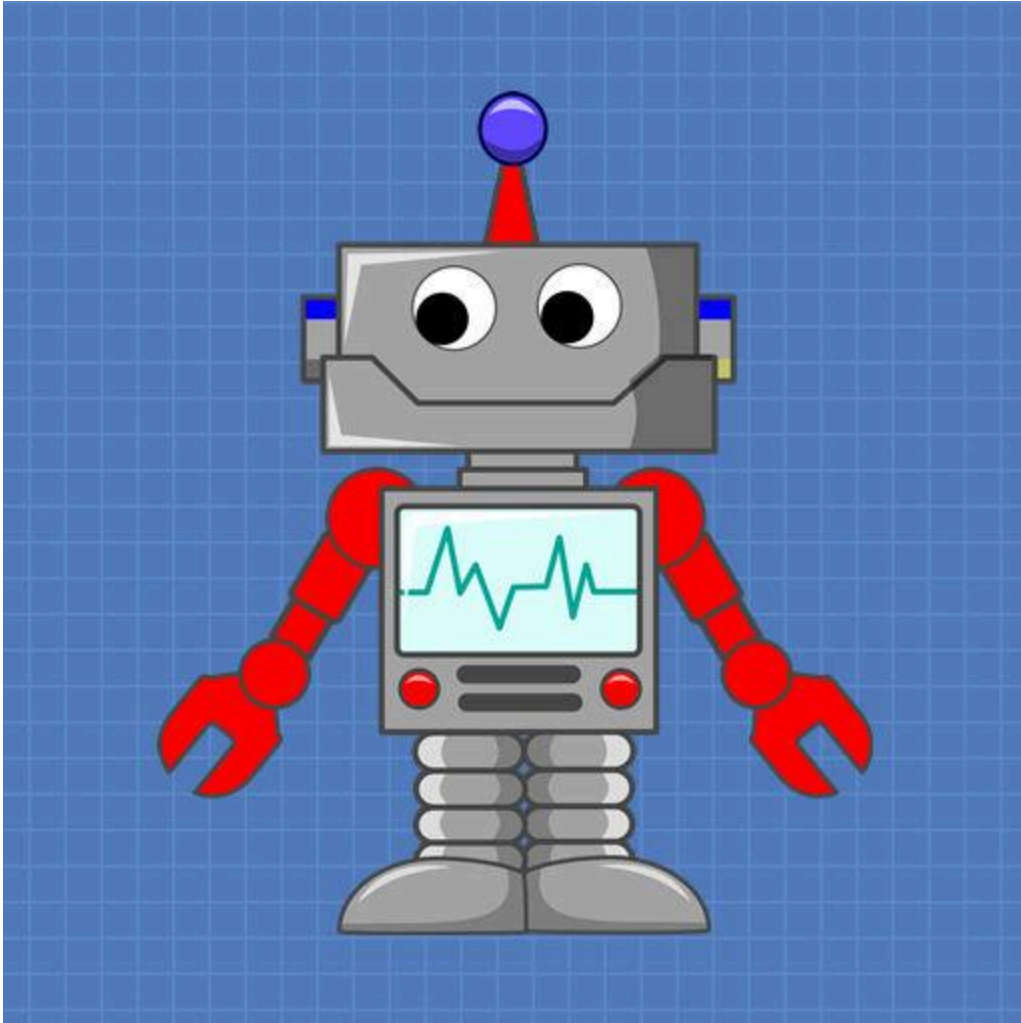


ESP32 Non-Volatile Storage

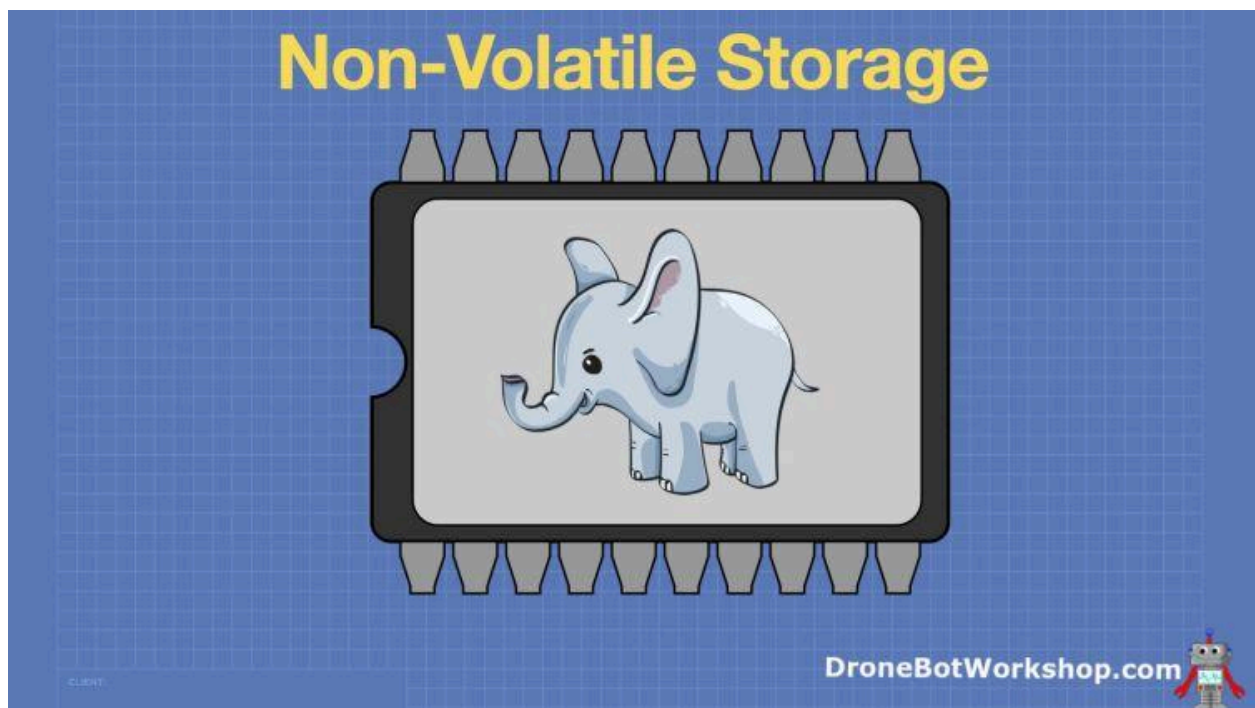


DroneBot Workshop Tutorial

<https://dronebotworkshop.com>

Introduction

When you're working on ESP32 projects, you'll often have a requirement to store data that persists even when the power goes out. Whether it's WiFi credentials, sensor calibration values, user preferences, or logged data from your latest environmental monitoring project, non-volatile storage is essential for creating robust applications that retain their data even when power is removed.



The ESP32 offers many options for persistent data storage, ranging from its built-in flash memory to external storage solutions that can dramatically expand your project's capabilities. In this article and its associated video, we'll explore four approaches to non-volatile storage:

- ESP32's built-in NVS (Non-Volatile Storage) Flash memory.
- EEPROM memory.
- SPI Flash modules.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

- FRAM (Ferroelectric RAM) technology.

Each storage technology brings unique advantages and trade-offs. By learning more about these memory technologies, you'll be able to select the correct NV storage option for your ESP32 design.

BTW, you might be wondering why microSD cards are not being included, as they definitely qualify as “non-volatile storage”. Today, we are just focusing on non-removable storage. Plus, we have covered microSD cards many times before!

Non-Volatile Storage

Non-volatile storage is any memory that retains its data even when power is removed from the system. Unlike RAM, which loses all its contents when power is lost, non-volatile storage keeps its data for years, decades, or even centuries.

NV Storage has many uses:

- Save Wi-Fi credentials.
- Store user preferences.
- Log sensor data over long periods.
- Remember the device's "last state".
- Store webpage files.

Not all NV storage options are equal, and compared to RAM, NV devices have different trade-offs. Some NV Memory devices have ample storage capacity but wear out if erased too frequently, while others are small but can be written to practically forever. Some are very fast, while others take a few milliseconds to commit data.

