Integrating MicroQuickJS with ESP32 ESP-IDF: A Complete Guide

Author: Manus AI
Date: December 28, 2025
Version: 1.0

Table of Contents

1. Introduction
2. Part I: Foundations
3. Part II: Practical Integration
4. Part III: Under the Hood
5. Part IV: Advanced Features
6. Part V: Optimization and Best Practices
7. Conclusion
8. References

Introduction

JavaScript has become one of the most popular programming languages in the world, powering everything from web applications to server-side systems. However, its presence in the embedded systems world has been limited due to the resource constraints of microcontrollers. MicroQuickJS changes this by bringing a lightweight, embeddable JavaScript engine to resource-constrained devices like the ESP32.

This guide provides a comprehensive exploration of integrating MicroQuickJS with the ESP-IDF framework for the ESP32 family of microcontrollers. It is designed to serve both as a practical tutorial for developers looking to get started quickly and as an in-

depth technical reference for those who want to understand the internals of how JavaScript runs on embedded systems.

Why JavaScript on Embedded Systems?

JavaScript offers several advantages for embedded development:

Rapid Prototyping: JavaScript's dynamic nature and high-level abstractions allow for faster development cycles compared to traditional C/C++ development. Developers can test ideas quickly without the compile-link-flash cycle.

Accessibility: JavaScript is one of the most widely known programming languages, with millions of developers worldwide. Bringing JavaScript to embedded systems lowers the barrier to entry for web and application developers who want to work with hardware.

Dynamic Behavior: Unlike compiled C code, JavaScript can be modified at runtime. This enables over-the-air updates, user-programmable devices, and dynamic configuration without reflashing firmware.

Ecosystem: The JavaScript ecosystem is vast, with countless libraries and tools. While not all of these can run on embedded systems, the familiarity with the language and its patterns can accelerate development.

Why MicroQuickJS?

MicroQuickJS is a fork of QuickJS, created by Fabrice Bellard (the creator of QEMU and FFmpeg). It is specifically optimized for embedded systems with the following characteristics:

Small Memory Footprint: MicroQuickJS can run with as little as 64KB of RAM, making it suitable for microcontrollers. The entire engine, including the runtime and garbage collector, fits within this constraint.

No Dynamic Memory Allocation: Unlike standard JavaScript engines that rely heavily on malloc/free, MicroQuickJS uses a pre-allocated memory pool. This is critical for embedded systems where heap fragmentation and unpredictable allocation failures can cause system instability.

ES5.1 Compliance: MicroQuickJS implements a substantial subset of ECMAScript 5.1, providing modern JavaScript features like closures, prototypes, and first-class functions while avoiding the complexity of ES6+ features that would increase code size.

Simple C API: The engine provides a clean, minimal C API that makes it easy to embed in existing projects and expose custom native functions to JavaScript.

No OS Dependencies: MicroQuickJS can run on bare metal or with an RTOS like FreeRTOS, making it portable across different embedded platforms.

About This Guide

This guide is structured in five parts:

Part I: Foundations provides the background knowledge needed to understand both ESP-IDF and MicroQuickJS, including their architectures and design philosophies.

Part II: Practical Integration walks through the step-by-step process of integrating MicroQuickJS into an ESP-IDF project, building a functional JavaScript REPL, and adding persistent storage.

Part III: Under the Hood dives deep into the internals of how MicroQuickJS works, how memory is managed, how the bytecode interpreter functions, and how the garbage collector operates.

Part IV: Advanced Features covers topics like exposing custom C functions to JavaScript, interfacing with hardware peripherals, optimizing performance, and debugging.

Part V: Optimization and Best Practices provides guidelines for writing efficient JavaScript code for embedded systems, managing memory effectively, and avoiding common pitfalls.

Part I: Foundations

Before diving into the integration process, it is essential to understand the two main components we will be working with: the ESP-IDF framework and the MicroQuickJS engine.

1.1. The ESP-IDF Framework

The ESP-IDF (Espressif IoT Development Framework) is the official development framework for Espressif's ESP32 family of microcontrollers. It provides a comprehensive set of libraries, tools, and APIs for building embedded applications.

Architecture Overview

ESP-IDF is built on top of FreeRTOS, a real-time operating system designed for embedded systems. The framework follows a layered architecture:

Hardware Abstraction Layer (HAL): At the lowest level, ESP-IDF provides hardware abstraction for peripherals like GPIO, UART, SPI, I2C, WiFi, and Bluetooth. These abstractions hide the complexity of register-level programming and provide a consistent API across different ESP32 variants.

FreeRTOS Kernel: The FreeRTOS kernel provides task scheduling, inter-task communication (queues, semaphores, mutexes), and memory management. ESP-IDF extends FreeRTOS with ESP32-specific features like dual-core support and tickless idle mode.

Component System: ESP-IDF uses a component-based architecture where functionality is organized into reusable modules. Each component has its own source files, include directories, and dependencies. The build system (CMake) automatically resolves dependencies and links components together.

Application Layer: User applications are built on top of the framework, using the provided APIs and components. Applications can create multiple FreeRTOS tasks, use WiFi and Bluetooth, access file systems, and interact with hardware peripherals.

