
WebAssembly Integration for Portable Embedded Applications

Ravinder Seth¹, Ankur Joshi²

Associate Professor, Assistant Professor

Department of Computer Science

Vidyadeep College of Engineering

ravinderseth55@gmail.com¹, ankur7ajoshi@yahoo.com²

Abstract

Embedded systems are increasingly required to run applications across heterogeneous hardware platforms with limited resources, strict power budgets, and long deployment lifetimes. Traditional approaches depend heavily on hardware-specific toolchains and operating systems, which reduces portability and increases development effort. WebAssembly (Wasm), originally designed for safe and fast execution inside web browsers, is now emerging as a promising portable runtime for embedded applications. Its sandboxed execution model, compact binary format, near-native performance, and language-agnostic compilation support make it suitable for resource-constrained devices. This paper reviews how WebAssembly can be integrated into embedded platforms to achieve portability, modularity, and secure execution. It discusses Wasm architecture, runtime models, memory handling, toolchains, and case studies of microcontroller and IoT deployments. Challenges such as real-time behavior, memory limits, and hardware interfacing are also analyzed. The study shows that WebAssembly provides a practical path toward platform-independent embedded software design, reducing vendor lock-in and simplifying cross-device deployment.

Keywords

WebAssembly, Embedded Systems, Portability, IoT, Sandbox Execution, Lightweight Runtime, Edge Computing, WASI

1. INTRODUCTION

Embedded applications traditionally depend on tightly coupled hardware and software design. Programs written for one microcontroller architecture often cannot be reused on another without major modification. This increases development cost and slows innovation. With the rise of IoT and edge devices, the need for portable and secure embedded software is becoming more important.

WebAssembly (Wasm) is a binary instruction format designed for stack-based virtual machines. Though initially intended for browsers, Wasm is now supported in standalone runtimes such as Wasmtime, Wasm3, and WAMR (WebAssembly Micro Runtime). These runtimes can execute Wasm modules on small embedded devices with very low overhead.

The interesting part is that WebAssembly allows code written in C, C++, Rust, Go and other languages to compile into a single portable binary that runs on multiple architectures without recompilation. This property can significantly simplify embedded software distribution.

2. OVERVIEW OF WEBASSEMBLY ARCHITECTURE (ELABORATED)

WebAssembly (Wasm) is a low-level, stack-based binary instruction format intended to be executed by a virtual machine. Although it was first introduced for web browsers, its architecture is highly suitable for embedded platforms because of its **platform neutrality, safety guarantees, and predictable execution**. The Wasm architecture separates **application logic** from **hardware interaction**, which is a key advantage for portable embedded development.

WebAssembly is designed around four core principles:

- **Portability across platforms**
- **Safety through sandboxed execution**
- **Compact binary representation**
- **Deterministic and predictable behavior**

These principles are achieved through a well-defined module structure, strict validation rules, and a runtime that mediates all interactions with the outside world.

2.1 Wasm Module Structure

A WebAssembly application is packaged as a **module**. This module is a binary file (.wasm) that contains:

- Function definitions
- Memory declarations
- Global variables
- Import and export tables
- Type definitions

The module is **self-contained** and cannot perform any operation outside its boundary unless explicitly permitted by the host runtime.

Table 1: Basic Sections of a Wasm Module

Section	Purpose
Type Section	Defines function signatures
Import Section	Declares external functions/memory needed
Function Section	Lists functions defined inside module
Memory Section	Declares linear memory requirements
Export Section	Exposes functions to host
Code Section	Contains compiled bytecode instructions

This structure ensures that the runtime can **validate** the module before execution, preventing malformed or unsafe code from running.

2.2 Stack-Based Virtual Machine

WebAssembly uses a **stack machine model** instead of register-based architecture. Instructions push and pop values from a stack.

Example (conceptual):

```
i32.const 5
i32.const 3
i32.add
```

This pushes 5 and 3 to the stack and adds them. The simplicity of this model allows Wasm runtimes to be very small, which is ideal for microcontrollers.

2.3 Sandbox Execution Model

One of the most important architectural aspects is the **sandbox**.

- Wasm code cannot access arbitrary memory
- No direct hardware or OS access
- All external interaction happens via **imports** (host functions)

This prevents:

- Buffer overflow attacks
- Unauthorized peripheral access
- Corruption of system memory

For embedded devices deployed in the field (IoT, industrial nodes), this is a major advantage.