You need to be able to match the NV memory to your application; hopefully, this article and its associated video will provide some assistance in this regard.

Built-in Flash – NVS and LittleFS

The easiest NV storage solution is one that is already built into your ESP32. All ESP32 devices have built-in Flash Memory, the amount ranges from 4MB to 16MB, and depends on the model of ESP32. This internal flash is partitioned into several areas: one for your program code, another for system functions, and dedicated partitions for user data storage.

The built-in Flash is very useful, but the technology has its limitations. Write operations are slower than reads, and the memory has limited write endurance (typically 10,000 to 100,000 write cycles per memory cell). While wear leveling helps distribute wear evenly, applications that frequently update stored data should be designed carefully to avoid excessive write operations.

But for data that doesn't change often, like configuration data or static web pages and small images, the internal Flash is a great choice.

There are two methods of storing data in the internal Flash:

- The **NVS** (Non-Volatile Storage) system.
- The **LittleFS** file system.

The correct one to use depends on the type of data you want to store.

NVS (Parameters)

When it comes to storing small amounts of data, such as parameters or settings, Non-Volatile Storage (NVS) is the ideal choice. NVS offers a user-friendly key-value storage system similar to a dictionary or hash table. You assign a descriptive name, known as a key, and associate it with a value. NVS takes care of the complex management of flash memory in the background. It automatically implements wear leveling, which distributes write operations across different areas of the flash memory. This process helps prevent any single memory cell from wearing out prematurely.

You access NVS in the Arduino IDE using the Preferences library (*Preferences.h*). You can save and retrieve all the standard data types—integers, floats, strings, and even raw binary data.

NVS is the perfect tool for storing configuration data, API keys, device state, or any small piece of information that needs to survive a reboot but doesn't change frequently.

