Write frequency in SPIFFS in ESP32 CASESTUDY

---

1. Role of FS vs SPIFFS

- **FS (FileSystem wrapper)**: Provides the **interface** (like `open`, `read`, `write`, `remove`, etc.).
  It doesn't know how to talk to flash directly.

- **SPIFFS (implementation)**: Provides the actual **backend driver** that knows how to store bytes into ESP32's SPI flash using the SPIFFS filesystem logic (metadata, wear-leveling, etc.).

So when you do `SPIFFS.open(...)`, you're telling the FS wrapper:
"Use the SPIFFS implementation for this file."

---

2. What happens in `file.write(buff, len)`

- You're calling the **FS wrapper function** `File::write(...)`.

- This function checks which filesystem implementation created the file (`SPIFFS` in this case).

- Then it forwards the call to **SPIFFS driver's write routine**.

- SPIFFS driver takes your `buff` and writes it into flash memory according to its rules (block allocation, wear leveling, etc.).

Answer:
In `file.write(buff, len);` —

- The call itself is defined in **FS.h / FS.cpp** (the wrapper).
- But the actual writing into flash is performed by the **SPIFFS driver**, because the file was opened with `SPIFFS.open(...)`.

3. `File` object

When you call:

```
File file = SPIFFS.open(filename, FILE_WRITE);
```

- `file` is an object of type `File`.
- This `File` class is defined in **FS.h**.
- But `FS.h` is just a **generic filesy**

Who controls the speed of `file.write()`?

- **FS (the wrapper)** → does not define speed. It only forwards calls.
- **SPIFFS (the backend driver)** → is what actually controls how bytes are laid down in flash, and therefore **defines the write speed**.

🔹 What factors inside SPIFFS affect speed

1. **Flash hardware**
    - ESP32 flash is connected via QSPI (usually 40–80 MHz clock).
    - Writing to flash requires **erase/program cycles** that take microseconds to milliseconds.
2. **SPIFFS design**
    - SPIFFS is optimized for **small metadata + wear leveling**, not raw speed.
    - Every `write()` may trigger extra internal operations:
        - block allocation
        - page buffering
        - wear leveling bookkeeping
    - That overhead is what limits SPIFFS write speeds to around **100–200 KB/s** in practice.
3. **Buffer size**
    - Larger buffers → fewer SPIFFS operations per KB written → better throughput.
    - But after a point, flash latency dominates, so speed plateaus.

SPIFFS Bottlenecks

1. **SPIFFS design bottleneck**
    - It does extra work (metadata updates, page buffering, garbage collection) on every `write()`.
    - In practice, you only get **100–200 KB/s sustained writes** on ESP32 with SPIFFS.
2. **Flash hardware latency**
    - Even if you buffer 32 KB or 64 KB in RAM, writing to flash takes **milliseconds per block**.
    - That latency caps throughput no matter how fast your Wi-Fi pulls data.
3. **You're already seeing the plateau**
    - With 1 KB buffer → speed was ~tens of KB/s.
    - With 16–32 KB buffer → you reached ~160 KB/s.
    - Increasing the buffer further doesn't help much, because **SPIFFS itself is the limiter**.

Why 400 KB/s is realistic elsewhere

- **ESP32 Wi-Fi**: can do multiple **Mbps** (no issue there).
- **RAM operations**: copying 400 KB/s in RAM is trivial.
- **Flash writes via SPIFFS**: that's the choke point.

BUFFER —--> INVERSELY PROPORTIONAL —--> SPIFFS.WRITE()

OperationsHigh Buffer = Less write operations
Less Buffer = More write operations

Memory Types

[中文]

ESP32 chip has multiple memory types and flexible memory mapping features. This section describes how ESP-IDF uses these features by default.

ESP-IDF distinguishes between instruction memory bus (IRAM, IROM, RTC FAST memory) and data memory bus (DRAM, DROM). Instruction memory is executable, and can only be read or written via 4-byte aligned words. Data memory is not executable and can be accessed via individual byte operations. For more information about the different memory buses consult the *ESP32 Technical Reference Manual* > *System and Memory* [PDF].

DRAM (Data RAM)

Non-constant static data (.data) and zero-initialized data (.bss) is placed by the linker into Internal SRAM as data memory. The remaining space in this region is used for the runtime heap.

By applying the `EXT_RAM_BSS_ATTR` macro, zero-initialized data can also be placed into external RAM. To use this macro, the CONFIG_SPIRAM_ALLOW_BSS_SEG_EXTERNAL_MEMORY needs to be enabled. See Allow .bss Segment to Be Placed in External Memory.

The available size of the internal DRAM region is reduced by 64 KB (by shifting start address to `0x3FFC0000`) if Bluetooth stack is used. Length of this region is also reduced by 16 KB or 32 KB if trace memory is used. Due to some memory fragmentation issues caused by ROM, it is also not possible to use all available DRAM for static allocations - however the remaining DRAM is still available as heap at runtime.