The Build System

ESP-IDF uses CMake as its build system. The build process involves several steps:

Configuration: The `idf.py menuconfig` command launches a text-based configuration tool where developers can set options like target chip, partition table, WiFi settings, and component-specific parameters. These settings are saved in `sdkconfig`.

Component Discovery: CMake scans the project for components, which can be in the `components` directory, the ESP-IDF framework itself, or external repositories. Each

component must have a `CMakeLists.txt` file that registers it with the build system.

Dependency Resolution: CMake analyzes the dependencies between components and determines the correct build order. If component A requires component B, B will be built first.

Compilation: The toolchain (xtensa-esp32-elf-gcc for Xtensa-based chips or riscv32-esp-elf-gcc for RISC-V) compiles each component into object files. The compiler applies optimizations, generates debug symbols, and produces position-independent code.

Linking: The linker combines all object files into a single ELF executable. The linker script defines the memory layout, placing code in flash (IROM) and data in RAM (DRAM/IRAM). The ESP32 has a complex memory architecture with multiple regions, and the linker must carefully manage placement to avoid overflows.

Binary Generation: The ELF file is converted into a binary format suitable for flashing. This includes the bootloader, partition table, and application binary. The `esptool.py` utility handles the conversion and flashing process.

Memory Architecture

Understanding the ESP32's memory architecture is crucial for embedded development, especially when working with a memory-constrained JavaScript engine.

IRAM (Instruction RAM): Fast internal RAM used for interrupt handlers and performance-critical code. Limited to around 128KB on ESP32-S3. Code placed in IRAM executes faster than code in flash because there is no cache miss penalty.

DRAM (Data RAM): General-purpose RAM for data storage. The ESP32-S3 has around 512KB of SRAM, divided into multiple banks. DRAM is used for the heap, stack, and static variables.

IROM (Instruction ROM): Code stored in external flash memory. Most application code resides here. The CPU fetches instructions through a cache, so frequently executed code benefits from cache hits.

DROM (Data ROM): Read-only data stored in flash, such as string literals and const arrays. Like IROM, access goes through a cache.

RTC Memory: A small amount of memory (around 8KB) that retains its contents during deep sleep. Useful for storing state across sleep cycles.

The ESP32 uses a Memory Management Unit (MMU) to map flash memory into the CPU's address space. This allows the CPU to execute code directly from flash (execute-in-place) without copying it to RAM first.

1.2. The MicroQuickJS Engine

MicroQuickJS is a minimal JavaScript engine designed specifically for embedded systems. It is derived from QuickJS, which was created by Fabrice Bellard and Charlie Gordon.

Design Philosophy

MicroQuickJS makes several design trade-offs to achieve its small footprint:

Pre-allocated Memory Pool: Instead of using malloc/free, MicroQuickJS operates on a fixed-size memory buffer provided by the application. All JavaScript objects, strings, and execution state are allocated from this pool. This eliminates heap fragmentation and makes memory usage predictable.

Bytecode Interpreter: MicroQuickJS compiles JavaScript source code into a compact bytecode format and then interprets this bytecode. This is slower than just-in-time (JIT) compilation but requires far less memory and code complexity.

Mark-and-Sweep Garbage Collector: The engine uses a simple mark-and-sweep garbage collector to reclaim unused objects. The GC runs periodically or when memory pressure is detected. Unlike generational or incremental collectors, mark-and-sweep is straightforward to implement and has predictable behavior.

Minimal Standard Library: MicroQuickJS includes only the essential JavaScript built-ins (Object, Array, Function, String, Number, Boolean). Advanced features like regular expressions, JSON, and Date are optional and can be excluded to save memory.

No JIT Compilation: JIT compilers generate native machine code at runtime, which requires significant memory for code caches and complex optimization passes. MicroQuickJS avoids this complexity entirely.

The JavaScript Value Representation

In MicroQuickJS, every JavaScript value is represented by a `JSValue` structure. This structure uses a technique called “NaN-boxing” to pack different value types into a single 64-bit word.

JavaScript has several primitive types: `undefined`, `null`, `boolean`, `number`, `string`, and `object`. In a naive implementation, you might use a tagged union:

```
struct JSValue {
    enum { TYPE_UNDEFINED, TYPE_NULL, TYPE_BOOLEAN, ... } type;
    union {
        bool boolean_value;
        double number_value;
        char *string_pointer;
        // ...
    } data;
};
```

This approach wastes space because the type tag and the union together take more than 8 bytes. NaN-boxing is a clever optimization that exploits the fact that IEEE 754 double-precision floats have many possible representations for “Not-a-Number” (NaN).

A double is 64 bits: 1 sign bit, 11 exponent bits, and 52 mantissa bits. A NaN has all exponent bits set to 1 and a non-zero mantissa. There are 2^52 possible NaN values, but JavaScript only needs one. MicroQuickJS uses the unused NaN values to encode other types.

For example:

- If the value is a valid double (not NaN), it represents a JavaScript number.
- If the value is a NaN with a specific bit pattern, the lower bits encode a pointer to a string or object.
- Special NaN patterns represent `undefined`, `null`, `true`, and `false`.

This technique allows `JSValue` to fit in a single 64-bit word, reducing memory usage and improving cache locality.