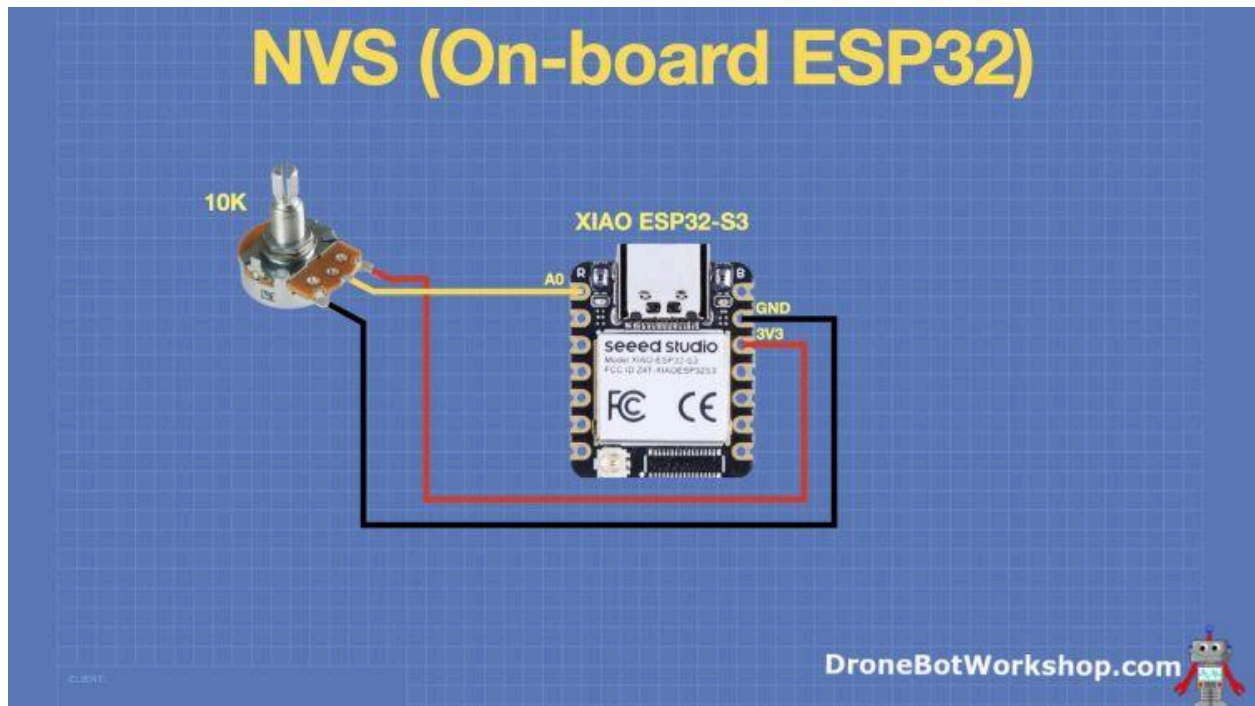
LittleFS

LittleFS provides a more sophisticated storage solution when you need file-based operations rather than simple key-value storage. Unlike NVS's key-value approach, LittleFS storage is more like a traditional file system with files and directories. You can create, read, write, and delete files using familiar operations.

LittleFS is ideal for storing configuration files, web page content, data logs, and any content that uses files.

Built-in Flash Experiments

We can perform a couple of quick experiments to see how the ESP32's built-in Flash memory is used. All of our experiments will use the same hookup:



I'm using a Seeedduino XIAO ESP32-S3 board for these experiments, but any ESP32 board will work. If you use a different board, you may need to change the pin numbers for the potentiometer; otherwise, all the code should work fine.

