ESP32 & MicroQuickJS: A Developer’s Playbook

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

Introduction

This playbook provides a comprehensive guide for setting up an ESP-IDF development environment, integrating the MicroQuickJS JavaScript engine, and building a serial REPL firmware for the ESP32-S3. It covers everything from initial setup to advanced features like flash storage, offering both practical steps and in-depth explanations of the underlying concepts.

Who This Guide Is For

- **Embedded Developers** new to ESP32 or ESP-IDF
- **JavaScript Developers** interested in embedded systems
- **Hobbyists and Makers** looking to run JavaScript on microcontrollers

What You Will Learn

- How to set up the ESP-IDF v5.5.2 development environment
- How to build and run a simple “Hello, World!” firmware in QEMU
- How to integrate the MicroQuickJS engine into an ESP-IDF project
- How to build a fully functional JavaScript REPL
- How to add flash storage with SPIFFS for persistent libraries
- How to debug and troubleshoot common issues

Part 1: ESP-IDF Setup and “Hello, World!”

This section walks you through the complete process of setting up the ESP-IDF environment and running your first firmware in the QEMU emulator.

1.1. Installing ESP-IDF v5.5.2

The ESP-IDF (Espressif IoT Development Framework) is the official development framework for the ESP32. We will use version 5.5.2 for this guide.

Step 1: Clone the Repository

It is recommended to use a shallow clone to speed up the download process.

```
# Clone the specific version with a shallow depth
git clone -b v5.5.2 --depth 1 https://github.com/espressif/esp-idf.git esp-idf-v5.5.2

# Navigate into the directory
cd esp-idf-v5.5.2
```

Step 2: Install ESP-IDF Tools

The `install.sh` script downloads and installs the necessary toolchains, compilers, and debuggers.

```
# Run the installation script for the esp32s3 target
./install.sh esp32s3
```

Step 3: Set Up the Environment

The `export.sh` script sets up the required environment variables. You will need to source this script in every new terminal session.

```
# Source the environment script
. ./export.sh
```

> ***Under the Hood***: *The `export.sh` script modifies your `PATH` to include the ESP-IDF toolchain and sets other variables like `IDF_PATH` to point to the framework directory. This allows you to use `idf.py` and other tools from anywhere.*

1.2. Creating a “Hello, World!” Project

Now that the environment is set up, let’s create a simple project.

Step 1: Create a New Project

ESP-IDF provides a template project that you can copy.

```
# Navigate to your workspace
cd ~

# Copy the hello_world example
cp -r $IDF_PATH/examples/get-started/hello_world ./

# Navigate into the project
cd hello_world
```

Step 2: Configure the Target

Set the target chip to ESP32-S3.

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

Step 3: Build the Firmware

This command compiles the project and creates a binary firmware file.

```
idf.py build
```

1.3. Running in QEMU

QEMU is an open-source machine emulator that can run firmware for various architectures, including the ESP32-S3.

Step 1: Install QEMU for ESP-IDF

If you haven't already, install the QEMU tools using the `install.sh` script.

```
# Navigate back to the ESP-IDF directory
cd ~/esp-idf-v5.5.2

# Install QEMU
./install.sh qemu
```

Step 2: Run the Firmware in QEMU

The `qemu` command in `idf.py` handles the entire process of building, creating a flash image, and launching the emulator.

```
# Navigate back to your project
cd ~/hello_world

# Run in QEMU with monitor
idf.py qemu monitor
```

You should see the "Hello, World!" output in the terminal, along with other boot messages.

> ***Under the Hood***: *The `qemu monitor` command does several things:*
>
> 1. ***Builds*** *the project if it has changed.*
> 2. ***Creates a flash image*** *(`qemu_flash.bin`) that combines the bootloader, partition table, and application binary.*
> 3. ***Launches QEMU*** *with the correct machine type (`-M esp32s3`) and loads the flash image.*
> 4. ***Starts `idf_monitor.py`*** *to connect to the QEMU serial port, allowing you to see the output and interact with the firmware.*

Part 2: Integrating MicroQuickJS

This section details how to integrate the MicroQuickJS engine into your ESP-IDF project to create a JavaScript REPL.

2.1. Understanding MicroQuickJS

MicroQuickJS is a lightweight, embeddable JavaScript engine derived from Fabrice Bellard's QuickJS. It is designed for microcontrollers with limited resources.

Key Features:

- **Small Footprint**: Can run with as little as 64KB of RAM.
- **ES5.1 Compliant**: Supports a modern subset of JavaScript.
- **No OS Dependencies**: Can run on bare metal or with an RTOS like FreeRTOS.
- **Simple C API**: Easy to integrate with C/C++ projects.