The Bytecode Format

When JavaScript source code is parsed, it is compiled into a bytecode format. Bytecode is a low-level, stack-based instruction set that the interpreter can execute efficiently.

Consider this JavaScript code:

```
var x = 10;
var y = 20;
var z = x + y;
```

The bytecode might look like this (simplified):

```
PUSH_CONST 10      // Push 10 onto the stack
SET_VAR x          // Store top of stack in variable x
PUSH_CONST 20      // Push 20 onto the stack
SET_VAR y          // Store top of stack in variable y
GET_VAR x          // Push value of x onto the stack
GET_VAR y          // Push value of y onto the stack
ADD                // Pop two values, add them, push result
SET_VAR z          // Store result in variable z
```

The bytecode is stored in a compact binary format, with each instruction taking 1-4 bytes depending on its operands. This is much smaller than the original source code and faster to execute than re-parsing the source every time.

The Execution Model

MicroQuickJS uses a stack-based virtual machine to execute bytecode. The VM maintains several key data structures:

Value Stack: A stack of `JSValue` objects used for intermediate results during expression evaluation. When you add two numbers, both are pushed onto the stack, the ADD instruction pops them, computes the sum, and pushes the result.

Call Stack: A stack of activation records (also called stack frames) that track function calls. Each frame contains the function's local variables, the return address (the bytecode offset to resume after the call), and a pointer to the previous frame.

Global Object: A JavaScript object that holds all global variables and built-in functions. When you access a global variable like `Math`, the VM looks it up in the global object.

Context: The `JSContext` structure holds all the state for a JavaScript execution environment, including the value stack, call stack, global object, and memory pool.

When the interpreter executes bytecode, it follows a simple fetch-decode-execute loop:

1. **Fetch**: Read the next bytecode instruction from the current position.
2. **Decode**: Determine what operation the instruction represents and extract any operands.
3. **Execute**: Perform the operation, which might involve pushing/popping values from the stack, calling functions, or allocating objects.
4. **Repeat**: Move to the next instruction and repeat.

This model is conceptually similar to how a physical CPU works, but at a higher level of abstraction.

Memory Management and Garbage Collection

One of the most critical aspects of MicroQuickJS is its memory management. Since the engine operates on a fixed-size memory pool, it must carefully allocate and reclaim memory to avoid running out of space.

Allocation: When JavaScript code creates a new object, string, or array, the engine allocates memory from the pool. The allocator uses a simple bump-pointer or free-list strategy. If there is not enough free memory, the garbage collector is triggered.

Garbage Collection: The mark-and-sweep collector works in two phases:

1. **Mark Phase**: Starting from the root set (global variables, local variables on the call stack, and the value stack), the collector traverses all reachable objects and marks them as “alive.” Any object that is not reachable is considered garbage.
2. **Sweep Phase**: The collector scans the entire memory pool and reclaims any objects that were not marked. These objects are added back to the free list, making their memory available for future allocations.

The GC is conservative in the sense that it assumes any value that looks like a pointer might be a pointer. This prevents the collector from accidentally freeing objects that are still in use, but it can also lead to memory leaks if non-pointer data happens to look like a valid pointer.

Reference Counting: In addition to mark-and-sweep, MicroQuickJS uses reference counting for certain objects. Each object has a reference count that tracks how many references to it exist. When the count drops to zero, the object is immediately freed. This provides deterministic cleanup for resources like file handles.

Part II: Practical Integration

Now that we have covered the foundational concepts, let's walk through the practical steps of integrating MicroQuickJS into an ESP-IDF project.

2.1. Setting Up the Development Environment

Before you can build firmware, you need to set up the ESP-IDF development environment.

Step 1: Install Prerequisites

ESP-IDF requires several tools and libraries. On Ubuntu/Debian, you can install them with:

```
sudo apt-get update
sudo apt-get install git wget flex bison gperf python3 python3-pip \
    python3-venv cmake ninja-build ccache libffi-dev libssl-dev \
    dfu-util libusb-1.0-0
```

Step 2: Clone ESP-IDF

Clone the ESP-IDF repository and check out the desired version:

```
mkdir -p ~/esp
cd ~/esp
git clone -b v5.5.2 --recursive https://github.com/espressif/esp-idf.git
esp-idf-v5.5.2
cd esp-idf-v5.5.2
```

The `--recursive` flag ensures that all submodules are cloned. ESP-IDF has many submodules for components like FreeRTOS, lwIP, and mbedTLS.

Step 3: Install ESP-IDF Tools

Run the installation script to download the toolchains and other tools:

```
./install.sh esp32s3
```

This downloads the Xtensa toolchain, OpenOCD for debugging, and other utilities. The tools are installed in `~/.espressif`.

Step 4: Set Up the Environment

Every time you open a new terminal, you need to source the `export.sh` script to set up the environment:

```
. ~/esp/esp-idf-v5.5.2/export.sh
```

This modifies your `PATH` and sets environment variables like `IDF_PATH`.

2.2. Creating the Project Structure

Now, let's create a new ESP-IDF project for our JavaScript REPL.