Preferences (NVS) Code

The first code example that we will look at uses the Preferences library to store name-value pairs. In the case of our example, we are just storing the value of the potentiometer connected to the ESP32's analog input. We'll also count the number of times the ESP32 has been rebooted, which demonstrates the ability to retain data even after reboot or power-down.

```
/*  
  ESP32 Built-in Flash memory demo - Preferences  
  esp32-preferences-demo.ino  
  Demonstrates use of Preferences library with onboard Flash Memory  
  Uses Seeeduino XIAO ESP32-S3  
  
  DroneBot Workshop 2025  
  https://dronebotworkshop.com  
*/  
  
// Include Preferences Library  
#include <Preferences.h>  
  
Preferences prefs;  
  
// Define Potentiometer pin  
const int POT_PIN = A0;  
  
void setup() {  
  // Start Serial Monitor  
  Serial.begin(115200);  
  
  // Start Preferences  
  prefs.begin("nv-demo", false);  
  
  // Persisted values  
  uint32_t boots = prefs.getUInt("boots", 0);  
  uint16_t lastPot = prefs.getUShort("lastPot", 0);  
}
```

```
// Increment boots value & store in memory

boots++;

prefs.putUInt("boots", boots);


Serial.println("\n=== NVS (Preferences) Demo ===");

Serial.printf("Boots so far: %lu\n", (unsigned long)boots);

Serial.printf("Last saved pot: %u\n", lastPot);


// Add delay for demo

delay(2000);

}


void loop() {

    // Read potentiometer as a simple 0-4095 value

    uint16_t pot = analogRead(POT_PIN);


    // Reset timer

    static uint32_t t0 = 0;


    // Update & save every 1.5 seconds

    if (millis() - t0 > 1500) {

        t0 = millis();

        prefs.putUShort("lastPot", pot);

        Serial.printf("Saved pot = %u\n", pot);

    }

}
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

This sketch demonstrates how to use the ESP32's built-in flash memory through the Preferences library to store and retrieve values that persist across resets or power cycles.

This sketch demonstrates how to use the ESP32's built-in flash memory through the Preferences library to store and retrieve values that persist across resets or power cycles.

At the top, a Preferences object named *prefs* is created, and the potentiometer input pin is defined as *POT_PIN* (A0 on the Seeeduino XIAO ESP32-S3).

In *setup()*, the serial monitor is initialized for output and *prefs.begin("nv-demo", false)* opens a namespace called "*nv-demo*" in non-volatile storage.

Two persisted values are then read: *boots* (a 32-bit unsigned integer counting how many times the board has started) and *lastPot* (a 16-bit unsigned integer storing the last potentiometer reading).

The *boots* counter is incremented and written back to flash with *putUInt()*, and both values are printed to the serial monitor.

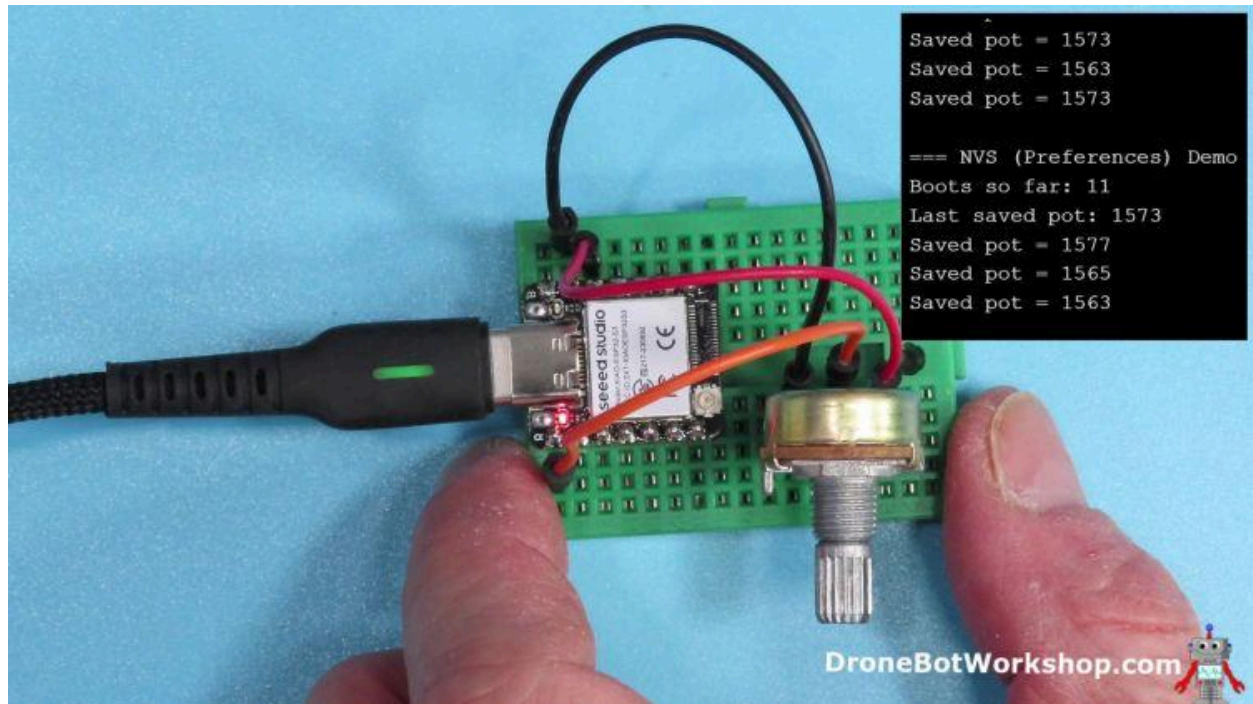
The *loop()* continuously reads the potentiometer as a raw ADC value between 0 and 4095.

Every 1.5 seconds, determined by comparing *millis()* to *t0*, the current potentiometer value is saved to non-volatile storage using *prefs.putUShort("lastPot", pot)*. This ensures that the most recent reading is always available for retrieval the next time the board starts.

Load the code to the ESP32 and watch the serial monitor. Observe the pot value. Now pull the USB-C cable from the XIAO, wait a few seconds (or longer), and plug it back in

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

while observing the serial monitor. You should note that the pot value is retained from the last session. You'll also observe that the boot count has incremented.



<https://dronebotworkshop.com>

LittleFS Code Examples

If you are storing strings of text or image files (small ones), then LittleFS may be the way to go. As long as you observe the restrictions of Flash RAM (limited number of read-write cycles), it can be a convenient method of saving at least 4MB of data (more on some ESP32 boards).

We will demonstrate the use of LittleFS using two sketches:

- One that writes into a text file, appending a line of text with the current potentiometer value.
- One that reads the text file and displays its contents on the serial monitor.

Here is the first sketch:

```
/*
  ESP32 Built-in Flash memory demo - LittleFS 1
  esp32-littlefs-save-demo.ino
  Demonstrates saving file in Built-in Flash Memory with LittleFS
  Seeeduino XIAO ESP32-S3 with Potentiometer on A0

  DroneBot Workshop 2025
  https://dronebotworkshop.com
*/

// Include Required Libraries
#include <FS.h>
#include <LittleFS.h>

// Potentiometer Pin
const int POT_PIN = A0;

// File name in LittleFS filesystem
const char* LOG_PATH = "/potlog.txt";

// Append Line Function
void appendLine(const char* path, const String& line) {
  File f = LittleFS.open(path, LittleFS.exists(path) ? FILE_APPEND : FILE_WRITE);
  if (!f) {
    Serial.println("Append failed");
    return;
  }
  f.println(line);
}
```

```
f.close();
}

void setup() {
    // Start Serial Monitor
    Serial.begin(115200);

    // Give the USB-CDC time to come up (helps on S2/S3 boards)
    uint32_t tStart = millis();
    while (!Serial && (millis() - tStart < 2000)) {}

    Serial.println("\n=== LittleFS WRITER (XIAO ESP32-S3) ===");