2.4 Host Environment and System Calls

Wasm modules rely on a **host environment** for performing tasks like:

- Reading sensors
- Writing to UART/SPI/I2C
- Accessing timers
- File or network operations

This is done using **WASI (WebAssembly System Interface)** or custom host bindings mapped to the Hardware Abstraction Layer (HAL).

Execution Flow

Because of this separation, the same Wasm module can run on different hardware by only changing the host layer.

2.5 Linear Memory Model

Wasm uses a **linear memory**, which is a contiguous block of bytes. The module can only access this memory region using safe, bounds-checked instructions.

- Memory grows in 64 KB pages
- No pointer arithmetic outside allocated memory
- Prevents memory corruption

This model is simpler than traditional heap/stack memory used in C/C++ and suits embedded predictability.

2.6 Deterministic Execution

Wasm is designed to be deterministic:

- No undefined behavior
- Strict typing rules
- Validated before execution
- Predictable instruction timing (important for embedded tasks)

This helps in designing reliable firmware components where timing and behavior must be known.

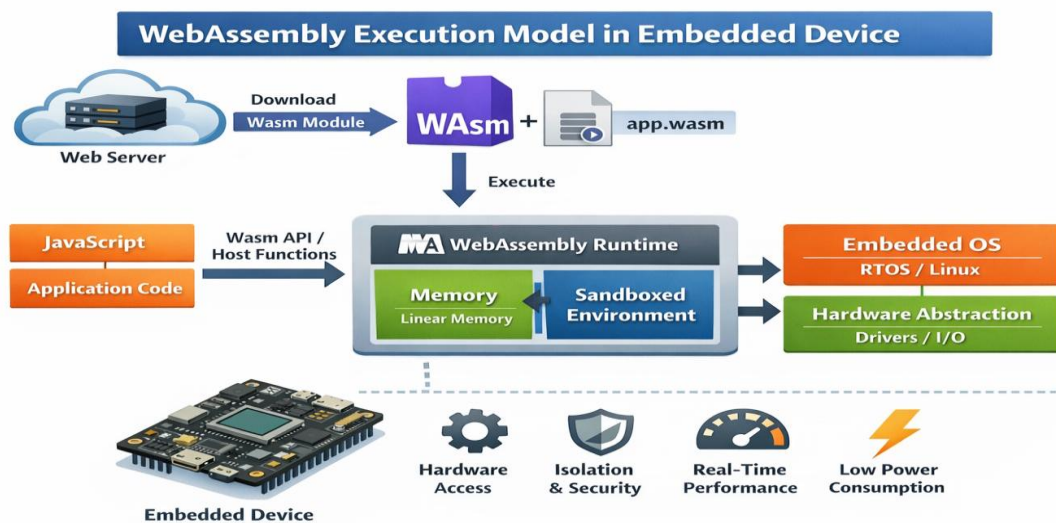


Figure 1: WebAssembly Execution Model in Embedded Device

3. WHY WEBASSEMBLY FOR EMBEDDED SYSTEMS (ELABORATED)

Embedded systems today are no longer single-purpose, static devices. They are connected, frequently updated, and often expected to support third-party logic, edge analytics, and long life cycles across changing hardware. In such an environment, traditional firmware

development—tightly bound to processor architecture and toolchains—creates maintenance and portability challenges.

WebAssembly offers a **runtime-based execution model** where application logic is separated from hardware specifics. This makes it very attractive for modern embedded and IoT deployments.

3.1 Portability

One of the biggest difficulties in embedded development is **architecture dependency**. Firmware written for ARM Cortex-M cannot directly run on RISC-V or Xtensa processors. Developers must recompile, retest, and sometimes redesign portions of the software.

WebAssembly solves this by introducing a **hardware-neutral binary format**.

- The .wasm file contains bytecode, not machine code.
- A Wasm runtime translates this bytecode to native instructions at execution time.
- The same Wasm module can run on different processors as long as a compatible runtime exists.

Implications for embedded design:

- Firmware logic can be reused across product lines.
- Easier migration from one microcontroller vendor to another.
- Reduced vendor lock-in.
- Faster prototyping across boards.

For example, a sensor data filtering algorithm compiled to Wasm can run unchanged on STM32 (ARM), ESP32 (Xtensa), and a RISC-V development board.

3.2 Security

Security is a major concern for connected embedded devices deployed in remote or hostile environments. Memory corruption, buffer overflows, and unauthorized hardware access are common attack vectors in C/C++ based firmware.

