing Application

The project implements a grayscale image processing system capable of converting RGB image data provided in hexadecimal format into grayscale values. The core algorithm employs the luminance formula [16], which converts RGB values to grayscale by applying specific weights to each

color channel: 0.299 for red, 0.587 for green, and 0.114 for blue. These coefficients have been carefully selected to match human eye sensitivity to different colors. In the specific implementation, the system first performs strict format validation on the input hexadecimal string, ensuring its length is even and contains only valid hexadecimal characters (0-9 and A-F). After validation, the hexadecimal string is parsed and converted into a byte array, where every three bytes represent one pixel's RGB values. The processing adopts a chunk-based strategy, handling one RGB triplet at a time to calculate the corresponding grayscale value through the luminance formula. This processing approach not only ensures memory usage efficiency but also facilitates subsequent parallel processing optimization. The system assumes a default image size of 100x100 pixels, with output values ranging from 0 to 255, where each output pixel represents the weighted average of the input RGB components. To ensure conversion accuracy, floating-point numbers are used for intermediate calculations before converting the final results to 8-bit unsigned integers, thereby maximally preserving image detail information.

The implementation process involved developing two parallel versions of the code in C and Rust, both following the same logical structure outlined in the project documentation. The C implementation was compiled to WebAssembly using the WASI SDK's Clang compiler with optimization level O3 and appropriate export flags, as specified in the build script. The Rust version was compiled using rustc and Cargo with WebAssembly as the target. Both versions were successfully tested in a Linux environment using the WAMR. Following successful execution, the Wasm binaries were converted to C header files using binarydump and xxd commands, facilitating deployment to the ESP-IDF platform for embedded system testing. This dual-language approach allowed for performance comparison and validation of the implementation across different platforms while maintaining consistent functionality.

C. WAMR integration and Wasm application execution

The principle of running a Wasm application in embedded systems can be summarized in three main steps below.

- Step 1: Start the host environment. It can be freeRTOS in the ESP-IDF project or the bare-metal environment in the Renesas RISC-V project.
- Step 2: Allocate memory and start the WAMR runtime, different running modes, including interpreter, JIT, and AoT, are supported, but this project focuses on the interpreter and JIT mode only.
- Step 3: Load the Wasm application into the WAMR runtime and execute it.

The integration of the WAMR runtime into the ESP-IDF project can be done without any additional development in the WAMR runtime. This is because all the necessary functions used in WAMR are available in the real-time operating system freeRTOS. The runtime is defined as a

dependency in the main/idf_component.yml of the ESP-IDF project so that the compiler can automatically integrate the WAMR runtime into the freeRTOS.

One of two memory allocation methods, either WAMR using the host allocator or getting a fixed heap memory, must be configured in the freeRTOS or bare-metal environment before initializing the WAMR runtime as below.

```
#if WASM_ENABLE_GLOBAL_HEAP_POOL != 0
static char global_heap_buf[WASM_GLOBAL_HEAP_SIZE] = { 0 };
#endif

    /* configure memory allocation */
#if WASM_ENABLE_GLOBAL_HEAP_POOL == 0
    init_args.mem_alloc_type = Alloc_With_Allocator;
    init_args.mem_alloc_option.allocator.malloc_func = (void *)os_malloc;
    init_args.mem_alloc_option.allocator.realloc_func = (void *)os_realloc;
    init_args.mem_alloc_option.allocator.free_func = (void *)os_free;
#else
    init_args.mem_alloc_type = Alloc_With_Pool;
    init_args.mem_alloc_option.pool.heap_buf = global_heap_buf;
    init_args.mem_alloc_option.pool.heap_size = sizeof(global_heap_buf);
#endif

    /* initialize runtime environment */
    if (!wasm_runtime_full_init(&init_args)) {
        return;
    }
```

After that, the Wasm application is loaded to the WAMR runtime.

```
/* load WASM byte buffer from byte buffer of include file */
wasm_file_buf = (uint8_t *)wasm_test_file_interp;
wasm_file_buf_size = sizeof(wasm_test_file_interp);

/* load WASM module */
if (!(wasm_module = wasm_runtime_load(wasm_file_buf, wasm_file_buf_size,
                                      error_buf, sizeof(error_buf)))) {
    goto fail_1;
}
```

Finally, the WAMR is instantiated, followed by the Wasm application being instantiated inside the WAMR. It is important to allocate the appropriate stack size and heap size for the Wasm

application. These numbers must align with the stack size and heap size that the developer defined while compiling the application to Wasm format.

```
/* instantiate the module (WAMR) */
if (!(wasm_module_inst = wasm_runtime_instantiate(
      wasm_module, stack_size, heap_size, error_buf, sizeof(error_buf)))) {
    goto fail_2;
}
/* instantiate the app (Wasm app) */
app_instance_main(wasm_module_inst);

/* destroy the module instance */
wasm_runtime_deinstantiate(wasm_module_inst);
```

The integration of WAMR runtime into the Renesas RISC-V project requires developing new functions to be used in the WAMR runtime because the bare-metal environment does not provide them. This includes some memory management functions, like os_malloc(), os_realloc(), os_free(), redirect printf() function to the debug console port, and so on.