    // Mount filesystem if it exists
    if (!LittleFS.begin(true)) { // true = format on first run if mount fails
        Serial.println("LittleFS mount failed");
        while (1) {}
    }

    // Set Potentiometer pin mode
    pinMode(POT_PIN, INPUT);

    // If file is new/empty, write a header row
    if (!LittleFS.exists(LOG_PATH) || LittleFS.open(LOG_PATH, FILE_READ).size() == 0) {
        appendLine(LOG_PATH, "ms,pot");
    }

    // get parameters
```


For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
size_t used = LittleFS.usedBytes();

size_t total = LittleFS.totalBytes();

Serial.printf("Mounted. Used %u / %u bytes\n", (unsigned)used, (unsigned)total);

Serial.printf("Logging to %s every 1s. Open Serial Monitor to watch.\n", LOG_PATH);
}

void loop() {

    static uint32_t t0 = 0;

    // Get pot value every second

    if (millis() - t0 >= 1000) {

        t0 = millis();

        int pot = analogRead(POT_PIN);

        String line = String(millis()) + "," + String(pot);

        // Append to potlog.txt file

        appendLine(LOG_PATH, line);

        Serial.println("APPEND: " + line);

    }

}
```

At the beginning of the sketch, the potentiometer input pin is defined as *POT_PIN*, and the log file path is set in *LOG_PATH*.

The helper function *appendLine()* takes a file path and a string, opens the file in append mode if it exists (or write mode if it doesn't), writes the string as a new line, and then closes the file.

In *setup()*, the serial monitor is initialized, and a short delay allows the USB-C connection to establish on S2/S3 boards.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

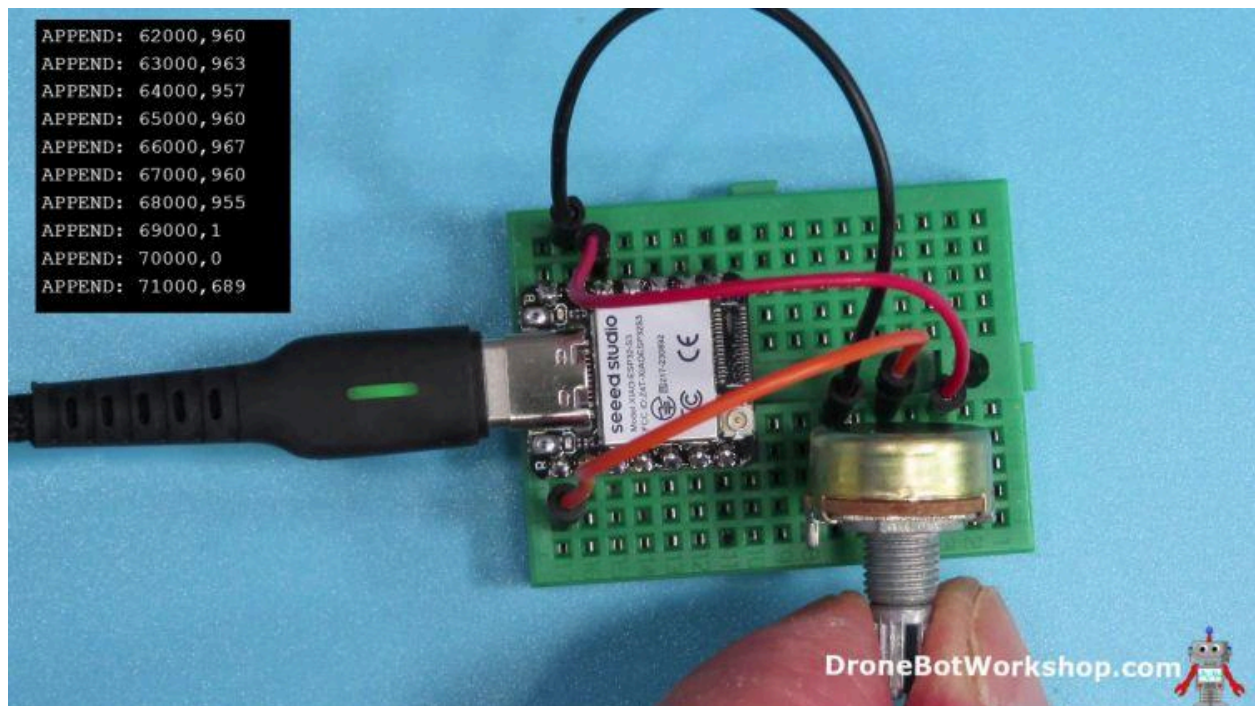
The code then mounts the LittleFS filesystem with *LittleFS.begin(true)*, where the *true* parameter tells it to format the flash if mounting fails.

The potentiometer pin is configured as an input, and if the log file doesn't exist or is empty, a CSV header row "*ms,pot*" is written to it. The sketch also queries and prints the total and used bytes in the filesystem, giving the user feedback on available storage.

In the loop() function, every 1000 ms, the code reads the potentiometer value with *analogRead(POT_PIN)*.

It then constructs a string containing the current *millis()* timestamp and the potentiometer reading, appends it to the log file using *appendLine()*, and prints the same line to the serial monitor.

Load the sketch onto the ESP32 and run it for a while, adjusting the potentiometer to give varied readings. These should be recorded into the text file, which is being saved in the onboard flash memory.