2.2. Creating an ESP-IDF Component

The best way to integrate a third-party library like MicroQuickJS is to create a custom ESP-IDF component.

Step 1: Create the Project Structure

```bash
# Create a new project for the REPL
cd ~
mkdir esp32-mqjs-repl && cd esp32-mqjs-repl

# Create a components directory
mkdir components

# Create the main directory
mkdir main
```

Step 2: Clone MicroQuickJS

```
# Clone the repository into the components directory
cd components
git clone https://github.com/bellard/mquickjs.git
cd ..
```

Step 3: Create the Component `CMakeLists.txt`

Create a `CMakeLists.txt` file inside the `components/mquickjs` directory to tell ESP-IDF how to build the engine.

```
 # /components/mquickjs/CMakeLists.txt

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

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

> ***Under the Hood**: `idf_component_register` is the key function for creating a component. It specifies the source files, include directories, and other build settings. We also disable certain warnings that can cause the build to fail with third-party code.*

2.3. Building the REPL Firmware

Now, let's create the main application logic for the REPL.

Step 1: Create the `main.c` File

This file will contain the core logic for initializing the UART, the JavaScript engine, and the REPL task.

```c
// /main/main.c - Abridged for clarity

#include "esp_system.h"
#include "esp_log.h"
#include "driver/uart.h"
#include "mquickjs.h"

// ... (other includes)

void app_main(void) {
    // Initialize UART
    uart_config_t uart_config = { ... };
    uart_param_config(UART_NUM_0, &uart_config);
    uart_driver_install(UART_NUM_0, ...);

    // Initialize JavaScript engine
    static uint8_t js_mem_buf[64 * 1024];
    JSContext *js_ctx = JS_NewContext(js_mem_buf, sizeof(js_mem_buf), NULL);

    // Create REPL task
    xTaskCreate(repl_task, "repl_task", 8192, NULL, 5, NULL);
}
```

Step 2: Create the Main `CMakeLists.txt`

Create a `CMakeLists.txt` file in the `main` directory.

```
 # /main/CMakeLists.txt

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

Step 3: Build and Run

Now you can build and run the REPL firmware in QEMU.

```
cd ~/esp32-mqjs-repl
idf.py set-target esp32s3
idf.py build
idf.py qemu monitor
```

You should see the REPL prompt `js>` in the terminal.

Part 3: Adding Flash Storage with SPIFFS

This section explains how to add a flash file system to store and load JavaScript libraries on startup.

3.1. Understanding SPIFFS

SPIFFS (SPI Flash File System) is a lightweight file system designed for NOR flash devices, which are common in microcontrollers. It is ideal for storing small files, configuration, and scripts.

3.2. Creating a Custom Partition Table

To use SPIFFS, you need to define a partition on the flash memory.

Step 1: Create `partitions.csv`

Create a file named `partitions.csv` in your 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, # 960KB for storage
```

Step 2: Configure the Project

Run `idf.py menuconfig` and navigate to **Partition Table > Partition Table** and select **Custom partition table CSV**. Then set the **Custom partition CSV file** to `partitions.csv`.

3.3. Initializing SPIFFS in Firmware

Now, add the code to initialize the SPIFFS file system in your `main.c`.

```c
// In main.c

#include "esp_spiffs.h"

static 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);
    // ... (error handling)
    return ret;
}

// In app_main()
init_spiffs();
```

3.4. Loading Libraries from Flash

Now you can create an `autoload` directory on the flash and load any `.js` files from it on startup.

Step 1: Create Example Libraries

In your `main.c`, you can create some example libraries on the first boot.

```
// In main.c

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

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

Step 2: Load Libraries on Startup

In `app_main()`, after initializing the JavaScript engine, you can iterate through the `autoload` directory and load each file.

```
// In main.c

void load_autoload_libraries() {
    DIR *dir = opendir("/spiffs/autoload");
    // ... (iterate through directory)

    // For each file, load and evaluate
    char *content = load_file(filepath, &size);
    JS_Eval(js_ctx, content, size, ...);
    free(content);
}
```

Conclusion

This playbook has provided a comprehensive guide to building a fully functional JavaScript REPL on the ESP32-S3 with flash storage. You have learned how to:

- Set up the ESP-IDF environment
- Build and run firmware in QEMU

- Integrate the MicroQuickJS engine
- Implement a serial REPL
- Add persistent storage with SPIFFS

With this foundation, you can now explore more advanced features like adding custom C functions to your JavaScript environment, interacting with hardware peripherals, and building complex embedded applications with the power and flexibility of JavaScript.

Happy Hacking!