V. PROJECT PLAN AND MILESTONES

Part B of the project happens within three months of a semester, starting from January 7, 2025, and will be finished at the end of March 2025. This project has five important milestones, and its task assignment details are described in Table 2 below.

- Milestone 1: end of January.
  Goal: be able to run a simple Wasm application for image processing on the two boards, ESP32C6-DevKit1 and Renesas R9A02G021 Fast Prototyping board.
- Milestone 2: mid of February.
  Goal: be able to run a simple Lua application for image processing on the two boards, ESP32C6-DevKit1 and Renesas R9A02G021 Fast Prototyping board.
- Milestone 3: end of February.
  Goal: complete the footprint comparison and select suitable benchmark tools for the next phase.
- Milestone 4: mid of March.
  Goal: complete the benchmark tests and select a simple deep-learning model for running on ESP32C6-DevKit1 in the next phase.
- Milestone 5: end of March.
  Goal: define the process to run a lite deep-learning model on ESP32C6-DevKit1.

According to the first review at the end of January, we completed the most important tasks of the first milestones, but we are still behind schedule by one week, so we have put so much effort in February to catch up with the schedule. Currently, we can integrate WebAssembly Micro Runtime to the ESP-IDF project and run the Wasm application for image processing on the ESP32-C6-DevKit-1 board, but similar tasks on the Renesas R9A02G021 Fast Prototyping board are not completed. This is because the ESP-IDF development framework provides a real-time operating system, freeRTOS, and useful scripts and tools for integrating WAMR into its project. In contrast, Renesas only provides a simple bare-metal environment, so developers must implement system-level functions and scripts to integrate WAMR.

Table 2. Project Plan Details

Milestone	Task	Assign to	From	To	Duration	Status
End of January	Prerequisites: install Linux and software for use in the project	All	Jan 7	Jan 10	1 week	Done
	Pickup hardware and run the “Hello world” application	All	Jan 13	Jan 17	1 week	Done
	Create a Wasm application for image processing from C/Rust	Ziyang	Jan 20	Jan 24	1 week	Done (C only)
	Integrate Wasm Micro Runtime to ESP-IDF project	Yuxuan & Trong	Jan 20	Jan 31	2 weeks	Done
	Run Wasm application on ESP32C6-DevKit1 board	All	Jan 27	Jan 31	1 week	Done
	Integrate Wasm Micro Runtime to Renesas RISCV project	Trong	Jan 20	Jan 31	2 weeks	(to update on Feb12)
	Run Wasm application on Renesas R9A02G021 Fast Prototyping board	Trong & Ziyang	Jan 27	Jan 31	1 week	(to update on Feb 12)
Mid of February	Create a Lua application for image processing	Ziyang	Feb 3	Feb 14	2 weeks	Ongoing
	Integrate Lua runtime to ESP-IDF project	Yuxuan	Feb 3	Feb 14	2 weeks	Ongoing
	Test Lua application on ESP32C6-DevKit1 board	Yuxuan & Ziyang	Feb 10	Feb 14	1 week	Not started
	Integrate Lua runtime to Renesas RISCV project	Trong	Feb 3	Feb 14	1 week	Ongoing

	Test Lua application on Renesas R9A02G021 Fast Prototyping board	Trong & Ziyang	Feb 10	Feb 14	1 week	Not started
End of February	Compare footprints among Wasm, Lua, Native	All	Feb 17	Feb 28	2 weeks	Not started
	Investigate and select benchmark tools for performance analysis	All	Feb 24	Feb 28	1 week	Not started
Mid of March	Select a simple trained deep-learning model and compile it to Wasm for running on ESP32C6	Ziyang	Mar 3	Mar 14	2 weeks	Not started
	Benchmarks test on the ESP32C6-DevKit1 board (Wasm, Lua, Native)	Yuxuan	Mar 3	Mar 14	2 weeks	Not started
	Benchmarks test on the Renesas R9A02G021 Fast Prototyping board (Wasm, Lua, Native)	Trong	Mar 3	Mar 14	2 weeks	Not started
End of Match	Define a process to run the selected deep-learning model on the ESP32C6-DevKit-1 board	All	Mar 17	Mar 28	2 weeks	Not started

VI. QUESTIONS AND PROBLEMS

During the development process, we encountered a significant technical challenge when deploying Wasm applications compiled from Rust to the ESP-IDF platform. Specifically, while the build and flash drive processes were completed successfully, the system encountered malloc linear function error, which means this function is not successfully working during the monitoring phase when attempting to execute the WebAssembly module generated from Rust source files. This integration issue between Rust-generated WebAssembly modules and the ESP-IDF environment represents a critical obstacle that needs to be resolved to ensure the successful deployment and execution of our applications.

