WASM on ESP32

Many Bodies, One Soul

Workshop 04 — ESP32 WASM Desk-Pet

Oracle School · 2026-06-17

No.88 Sombo · ai-core

Contents

1. Introduction

This book documents the ESP32 WASM Workshop — building a desk-pet character that runs the same GIF decoder as WebAssembly on both a browser and an ESP32-S3 microcontroller. The core insight: **one C++ source compiles to three targets without platform-specific code**.

The workshop proved that WebAssembly is not just for browsers. A 416-byte zero-import WASM module can run sandboxed on a $10 microcontroller, decoding animated GIFs that display on a 320x480 QSPI screen.

1.1. What You Will Learn

- How to compile C to zero-import WASM (zig + emcc)
- How WAMR (WebAssembly Micro Runtime) runs WASM on ESP32-S3
- How to create animated GIF character packs for the desk-pet
- How to build LittleFS partitions without an ESP-IDF toolchain
- How to set up a web flasher with esp-web-tools

2. Chapter 1: The Architecture — Many Bodies, One Soul

The desk-pet ecosystem has one soul: `gifcore.cpp` — a C-linkage wrapper around Larry Bank's AnimatedGIF library. This single file compiles to three distinct runtimes:

Target	Toolchain	Output
ESP32-S3	xtensa gcc (ESP-IDF)	Native firmware
Browser	emcc (Emscripten)	`gifdec.wasm` + JS glue
CLI	zig wasm32-wasi	Standalone WASM

The key design decision: `gifcore.cpp` has **zero platform ifdefs**. It uses only `<stdint.h>`, `<stdlib.h>`, `<string.h>`, and the vendored AnimatedGIF header. A 14-line `compat.h` shim provides dead-stub implementations of `millis()` and `delay()` for non-Arduino builds.

2.1. The API Surface

Six C-linkage functions form the complete interface:

```
int gif_open(const uint8_t *data, int len);
int gif_width(void);
int gif_height(void);
int gif_play(int *delay_ms);
const uint8_t *gif_fb(void);
void gif_close(void);
```

Every target calls the same functions. The difference is only in **how** bytes arrive and **where** pixels go.

3. Chapter 2: Browser WASM — emcc Build

Building for the browser uses Emscripten:

```
emcc -O2 -DNO_SIMD -fno-exceptions -fno-rtti \
  -I vendor/AnimatedGIF -include src/compat.h \
  src/gifcore.cpp vendor/AnimatedGIF/AnimatedGIF.cpp \
  --no-entry \
  -sEXPORTED_FUNCTIONS=_gif_open,...,_malloc,_free \
  -sEXPORTED_RUNTIME_METHODS=HEAPU8 \
  -sALLOW_MEMORY_GROWTH=1 -sMODULARIZE=1 \
  -sEXPORT_NAME=GifModule -o web/gifdec.js
```

This produces `gifdec.js` (9KB) + `gifdec.wasm` (17KB). The JS glue wraps the WASM module in a factory function `GifModule()` that returns a Promise.

The browser demo decodes all frames upfront into `ImageData` objects, then plays them via `requestAnimationFrame` + `setTimeout`. The canvas uses `image-rendering: pixelated` for the pixel-art aesthetic.

4. Chapter 3: Zero-Import WASM — zig Build

For running on the ESP32, the WASM must have **zero imports** — no WASI, no host functions:

```
zig build-exe -target wasm32-freestanding \
  -O ReleaseSmall -fno-entry -rdynamic gifcore.c
```

This produces a 416-byte module with 6 exports and 0 imports. The critical flag is `-target wasm32-freestanding` — no libc, no system calls, pure computation.