WebAssembly introduces a **sandbox execution model**:

- No direct pointer access to system memory
- Strict bounds checking on memory
- No direct access to peripherals
- All hardware interaction via controlled host functions

This prevents common vulnerabilities such as:

- Stack smashing
- Heap corruption
- Arbitrary code execution
- Unauthorized register access

Additionally, Wasm modules are **validated before execution**, ensuring that malformed or unsafe binaries cannot run.

This makes WebAssembly suitable for:

- Secure firmware plugins
- Third-party logic deployment
- Industrial IoT nodes requiring high reliability

3.3 Small Footprint

A common misconception is that virtual machines are heavy and unsuitable for microcontrollers. However, embedded Wasm runtimes are specifically optimized for low memory usage.

Examples:

- **Wasm3** interpreter can run in ~50–60 KB RAM
- **WAMR** can run in under 100 KB
- Wasm modules themselves are often only a few KB

This makes WebAssembly practical for:

- Cortex-M4/M7 microcontrollers
- ESP32 class devices
- IoT sensor nodes

Because the runtime is small and the modules are compact, flash and RAM requirements remain within acceptable limits for many embedded platforms.

3.4 Language Support

Traditional embedded programming relies heavily on C/C++ and sometimes assembly. This increases development complexity and risk of memory errors.

WebAssembly allows developers to use **modern, safer languages**:

- C / C++
- Rust
- Go
- TinyGo
- AssemblyScript

These languages compile into Wasm bytecode using LLVM or other toolchains.

Benefits:

- Developers avoid low-level hardware coding
- Memory-safe languages like Rust reduce bugs
- Easier onboarding of software developers into embedded domain
- Reuse of existing libraries compiled to Wasm

This significantly reduces development time and improves code reliability.

4. WEBASSEMBLY RUNTIME OPTIONS FOR EMBEDDED DEVICES

For WebAssembly to run on embedded hardware, a **Wasm runtime (virtual machine)** must be present on the device. This runtime is responsible for:

- Loading and validating the .wasm module
- Managing memory
- Translating Wasm bytecode into native instructions
- Providing host functions for hardware access

Different runtimes are designed with different trade-offs in **memory usage, speed, and feature support**. Selecting the proper runtime is a key design decision when integrating WebAssembly into embedded platforms.

4.1 Types of Wasm Execution Modes

Embedded runtimes generally support one or both of the following modes:

1. **Interpreter Mode**

- Executes Wasm instructions one by one
- Very small memory footprint
- Slower execution
- Suitable for microcontrollers

2. **AOT (Ahead-of-Time) Compilation**

- Converts Wasm into native machine code before execution
- Faster performance
- Slightly higher memory usage
- Suitable for gateways and embedded Linux devices

Some runtimes support both modes, allowing flexibility.

4.2 Popular WebAssembly Runtimes for Embedded Systems

Wasm3

- Ultra-lightweight interpreter
- Designed specifically for microcontrollers
- No dependency on OS
- Runs on bare metal, FreeRTOS, Zephyr
- RAM usage ~50–60 KB

Best for: STM32, ESP32, Cortex-M boards, sensor nodes

WAMR (WebAssembly Micro Runtime)

- Developed by Bytecode Alliance / Intel
- Supports interpreter and AOT modes
- Provides partial WASI support
- Good balance between size and performance
- RAM usage ~80–100 KB

Best for: IoT gateways, industrial controllers

Wasmtime

- Full-featured Wasm runtime
- Complete WASI support
- Requires Embedded Linux
- High performance with JIT/AOT
- Memory >1 MB

Best for: Edge devices, Raspberry Pi class boards

WAVM

- High-performance runtime
- More suitable for desktop/edge than microcontrollers
- Rich feature set

Best for: Embedded Linux with more RAM

Table 2:

Runtime	RAM Requirement	Key Feature	Suitable Devices
Wasm3	~50 KB	Interpreter, ultra-small	Cortex-M, ESP32
WAMR	~85 KB	AOT & Interpreter modes	IoT gateways
Wasmtime	>1 MB	Full WASI support	Edge computers
WAVM	>2 MB	High performance	Embedded Linux

These runtimes provide different trade-offs between speed and memory usage.