<https://dronebotworkshop.com>

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Then disconnect the sketch. If all worked well, we should have a text file in our Flash memory, waiting to be read.

The following sketch will read that text file and display its contents on the serial monitor.

```
/*  
  ESP32 Built-in Flash memory demo - LittleFS 2  
  esp32-littlefs-read-demo.ino  
  Demonstrates reading file stored in Built-in Flash Memory with LittleFS  
  * Run after running esp32-littlefs-save-demo.ino  
  Seeeduino XIAO ESP32-S3 with Potentiometer on A0  
  
  DroneBot Workshop 2025  
  https://dronebotworkshop.com  
*/  
  
// Include required libraries  
#include <FS.h>  
#include <LittleFS.h>  
  
// Text file to read  
const char* LOG_PATH = "/potlog.txt";  
  
// Dump File Contents Function  
void dumpFile(const char* path) {  
  if (!LittleFS.exists(path)) {  
    Serial.printf("File not found: %s\n", path);  
    return;  
  }  
  File f = LittleFS.open(path, FILE_READ);  
  if (!f) {  
    Serial.println("Open failed");  
    return;  
  }  
}
```

```
    }

    Serial.printf("--- %s (size: %d bytes) ---\n", path, (int)f.size());

    // Stream out the whole file
    while (f.available()) {
        Serial.write(f.read());
    }

    Serial.println("\n-----");

    f.close();
}

// List Root Directory Function
void listRoot() {

    File root = LittleFS.open("/");

    if (!root || !root.isDirectory()) {
        Serial.println("Root open failed");
        return;
    }

    Serial.println("Listing /");

    File file = root.openNextFile();

    if (!file) Serial.println("(empty)");

    while (file) {
        Serial.printf("  %s (%d bytes)\n", file.name(), (int)file.size());

        file = root.openNextFile();
    }
}

void setup() {
```

```
// Start Serial Monitor

Serial.begin(115200);

// Give the USB-CDC time to come up (helps on S2/S3 boards)

uint32_t tStart = millis();

while (!Serial && (millis() - tStart < 2000)) {}

Serial.println("\n=== LittleFS READER (XIAO ESP32-S3) ===");

// IMPORTANT: don't auto-format here-if mount fails, we want to know.

if (!LittleFS.begin(false)) { // false = do NOT format on failure

    Serial.println("LittleFS mount failed (did you run the Save Demo first?).");

    while (1) {}

}

// Get Parameters

size_t used = LittleFS.usedBytes();

size_t total = LittleFS.totalBytes();

Serial.printf("Mounted. Used %u / %u bytes\n\n", (unsigned)used, (unsigned)total);

// Show Directory

listRoot();

Serial.println();

// Print file contents

dumpFile(LOG_PATH);
```

```
Serial.println("\nDone. Press EN/RESET to reprint.");  
}  
  
void loop() {  
    // nothing-contents are printed once at boot  
}
```

This sketch reads and displays the contents of the *potlog.txt* text file stored in the ESP32-S3's built-in flash memory using the LittleFS filesystem. Remember that you need to run the previous sketch and create the file before you run this one!

The constant *LOG_PATH* holds the file's path. Two helper functions are defined:

- *dumpFile()* checks if the specified file exists, opens it in read mode, prints its size, and then streams its entire contents to the serial monitor.
- *listRoot()* opens the root directory of the LittleFS filesystem and lists all files along with their sizes.

In *setup()*, the serial monitor is initialized, and a short delay allows the USB-C connection to establish on S2/S3 boards.

The code then mounts the LittleFS filesystem with *LittleFS.begin(false)*, where *false* ensures it will not auto-format if mounting fails – this is important to avoid erasing existing data.

It retrieves and displays the total and used bytes in the filesystem, calls *listRoot()* to show the directory contents, and finally calls *dumpFile(LOG_PATH)* to print the contents of *potlog.txt* to the serial monitor.

The *loop()* function is empty, as everything is done in *setup()*.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

Load this sketch, overwriting the previous one. Once it loads, observe the serial monitor. You should see the contents of the text file, several lines representing the potentiometer value. The last value should match the final one you saw when running the first sketch.



```
esp32-littlefs-read-demo.ino
77
78 // Show Directory

Serial Monitor x Output
Message (Enter to send message to 'XIAO_ESP32S3' on 'COM16') New Line 115200 baud

Mounted. Used 12288 / 1572864 bytes

Listing /
  potlog.txt  (1743 bytes)

--- /potlog.txt (size: 1743 bytes) ---
ms,pot
1000,1565
2000,1566
3000,1567
4000,1573
5000,1569
6000,1567
7000,1566
8000,1562

DroneBotWorkshop.com
Ln 67, Col 27 XIAO_ESP32S3 on COM16
```

This sketch, and the one before it, illustrate how simple it is to use the LittleFS filesystem.

If you download the ZIP file containing all the code from this article, you'll also receive a third sketch, one that erases the text file. You can use it to “clean” your ESP32 between experiments.

<https://dronebotworkshop.com>

EEPROM

EEPROM stands for **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory. While its name includes “Read-Only,” it can actually be rewritten many times—typically up to a million times per memory address. EEPROMs are “byte-addressable”, meaning you can read or write a single byte at a time without disturbing the rest of the memory.

EEPROMs are the oldest technology we will be looking at today, and although they have been around for about half a century, they are still widely used. Modern EEPROMs use the I²C bus, making them very easy to use.