Step 1: Create the Project Directory

```
mkdir -p ~/esp32-mqjs-repl/mqjs-repl
cd ~/esp32-mqjs-repl/mqjs-repl
```

Step 2: Create the Main Component

Every ESP-IDF project must have a `main` component that contains the `app_main()` function.

```
mkdir main
```

Create `main/CMakeLists.txt`:

```
idf_component_register(SRCS "main.c"
                    INCLUDE_DIRS ".")
```

Create a minimal `main/main.c`:

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void app_main(void) {
    printf("Hello from ESP32-S3!\\n");
}
```

Step 3: Create the Top-Level CMakeLists.txt

Create `CMakeLists.txt` in the project root:

```
 cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(mqjs-repl)
```

This tells CMake to include the ESP-IDF build system and defines the project name.

Step 4: Build the Project

Set the target and build:

```
idf.py set-target esp32s3
idf.py build
```

If everything is set up correctly, the project should compile successfully.

2.3. Adding MicroQuickJS as a Component

Now we will integrate MicroQuickJS into the project.

Step 1: Clone MicroQuickJS

```
mkdir components
cd components
git clone https://github.com/bellard/mquickjs.git
cd ..
```

Step 2: Create the Component Build File

Create `components/mquickjs/CMakeLists.txt`:

```
idf_component_register(
    SRCS "mquickjs.c" "cutils.c" "libm.c" "dtoa.c"
    INCLUDE_DIRS "."
    PRIV_INCLUDE_DIRS "."
)

# Disable warnings that cause errors with third-party code
target_compile_options(${COMPONENT_LIB} PRIVATE
    "-Wno-error=format-truncation"
    "-Wno-error=format"
)
```

This tells ESP-IDF to compile the MicroQuickJS source files and include the headers.

Step 3: Verify the Build

Rebuild the project to ensure the component is recognized:

```
idf.py build
```

You should see MicroQuickJS being compiled as part of the build process.

2.4. Implementing the JavaScript REPL

Now we will implement a Read-Eval-Print Loop (REPL) that allows interactive JavaScript execution over the serial console.

Step 1: Initialize the UART

The UART (Universal Asynchronous Receiver-Transmitter) is the serial communication peripheral. We will use UART0, which is typically connected to the USB-to-serial converter on development boards.

In `main/main.c`:

```
#include "driver/uart.h"

#define UART_NUM UART_NUM_0
#define BUF_SIZE (1024)

void init_uart(void) {
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
    };

    uart_param_config(UART_NUM, &uart_config);
    uart_driver_install(UART_NUM, BUF_SIZE * 2, 0, 0, NULL, 0);
}
```

Step 2: Initialize the JavaScript Engine

MicroQuickJS requires a pre-allocated memory buffer. We will use a 64KB buffer.

```
#include "mquickjs.h"

#define JS_MEM_SIZE (64 * 1024)
static uint8_t js_mem_buf[JS_MEM_SIZE];
static JSContext *js_ctx = NULL;

void init_js_engine(void) {
    js_ctx = JS_NewContext(js_mem_buf, JS_MEM_SIZE, NULL);
    if (!js_ctx) {
        printf("Failed to create JavaScript context\\n");
        return;
    }
    printf("JavaScript engine initialized\\n");
}
```

The third parameter to `JS_NewContext` is a pointer to a standard library definition. We pass `NULL` for now, which means no built-in functions will be available.

Step 3: Implement the REPL Task

FreeRTOS uses tasks for concurrency. We will create a task that reads input from the UART, evaluates it as JavaScript, and prints the result.

```
void repl_task(void *pvParameters) {
    uint8_t data[BUF_SIZE];
    char line_buf[512];
    int line_pos = 0;

    printf("js> ");
    fflush(stdout);

    while (1) {
        int len = uart_read_bytes(UART_NUM, data, BUF_SIZE, portMAX_DELAY);
        if (len > 0) {
            for (int i = 0; i < len; i++) {
                char c = data[i];

                if (c == '\\n' || c == '\\r') {
                    if (line_pos > 0) {
                        printf("\\n");
                        line_buf[line_pos] = '\\0';

                        // Evaluate the JavaScript code
                        JSValue val = JS_Eval(js_ctx, line_buf, line_pos, "
<input>", 0);

                        if (JS_IsException(val)) {
                            JSValue obj = JS_GetException(js_ctx);
                            printf("Error: ");
                            JS_PrintValueF(js_ctx, obj, JS_DUMP_LONG);
                            printf("\\n");
                        } else if (!JS_IsUndefined(val)) {
                            JS_PrintValueF(js_ctx, val, JS_DUMP_LONG);
                            printf("\\n");
                        }

                        line_pos = 0;
                    }
                    printf("js> ");
                    fflush(stdout);
                } else if (c == '\\b' || c == 127) {
                    if (line_pos > 0) {
                        line_pos--;
                        printf("\\b \\b");
                        fflush(stdout);
                    }
                } else if (c >= 32 && c < 127) {
                    if (line_pos < sizeof(line_buf) - 1) {
```

```
                        line_buf[line_pos++] = c;
                        putchar(c);
                        fflush(stdout);
                    }
                }
            }
        }
    }
}
```

Step 4: Start the REPL Task in app_main