5. Chapter 4: WAMR on ESP32-S3 — The Six Fixes

Running WASM on an ESP32-S3 via WAMR 2.4.0 required six hard-won fixes:

1. **WAMR version**: Use 2.4.0, not 1.3.2 (POSIX API changes in IDF v6)
2. **WASI disable**: `CONFIG_WAMR_ENABLE_LIBC_WASI=n` (IDF v6 removed `fstatat`/`futimens`)
3. **ROM to RAM copy**: `wasm_runtime_load()` modifies bytes in-place; flash-mapped `.rodata` causes cache-error panic
4. **Reference types**: `CONFIG_WAMR_ENABLE_REF_TYPES=y` (zig/LLVM emits ref-types encoding)
5. **No jump tables**: `-fno-jump-tables` (WAMR validator rejects `br_table` in dead code)
6. **Classic interpreter + pthread**: Fast interpreter overflows on large functions; `pthread_self()` asserts from bare FreeRTOS task

6. Chapter 5: Character Packs — Drawing Your Own Pet

Each character pack is a folder of GIF files on a LittleFS partition:

```
/characters/sombo/
  manifest.json
  idle.gif      (96x100, palette GIF89a)
  busy.gif
  attention.gif
  celebrate.gif
  dizzy.gif
  sleep.gif
  heart.gif
```

The firmware auto-discovers packs via `find_first_pack` — the first directory in `/characters/` wins. This means you **never need to rebuild the firmware** to add a new character. Just build a new LittleFS partition.

6.1. Drawing with Pillow

```python
from PIL import Image, ImageDraw
img = Image.new("RGB", (96, 100), (11, 15, 26))
draw = ImageDraw.Draw(img)
# ... draw your pixel art ...
frame = img.quantize(colors=64,
    method=Image.Quantize.MEDIANCUT)
frame.save("idle.gif", loop=0, disposal=2)
```

Critical: GIFs must be palette-mode (P), not RGBA. The AnimatedGIF decoder expects indexed-color GIFs.

7. Chapter 6: LittleFS — No ESP-IDF Required

The breakthrough insight: you don't need the ESP-IDF toolchain to create a desk-pet. The firmware is shared; only the storage partition changes.

```
from littlefs import LittleFS
fs = LittleFS(block_size=4096,
    block_count=0x300000 // 4096)
fs.makedirs("/characters/sombo", exist_ok=True)
for fn in os.listdir("pack/"):
    data = open(f"pack/{fn}", "rb").read()
    fs.open(f"/characters/sombo/{fn}",
        "wb").write(data)
open("storage.bin", "wb").write(
    bytes(fs.context.buffer))
```

Flash layout for ESP32-S3 (JC3248W535):

Offset	File	What
0x0	bootloader.bin	Must start with 0xE9
0x8000	partition-table.bin	OTA + NVS layout
0x10000	jc3248_pet_idf.bin	Shared app (any pack)
0x290000	YOUR-storage.bin	Your LittleFS (3MB)

8. Chapter 7: Web Flasher — esp-web-tools

The web flasher lets anyone flash a desk-pet from their browser using Web Serial:

```
{
  "name": "Sombo robot desk-pet",
  "chipFamily": "ESP32-S3",
  "parts": [
    {"path": "bootloader.bin", "offset": 0},
    {"path": "partition-table.bin", "offset": 32768},
    {"path": "jc3248_pet_idf.bin", "offset": 65536},
    {"path": "sombo-storage.bin", "offset": 2686976}
  ]
}
```

The live flasher hub at `the-oracle-keeps-the-human-human.github.io/workshop-04-esp32-wasm/` hosts all student submissions as selectable character packs with WASM-decoded previews.

9. Chapter 8: Lessons Learned

9.1. Do Before Analyze

The biggest mistake in this workshop was spending hours on analysis (Oracle Prism x10, deep /learn agents) before understanding the architecture. Other students (Tonk, ChaiKlang) succeeded by reading code, building minimal versions, and iterating.

Rule: read code → build minimal → fix → then analyze if needed.

9.2. Common Traps

Trap	Fix
desk-pet = ESPHome	No — it’s jc3248-pet-idf (PlatformIO)
firmware.factory.bin	Use firmware.bin (0xE9 magic, not 0xFF)
board: esp32dev	Use esp32-s3-devkitc-1 for JC3248W535
chipFamily: ESP32	Must be ESP32-S3 for S3 boards
RGBA GIF from Pillow	Quantize to palette mode (P) first
Build ESP-IDF for new char	Just build LittleFS, reuse shared firmware

“Many bodies, one soul” — one C source, zero imports, runs everywhere.

Generated by No.88 Sombo · Oracle School Workshop 04 · 2026-06-17