One notable feature that sets EEPROM apart from newer storage technologies is its write cycle time. After writing data to EEPROM, you need to wait approximately 5 to 10 milliseconds for the write operation to finish. This delay can accumulate when writing large amounts of data; therefore, EEPROMs are best suited for situations where data is read more frequently than it is written.

EEPROM Experiments

We will be using a very popular EEPROM to run a simple experiment.

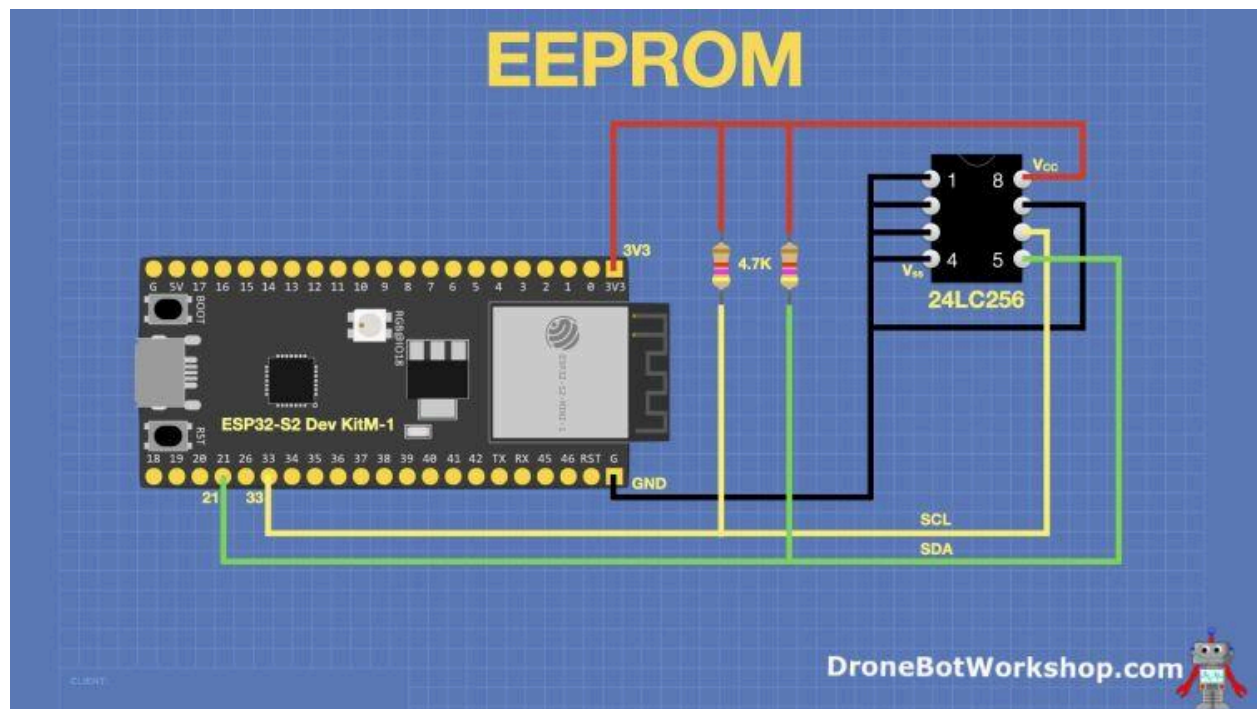
The [24LC256 from Microchip](#) offers 256 kilobits (32 kilobytes) of storage in a convenient 8-pin package. This capacity provides ample space for storing configuration data, user preferences, calibration values, and moderate amounts of logged information.

The device communicates using the standard I²C protocol, operating at speeds up to 400 kHz. Up to eight devices can share the same I²C bus by setting different combinations on the A0, A1, and A2 pins.