```
void app_main(void) {
    init_uart();
    init_js_engine();

    xTaskCreate(repl_task, "repl_task", 8192, NULL, 5, NULL);
}
```

Step 5: Build and Test

Build the firmware:

```
idf.py build
```

Flash it to your ESP32-S3 board:

```
idf.py -p /dev/ttyUSB0 flash monitor
```

Or run it in QEMU:

```
idf.py qemu monitor
```

You should see the `js>` prompt. Try entering JavaScript expressions like `1+2` or `var x = 10`.

2.5. Adding Persistent Storage with SPIFFS

SPIFFS (SPI Flash File System) allows us to store JavaScript libraries on the flash memory and load them on startup.

Step 1: Create a Custom Partition Table

Create `partitions.csv` in the project root:

```
# Name,   Type, SubType, Offset,  Size, Flags
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 0x100000,
storage,  data, spiffs,  0x110000,0xF0000,
```

This defines a 960KB partition for SPIFFS starting at offset 0x110000.

Step 2: Configure the Project

Add the following to `sdkconfig`:

```
CONFIG_PARTITION_TABLE_CUSTOM=y
CONFIG_PARTITION_TABLE_CUSTOM_FILENAME="partitions.csv"
CONFIG_PARTITION_TABLE_FILENAME="partitions.csv"
```

Or use `idf.py menuconfig` and navigate to **Partition Table**.

Step 3: Initialize SPIFFS in Firmware

Add the following to `main.c`:

```
#include "esp_spiffs.h"

esp_err_t init_spiffs(void) {
    esp_vfs_spiffs_conf_t conf = {
        .base_path = "/spiffs",
        .partition_label = "storage",
        .max_files = 5,
        .format_if_mount_failed = true
    };

    esp_err_t ret = esp_vfs_spiffs_register(&conf);
    if (ret != ESP_OK) {
        printf("Failed to initialize SPIFFS (%s)\\n", esp_err_to_name(ret));
        return ret;
    }

    size_t total = 0, used = 0;
    ret = esp_spiffs_info("storage", &total, &used);
    if (ret == ESP_OK) {
        printf("SPIFFS: total=%d bytes, used=%d bytes\\n", total, used);
    }

    return ESP_OK;
}
```

Call `init_spiffs()` in `app_main()` before initializing the JavaScript engine.

Step 4: Create and Load Libraries

You can create JavaScript libraries on the first boot:

```
void create_example_library(void) {
    mkdir("/spiffs/autoload", 0755);

    FILE *f = fopen("/spiffs/autoload/math.js", "w");
    if (f) {
        fprintf(f, "var MathUtils = {};\\n");
        fprintf(f, "MathUtils.square = function(x) { return x * x; };\\n");
        fclose(f);
    }
}
```

And load them on startup:

```
void load_js_file(const char *filepath) {
    FILE *f = fopen(filepath, "r");
    if (!f) return;

    fseek(f, 0, SEEK_END);
    long fsize = ftell(f);
    fseek(f, 0, SEEK_SET);

    char *content = malloc(fsize + 1);
    fread(content, 1, fsize, f);
    fclose(f);
    content[fsize] = '\\0';

    JSValue val = JS_Eval(js_ctx, content, fsize, filepath, 0);
    if (JS_IsException(val)) {
        JSValue obj = JS_GetException(js_ctx);
        printf("Error loading %s: ", filepath);
        JS_PrintValueF(js_ctx, obj, JS_DUMP_LONG);
        printf("\\n");
    }

    free(content);
}

void load_autoload_libraries(void) {
    DIR *dir = opendir("/spiffs/autoload");
    if (!dir) return;

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (strstr(entry->d_name, ".js")) {
            char filepath[256];
            snprintf(filepath, sizeof(filepath), "/spiffs/autoload/%s",
entry->d_name);
            load_js_file(filepath);
        }
    }

    closedir(dir);
}
```

Now, any `.js` files in `/spiffs/autoload` will be automatically loaded when the firmware boots.

Part III: Under the Hood

This section provides a deep dive into the internals of MicroQuickJS, explaining how the engine works at a low level.

3.1. The Compilation Pipeline

When you call `JS_Eval()`, the JavaScript source code goes through several stages before it can be executed.

Lexical Analysis (Tokenization)

The first step is to break the source code into tokens. A token is the smallest meaningful unit of code, such as a keyword (`var`, `function`), an identifier (`myVariable`), an operator (`+`, `=`), or a literal (`42`, `"hello"`).

The lexer (also called a scanner or tokenizer) reads the source code character by character and groups them into tokens. For example, the code:

```
var x = 10 + 20;
```

Would be tokenized as:

```
KEYWORD(var)
IDENTIFIER(x)
OPERATOR(=)
NUMBER(10)
OPERATOR(+)
NUMBER(20)
SEMICOLON
```

The lexer also handles comments, whitespace, and string escapes.

Syntax Analysis (Parsing)

The parser takes the stream of tokens and builds an Abstract Syntax Tree (AST). The AST is a hierarchical representation of the code's structure.

For the code above, the AST might look like:

```
VariableDeclaration
  ├─ Identifier: x
  └─ BinaryExpression
       ├─ Operator: +
       ├─ Left: Literal(10)
       └─ Right: Literal(20)
```

The parser enforces the grammar rules of JavaScript. If the code has a syntax error (like a missing semicolon or unmatched parenthesis), the parser will detect it and report an error.

MicroQuickJS uses a recursive descent parser, which is a top-down parsing technique. The parser has a function for each grammar rule, and these functions call each other recursively to build the AST.

Bytecode Generation

Once the AST is built, the compiler traverses it and generates bytecode. Each node in the AST corresponds to one or more bytecode instructions.

For example, the `BinaryExpression` node with operator `+` would generate:

```
PUSH_CONST 10
PUSH_CONST 20
ADD
```

The compiler also performs some optimizations at this stage, such as constant folding (evaluating `10 + 20` at compile time instead of runtime) and dead code elimination (removing code that can never be executed).

The bytecode is stored in a `JSFunction` object, which also contains metadata like the function's name, parameter count, and local variable names.

3.2. The Bytecode Interpreter

The interpreter is the heart of the MicroQuickJS engine. It executes bytecode instructions one at a time.

The Instruction Set

MicroQuickJS has around 100 different bytecode instructions. Some common ones include:

- **PUSH_CONST**: Push a constant value onto the stack.
- **PUSH_VAR**: Push the value of a variable onto the stack.
- **SET_VAR**: Pop a value from the stack and store it in a variable.
- **ADD, SUB, MUL, DIV**: Arithmetic operations.
- **CALL**: Call a function.
- **RETURN**: Return from a function.
- **JUMP**: Unconditional jump to a different bytecode offset.
- **JUMP_IF_FALSE**: Conditional jump based on a boolean value.

Each instruction is encoded as a single byte (the opcode) followed by zero or more operand bytes.

The Execution Loop

The interpreter's main loop looks like this (simplified):

```
void execute_bytecode(JSContext *ctx, JSFunction *func) {
    uint8_t *pc = func->bytecode;  // Program counter
    JSValue *sp = ctx->stack_top;  // Stack pointer

    while (1) {
        uint8_t opcode = *pc++;

        switch (opcode) {
            case OP_PUSH_CONST: {
                uint16_t index = read_u16(&pc);
                *sp++ = func->constants[index];
                break;
            }
            case OP_ADD: {
                JSValue b = *--sp;
                JSValue a = *--sp;
                *sp++ = js_add(ctx, a, b);
                break;
            }
            case OP_RETURN: {
                JSValue result = *--sp;
                return result;
            }
            // ... other opcodes
        }
    }
}
```

The program counter (`pc`) points to the current instruction. The stack pointer (`sp`) points to the top of the value stack. Each instruction manipulates these pointers and the stack.

Function Calls

When a function is called, the interpreter:

1. **Pushes a new stack frame**: This frame contains the function's local variables, the return address (the bytecode offset to resume after the call), and a pointer to the previous frame.

2. **Evaluates arguments**: The arguments are evaluated and pushed onto the stack.

3. **Jumps to the function's bytecode**: The program counter is set to the beginning of the function's bytecode.
4. **Executes the function**: The interpreter runs the function's bytecode.
5. **Returns**: When the function returns, the interpreter pops the stack frame, restores the previous program counter, and pushes the return value onto the stack.

This mechanism allows for recursion and nested function calls.

3.3. Memory Management Internals

Memory management is one of the most complex parts of any JavaScript engine. MicroQuickJS uses a combination of techniques to manage memory efficiently.

The Memory Pool

The memory pool is a contiguous block of memory provided by the application. It is divided into two main regions:

Object Heap: This is where JavaScript objects, strings, and arrays are allocated. The heap uses a free-list allocator, which maintains a linked list of free memory blocks. When an allocation is requested, the allocator searches the free list for a block of sufficient size.

Stack: The value stack and call stack grow from the opposite end of the memory pool. The stack grows downward (toward lower addresses) while the heap grows upward. If the two meet, the engine runs out of memory.

Object Representation

Each JavaScript object is represented by a `JSObject` structure:

```
struct JSObject {
    uint32_t ref_count;
    uint32_t flags;
    JSClass *class;
    JSProperty *properties;
    // ... other fields
};
```

The `ref_count` field tracks how many references to the object exist. When it drops to zero, the object can be freed immediately.

The `properties` field is a pointer to a hash table or array that stores the object's properties. Each property has a name (a string) and a value (a `JSValue`).

Garbage Collection Algorithm

The mark-and-sweep garbage collector runs in two phases:

Mark Phase:

1. Start with the root set: global variables, local variables on the call stack, and values on the value stack.
2. For each root, mark the object as "alive" by setting a flag.
3. Recursively traverse all objects reachable from the roots, marking each one.

Sweep Phase:

1. Scan the entire object heap.
2. For each object, check if it is marked.
3. If not marked, the object is garbage. Free it and add its memory back to the free list.
4. If marked, clear the mark flag for the next GC cycle.

The GC is triggered when:

- An allocation fails due to insufficient free memory.
- The amount of free memory falls below a threshold.
- The application explicitly requests a GC (e.g., during idle time).

Memory Fragmentation

Over time, the heap can become fragmented, with small free blocks scattered throughout. This makes it difficult to allocate large objects. MicroQuickJS mitigates fragmentation by:

- **Coalescing free blocks**: When two adjacent blocks are freed, they are merged into a single larger block.
- **Compaction**: Periodically, the GC can move objects to consolidate free space. However, compaction is expensive and not always performed.

3.4. The Standard Library

The standard library provides built-in JavaScript objects like `Object`, `Array`, `String`, and `Math`. In MicroQuickJS, the standard library is optional and can be customized.

The Stdlib Table

The standard library is defined as a table of bytecode and C functions. When the engine initializes, it loads this table and creates the built-in objects.

For example, the `Array` object might have methods like `push`, `pop`, `slice`, and `map`. Each method is implemented as either bytecode (for simple operations) or a native C function (for performance-critical operations).

Custom Standard Libraries

You can create a custom standard library by defining a `JSSTDLibraryDef` structure:

```
const JSSTDLibraryDef my_stdlib = {
    .stdlib_table = my_bytecode_table,
    .c_function_table = my_c_functions,
    .c_finalizer_table = my_finalizers,
    // ... other fields
};
```

Then pass this to `JS_NewContext`:

```
js_ctx = JS_NewContext(js_mem_buf, JS_MEM_SIZE, &my_stdlib);
```

This allows you to include only the functions you need, reducing memory usage.

Part IV: Advanced Features

This section covers advanced topics like exposing custom C functions to JavaScript, interfacing with hardware, and optimizing performance.

4.1. Exposing C Functions to JavaScript

One of the most powerful features of an embedded JavaScript engine is the ability to call native C functions from JavaScript. This allows you to expose hardware peripherals, sensors, and other low-level functionality.

Defining a C Function

A C function that can be called from JavaScript must have a specific signature:

```
JSValue my_function(JSContext *ctx, JSValueConst this_val, int argc,
JSValueConst *argv);
```

- `ctx`: The JavaScript context.
- `this_val`: The value of `this` in the function call.
- `argc`: The number of arguments.
- `argv`: An array of argument values.

The function returns a `JSValue`, which is the result of the function.

Example: GPIO Control

Let's create a function to control a GPIO pin:

```c
#include "driver/gpio.h"

JSValue js_gpio_set(JSContext *ctx, JSValueConst this_val, int argc,
JSValueConst *argv) {
    if (argc < 2) {
        return JS_ThrowTypeError(ctx, "Expected 2 arguments: pin, level");
    }

    int32_t pin, level;
    if (JS_ToInt32(ctx, &pin, argv[0]) < 0) {
        return JS_EXCEPTION;
    }
    if (JS_ToInt32(ctx, &level, argv[1]) < 0) {
        return JS_EXCEPTION;
    }

    gpio_set_direction(pin, GPIO_MODE_OUTPUT);
    gpio_set_level(pin, level);

    return JS_UNDEFINED;
}
```

Registering the Function

To make the function available in JavaScript, you need to register it with the global object:

```c
void register_gpio_functions(JSContext *ctx) {
    JSValue global = JS_GetGlobalObject(ctx);

    JS_SetPropertyStr(ctx, global, "gpioSet",
        JS_NewCFunction(ctx, js_gpio_set, "gpioSet", 2));

    JS_FreeValue(ctx, global);
}
```

Now you can call `gpioSet(2, 1)` from JavaScript to set GPIO pin 2 to high.

4.2. Interfacing with Hardware Peripherals

Embedded systems often need to interact with hardware peripherals like sensors, displays, and communication interfaces. You can expose these peripherals to JavaScript using the technique described above.

Example: I2C Sensor

Let's create a JavaScript API for reading from an I2C temperature sensor:

```
#include "driver/i2c.h"

#define I2C_MASTER_NUM I2C_NUM_0
#define SENSOR_ADDR 0x48

JSValue js_read_temperature(JSContext *ctx, JSValueConst this_val, int argc,
JSValueConst *argv) {
    uint8_t data[2];

    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (SENSOR_ADDR << 1) | I2C_MASTER_READ, true);
    i2c_master_read(cmd, data, 2, I2C_MASTER_LAST_NACK);
    i2c_master_stop(cmd);

    esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, 1000 /
portTICK_PERIOD_MS);
    i2c_cmd_link_delete(cmd);

    if (ret != ESP_OK) {
        return JS_ThrowInternalError(ctx, "I2C read failed");
    }

    int16_t raw = (data[0] << 8) | data[1];
    double temperature = raw * 0.0625;  // Convert to Celsius

    return JS_NewFloat64(ctx, temperature);
}
```

Register it:

```
JS_SetPropertyStr(ctx, global, "readTemperature",
    JS_NewCFunction(ctx, js_read_temperature, "readTemperature", 0));
```

Now you can call `var temp = readTemperature();` from JavaScript.

4.3. Asynchronous Operations

JavaScript is inherently single-threaded, but embedded systems often need to handle asynchronous events like interrupts, timers, and network packets. You can implement asynchronous operations using callbacks or promises.

Example: Timer with Callback

```
static JSValue timer_callback = JS_UNDEFINED;

void timer_isr(void *arg) {
    // This runs in an interrupt context
    // We need to defer the callback to the main task
    xTaskNotifyGive((TaskHandle_t)arg);
}