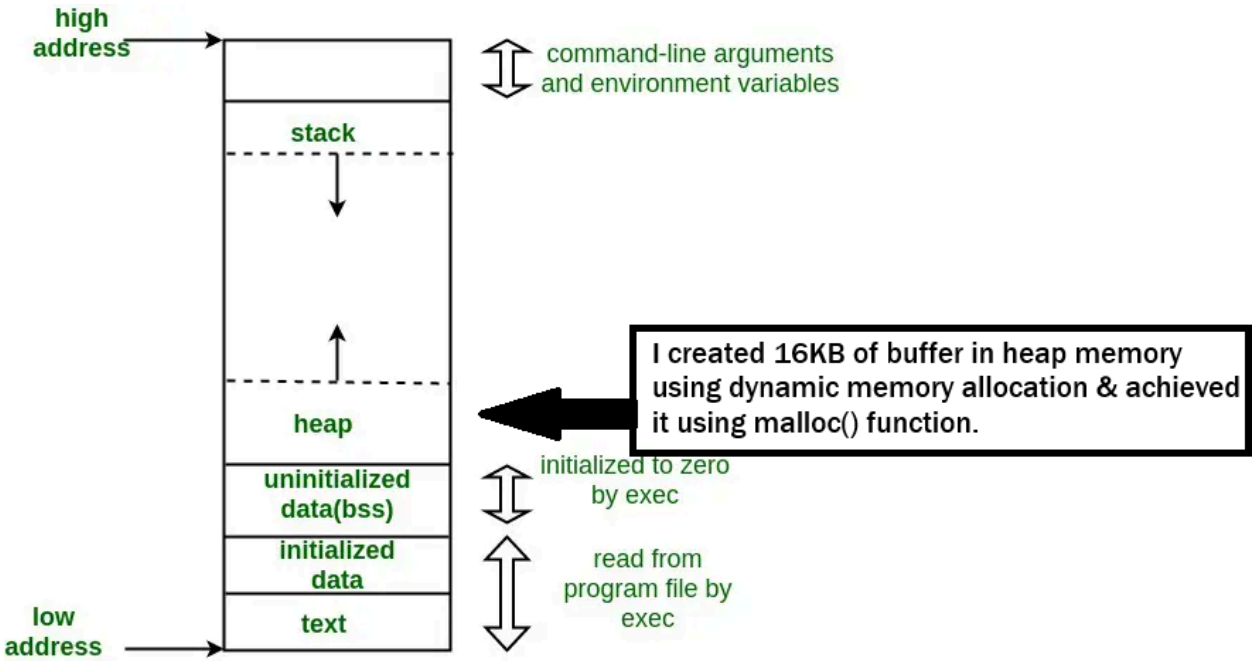
Note

There is 520 KB of available SRAM (320 KB of DRAM and 200 KB of IRAM) on the ESP32. However, due to a technical limitation, the maximum statically allocated DRAM usage is 160 KB. The remaining 160 KB (for a total of 320 KB of DRAM) can only be allocated at runtime as heap.

The ESP32 DevKit V1 has an ESP32-WROOM-32 module with a dual-core Xtensa LX6 microprocessor and **roughly 520 KB of internal SRAM (Static Random Access Memory) divided into DRAM and IRAM**, though a technical limitation restricts the maximum DRAM usage to about 160 KB. It can also support external PSRAM (Pseudo Static RAM), with some variants including this additional memory.

RAM Details

- **Internal SRAM:** The ESP32 microcontroller on the DevKit V1 includes approximately 520 KB of internal SRAM, which is used for DRAM and IRAM.
- **DRAM vs. IRAM:** The 520 KB of SRAM is split into Data RAM (DRAM) and Instruction RAM (IRAM).
- **DRAM Limitation:** Due to a technical limitation, the maximum amount of DRAM that can be used for static allocation is about 160 KB.
- **External PSRAM (Optional):** Some ESP32 DevKit V1 boards can optionally support external PSRAM, which is a type of external RAM that can expand the available memory.



Buffer allocation failed due to high buffer size

```
#include "FS.h"
#include "SPIFFS.h"
#include <WiFi.h>
#include <HTTPClient.h>
#include <WiFiClientSecure.h>

const char* ssid = "wisdom and wine";
const char* password = "permissiongr@nt
const char* filename = "/one.txt";
const char* url = "https://raw.githubus
#define BUFFER_SIZE 131072 //128KB

void setup() {
  Serial.begin(115200);

  //WiFi connection//
  WiFi.begin(ssid, password);
  Serial.print("Connecting");
  while(WiFi.status() != WL_CONNECTED){
    delay(500);
     Serial.print("..");
    }
   Serial.print(" \n Connected ");
```