5. INTEGRATION WITH EMBEDDED OPERATING SYSTEMS

WebAssembly can run over:

- FreeRTOS
- Zephyr OS
- Embedded Linux
- Bare metal systems

The runtime sits as a task or library within the OS. The host functions map Wasm calls to hardware drivers.

6. WEBASSEMBLY SYSTEM INTERFACE (WASI)

WASI provides standardized APIs for:

- File operations
- Time and clocks
- Random number generation
- Networking (limited)

In embedded systems, WASI calls are mapped to Hardware Abstraction Layer (HAL) functions.

7. MEMORY MANAGEMENT IN WASM FOR CONSTRAINED DEVICES

Wasm uses linear memory model. Memory is preallocated and grows in pages (64 KB each). Embedded runtimes often limit this to few pages.

Advantages:

- Predictable memory usage
- No dynamic heap corruption
- Safe bounds checking

Limitation: not ideal for very tiny (<32 KB RAM) devices.

8. TOOLCHAIN FOR BUILDING WASM EMBEDDED APPLICATIONS

Typical flow:

Rust is becoming popular due to memory safety and easy Wasm compilation.

9. USE CASES IN EMBEDDED AND IOT

9.1 IoT Sensor Nodes

Portable sensor logic deployed across devices.

9.2 Edge AI Preprocessing

Wasm used for signal filtering before cloud upload.

9.3 Secure Firmware Plugins

Vendors allow third-party Wasm modules safely.

9.4 Industrial Gateways

Protocol handlers written in Wasm for flexibility.

10. PERFORMANCE CONSIDERATIONS

Wasm execution is slower than native code but faster than interpreted Python/Lua.

Table 3:

Method	Relative Speed
Native C	1.0x
Wasm AOT	1.3x
Wasm Interpreter	2.5x
Python	8x

For many embedded tasks, this overhead is acceptable.

11. SECURITY ADVANTAGES

- No buffer overflow outside sandbox
- Controlled host calls
- Safe plugin model
- Code verification before execution

12. CHALLENGES IN WEBASSEMBLY EMBEDDED ADOPTION

- Real-time determinism issues
- Limited debugging tools
- Hardware interrupt handling
- Memory page granularity
- Lack of mature standards for embedded WASI

13. CASE STUDY: ESP32 RUNNING WASM3

Researchers demonstrated Wasm3 on ESP32 with 520 KB RAM:

- Sensor data filtering in Wasm
- Runtime size: 58 KB
- Wasm module: 12 KB
- Stable performance with low power use

14. COMPARISON WITH OTHER PORTABLE SOLUTIONS

Table 4:

Feature	WebAssembly	Java VM	MicroPython
Memory	Low	High	Medium
Speed	Medium	Medium	Low
Security	High	Medium	Low
Portability	High	High	High

15. FUTURE SCOPE

- WASI for embedded standardization
- Hardware accelerated Wasm execution
- Integration with RISC-V custom instructions
- Edge cloud unified deployment

16. DISCUSSION

WebAssembly does not replace native firmware but complements it. Critical drivers remain native, while application logic moves to Wasm layer. This hybrid model reduces firmware updates and allows modular deployment.

17. CONCLUSION

WebAssembly provides a new direction for portable embedded application development. Its sandbox security, compact runtime, and architecture independence make it suitable for IoT and edge devices. Though challenges remain in real-time support and hardware access, ongoing development in WASI and lightweight runtimes is addressing these gaps. WebAssembly can significantly reduce vendor dependency and simplify software portability across embedded platforms.

REFERENCES

1. Haas, A. et al., "Bringing the Web up to Speed with WebAssembly," ACM SIGPLAN, 2017.

2. Jangda, A. et al., “Not So Fast: Analyzing the Performance of WebAssembly vs Native Code,” USENIX, 2019.
3. WebAssembly.org, “WebAssembly Core Specification,” W3C, 2022.
4. Wasmtime Project Documentation, Bytecode Alliance, 2023.
5. Wasm3 Runtime GitHub Repository Documentation, 2023.
6. WAMR (WebAssembly Micro Runtime) Technical Overview, Intel, 2022.
7. K. S. Bhattacharya, “Portable IoT Applications using WebAssembly,” IEEE IoT Journal, 2021.
8. Zephyr Project Documentation: WebAssembly Integration Notes, 2023.
9. R. Sharma, “Security Advantages of Sandbox Execution in Embedded Devices,” Embedded Computing Design, 2022.
10. Bytecode Alliance, “WASI: WebAssembly System Interface Proposal,” 2023.