JSValue js_set_timer(JSContext *ctx, JSValueConst this_val, int argc,
JSValueConst *argv) {
    if (argc < 2) {
        return JS_ThrowTypeError(ctx, "Expected 2 arguments: delay_ms,
callback");
    }

    int32_t delay_ms;
    if (JS_ToInt32(ctx, &delay_ms, argv[0]) < 0) {
        return JS_EXCEPTION;
    }

    if (!JS_IsFunction(ctx, argv[1])) {
        return JS_ThrowTypeError(ctx, "Second argument must be a function");
    }

    // Store the callback
    JS_FreeValue(ctx, timer_callback);
    timer_callback = JS_DupValue(ctx, argv[1]);

    // Set up the timer
    // ... (ESP-IDF timer configuration)

    return JS_UNDEFINED;
}
```

When the timer fires, you would call the callback from the main task:

```
void main_task(void *arg) {
    while (1) {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        if (!JS_IsUndefined(timer_callback)) {
            JSValue result = JS_Call(js_ctx, timer_callback, JS_UNDEFINED,
0, NULL);
            if (JS_IsException(result)) {
                // Handle error
            }
            JS_FreeValue(js_ctx, result);
        }
    }
}
```

4.4. Performance Optimization

JavaScript on embedded systems will never be as fast as native C code, but there are several techniques to improve performance.

Minimize Allocations

Every object allocation can trigger garbage collection. Minimize allocations by:

- Reusing objects instead of creating new ones.
- Using primitive types (numbers, booleans) instead of objects when possible.
- Pre-allocating arrays to their final size.

Avoid String Concatenation

String concatenation creates new string objects. Instead of:

```
var str = "";
for (var i = 0; i < 100; i++) {
    str = str + i + ",";
}
```

Use an array and join:

```
var arr = [];
for (var i = 0; i < 100; i++) {
    arr[i] = i;
}
var str = arr.join(",");
```

Use Native Functions

For performance-critical operations, implement them in C and expose them to JavaScript. Native functions are much faster than JavaScript code.

Profile Your Code

Use the ESP-IDF profiling tools to identify bottlenecks. The `esp_timer_get_time()` function provides microsecond-resolution timing.

Part V: Optimization and Best Practices

This section provides guidelines for writing efficient, maintainable JavaScript code for embedded systems.

5.1. Memory Management Best Practices

Know Your Memory Budget: Always be aware of how much memory your application uses. Use `esp_get_free_heap_size()` to monitor free memory.

Avoid Memory Leaks: Make sure to free any resources you allocate. In JavaScript, this means avoiding circular references that prevent garbage collection.

Limit Recursion Depth: Deep recursion can exhaust the call stack. Use iteration instead of recursion when possible.

Preallocate Buffers: If you know the size of a buffer in advance, allocate it once and reuse it.

5.2. Code Organization

Modularize Your Code: Break your application into small, reusable modules. Each module should have a clear purpose and interface.

Use Libraries: Store commonly used functions in libraries on the flash file system. This keeps your main code clean and allows for easy updates.

Document Your Code: Add comments to explain complex logic. Future you (or other developers) will thank you.

5.3. Debugging Techniques

Use Logging: The `printf` function is your friend. Add log statements to trace execution and inspect variables.

Check Return Values: Always check the return value of functions that can fail. Handle errors gracefully.

Use the REPL: The interactive REPL is invaluable for testing small code snippets and debugging.

GDB Debugging: ESP-IDF supports GDB debugging over JTAG. This allows you to set breakpoints, step through code, and inspect memory.

5.4. Security Considerations

Validate Input: Always validate user input to prevent injection attacks or buffer overflows.

Limit Execution Time: Implement a timeout mechanism to prevent infinite loops or long-running scripts from hanging the system.

Sandbox Untrusted Code: If you allow users to upload JavaScript code, run it in a sandboxed environment with limited access to system resources.

Conclusion

This guide has provided a comprehensive exploration of integrating MicroQuickJS with the ESP-IDF framework for the ESP32 family of microcontrollers. We have covered the foundational concepts, practical integration steps, internal workings of the engine, advanced features, and best practices for optimization and debugging.

JavaScript on embedded systems opens up new possibilities for rapid prototyping, dynamic behavior, and accessibility to a broader developer audience. While it may not replace C/C++ for all embedded applications, it provides a powerful tool for certain use cases, especially those that benefit from dynamic scripting and ease of development.

By understanding both the high-level APIs and the low-level internals, you are now equipped to build sophisticated embedded applications that leverage the power and flexibility of JavaScript while respecting the constraints of resource-limited microcontrollers.

Happy hacking, and may your embedded JavaScript adventures be both productive and enlightening!

References

1. Espressif Systems. “ESP-IDF Programming Guide.” https://docs.espressif.com/projects/esp-idf/en/v5.5.2/
2. Bellard, Fabrice. “QuickJS JavaScript Engine.” https://bellard.org/quickjs/
3. Bellard, Fabrice. “MicroQuickJS.” https://github.com/bellard/mquickjs
4. FreeRTOS. “FreeRTOS Real-Time Operating System.” https://www.freertos.org/
5. ECMA International. “ECMAScript Language Specification, 5.1 Edition.” https://www.ecma-international.org/ecma-262/5.1/