To systematically address this challenge, we have designed a comprehensive troubleshooting strategy consisting of the following three comparative experiments:

1. Compilation Method Comparison: Using identical Rust source code, we will generate WebAssembly files using two different compilation methods: rustc and cargo. Both resulting Wasm files will be converted to header files using the same command and tested on ESP-IDF to determine whether the compilation method affects the final execution results.
2. Rustc Generation Path Testing: Using the Wasm file generated by Rustc, we will create header files using two different methods: binary dump (WAMR's tool) and xxd

commands. Both header files will be tested on ESP-IDF. This will help us determine whether the header file generation method affects system execution when using Rustc compilation.

3. Cargo Generation Path Testing: Using the Wasm file generated by cargo, we will similarly create header files using both binary dump and xxd commands and test them on ESP-IDF. This experiment will verify whether different header file generation methods produce varying results when using cargo compilation.

Through these systematic comparative experiments, we aim to precisely identify whether the issue stems from the Rust compilation process, the header file generation method, or a combination of both factors, thereby determining the most reliable method for deploying Rust-based WebAssembly modules to ESP-IDF.

VII. FUTURE WORK

During the remaining time of the project, our group will follow the project plan and fix the problem shown in section VI, which can be summarized in the following main topics:

1. Fix the Rust to Wasm problem: The Wasm application did not run on the ESP32C6-C1-DevKit1 board when building the project from Cargo, not Rustc.
2. Lua on embedded system: Integrate Lua runtime into the ESP32-C6-DevKit-1 and Renesas R9A02G021 project, then run Lua application for image processing on both hardware platforms.
3. Wasm from different programming languages: Create an image processing application in Go or Python, compile it to Wasm, and run it on ESP32-C6-DevKit-1 and Renesas R9A02G021 hardware platforms.
4. Footprint: Compare footprints of Wasm, Lua, and native (C).
5. Benchmarks: Compare the performance of Wasm, Lua, and native (C).
6. deep Learning: Investigate how to run a lite machine learning model as a Wasm application on the ESP32-C6-DevKit-1 board.

REFERENCES

[1] “WebAssembly Core Specification.” Accessed: Nov. 28, 2024. [Online]. Available: https://www.w3.org/TR/wasm-core-2/
[2] “Wasmtime.” Accessed: Nov. 27, 2024. [Online]. Available: https://docs.wasmtime.dev/stability-platform-support.html
[3] “Wasmer.” Accessed: Nov. 27, 2024. [Online]. Available: https://docs.wasmer.io/runtime/features
[4] “WAVM.” Accessed: Nov. 27, 2024. [Online]. Available: https://wavm.github.io/
[5] *WasmEdge*. (Nov. 27, 2024). C++. WasmEdge Runtime. Accessed: Nov. 27, 2024. [Online]. Available: https://github.com/WasmEdge/WasmEdge

[6] “bytecodealliance/wasm-micro-runtime: WebAssembly Micro Runtime (WAMR).” Accessed: Feb. 08, 2025. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime
[7] “wasm3.” Accessed: Nov. 27, 2024. [Online]. Available: https://github.com/wasm3/wasm3
[8] “The Programming Language Lua.” Accessed: Feb. 08, 2025. [Online]. Available: https://lua.org/
[9] “The LuaJIT Project.” Accessed: Feb. 08, 2025. [Online]. Available: https://luajit.org/
[10] “eLua - eluaproject.” Accessed: Nov. 29, 2024. [Online]. Available: https://eluaproject.net/
[11] “NodeMCU Documentation.” Accessed: Nov. 29, 2024. [Online]. Available: https://nodemcu.readthedocs.io/en/release/
[12] N. Wang, *Nicholas3388/LuaNode*. (Feb. 08, 2025). C. Accessed: Feb. 08, 2025. [Online]. Available: https://github.com/Nicholas3388/LuaNode
[13] “LuatOS documentation.” Accessed: Feb. 08, 2025. [Online]. Available: https://wiki.luatos.org/
[14] *whitecatboard/Lua-RTOS-ESP32*. (Feb. 06, 2025). C. Whitecat. Accessed: Feb. 09, 2025. [Online]. Available: https://github.com/whitecatboard/Lua-RTOS-ESP32
[15] W. J. Lee, C. H. Kim, and S. W. Kim, “A Last-Level Cache Management for Enhancing Endurance of Phase Change Memory,” in *2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, Jun. 2021, pp. 1–4. doi: 10.1109/ITC-CSCC52171.2021.9501266.
[16] “HSP Color Model.” Accessed: Feb. 12, 2025. [Online]. Available: https://alienryderflex.com/hsp.html