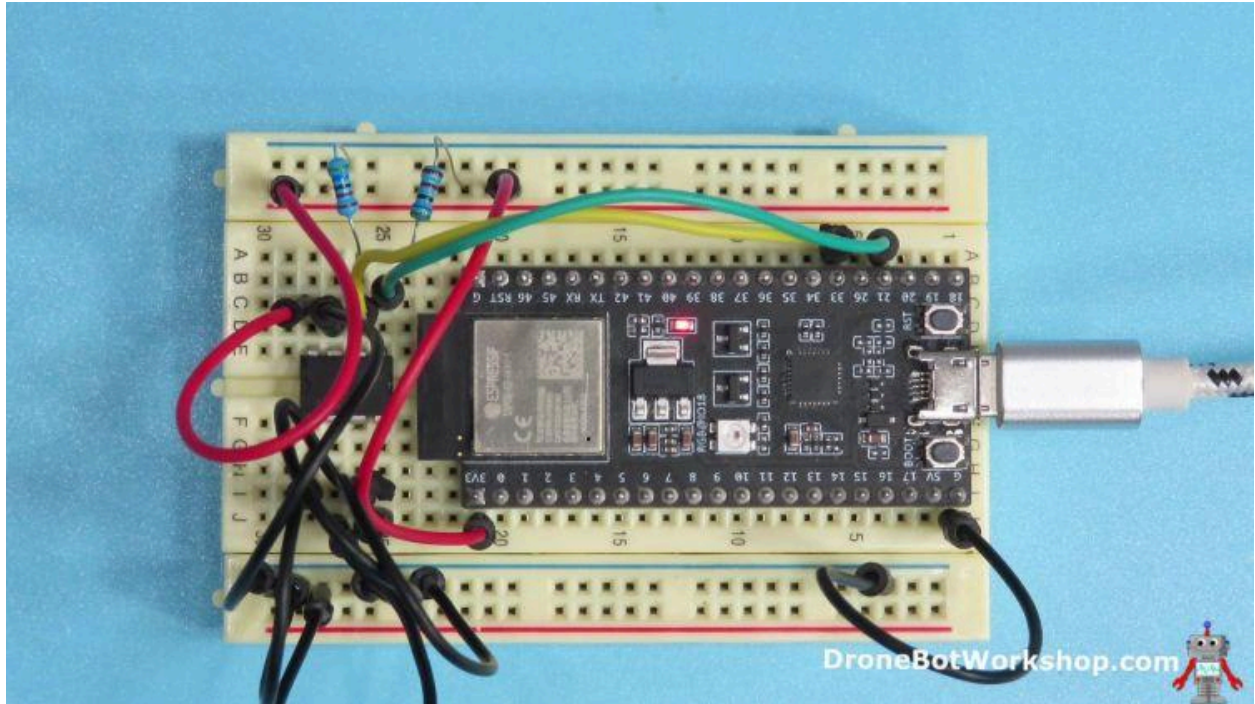
EEPROM Hookup

As the 24LC256 is an I²C device, the hookup is pretty simple. I'm using an ESP32-S2 DevKit, an older board, but any ESP32 board will work. If your board doesn't have the same pins as I used, use two alternative ones; most ESP32 boards allow you to use almost any pin for I²C. Be sure to update the code to reflect your pin changes!

Here is the hookup diagram:



Note the use of pull-up resistors. I used 4.7k, but the value isn't critical. Also note the grounding of all address pins, to give the module an I²C address of 0x50.



EEPROM Code

Although there are libraries created for EEPROMs, we can simply use the Wire library for I²C to communicate with our memory chip. This is illustrated in the following sketch, which demonstrates the use of the 24LC256 as follows:

- Creates 32 one-byte values (sequential numbers.
- Stores them in EEPROM.
- Reads them from EEPROM.
- Displays them on the serial monitor.

Here is the code to accomplish this:

```
/*
  ESP32 External EEPROM Demo
  esp32-eeeprom-demo.ino
  Demonstrates use of 24LC256 EEPROM with ESP32
  Uses ESP32-S2 Dev Kit

  DroneBot Workshop 2025
  https://dronebotworkshop.com
*/

// Include Wire Library for I2C
#include <Wire.h>

// Define constants
const uint8_t EEPROM_ADDR = 0x50;
const uint8_t SDA_PIN = 21;
const uint8_t SCL_PIN = 33;

// EEPROM Write Page Function
void eepromWritePage(uint16_t addr, const uint8_t* data, size_t len) {
  Wire.beginTransaction(EEPROM_ADDR);
  Wire.write((addr >> 8) & 0xFF);
  Wire.write(addr & 0xFF);
  for (size_t i = 0; i < len; i++) Wire.write(data[i]);
  Wire.endTransmission();

  // ACK polling until write cycle completes (~5 ms)
  while (true) {
```

```
Wire.beginTransaction(EEPROM_ADDR);

uint8_t err = Wire.endTransmission();

if (err == 0) break;

delay(1);

}

}

// EEPROM Read Block Function
void eepromReadBlock(uint16_t addr, uint8_t* data, size_t len) {

Wire.beginTransaction(EEPROM_ADDR);

Wire.write((addr >> 8) & 0xFF);

Wire.write(addr & 0xFF);

Wire.endTransmission(false);

Wire.requestFrom(EEPROM_ADDR, (uint8_t)len);

for (size_t i = 0; i < len && Wire.available(); i++) {

data[i] = Wire.read();

}

}

void setup() {

// Start Serial Monitor

Serial.begin(115200);

delay(500);

// Start I2C at 400kHz

Wire.begin(SDA_PIN, SCL_PIN);

Wire.setClock(400000);
```

```
// Align to page boundary (multiples of 64)

const uint16_t base = 0x0040;


// Create 32 values to write to EEPROM

uint8_t tx[32];

for (int i = 0; i < 32; i++) tx[i] = i;


// Write to EEPROM

Serial.println("Writing 32 bytes (page write)...");

eepromWritePage(base, tx, sizeof(tx));


// Read from EEPROM

Serial.println("Reading back 32 bytes:");

uint8_t rx[32] = { 0 };

eepromReadBlock(base, rx, sizeof(rx));


// Cycle through data & print

for (int i = 0; i < 32; i++) {

    Serial.printf("%02X ", rx[i]);

    if ((i + 1) % 16 == 0) Serial.println();

}

Serial.println();

}

void loop() {}
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

This sketch starts by including the Wire library for I²C . The EEPROM's I²C address is set with `EEPROM_ADDR (0x50)`, and the SDA and SCL pins are defined as GPIO 21 and GPIO 33, respectively.

Two key helper functions handle the memory operations:

- `eeepromWritePage()` writes a block of bytes starting at a given 16-bit address, sending the high and low address bytes first, followed by the data. After sending, it performs ACK polling — repeatedly attempting a transmission until the EEPROM signals it has finished its internal write cycle (about 5 ms).
- `eeepromReadBlock()` reads a specified number of bytes from a given address by sending the address, then requesting the data from the device.

In `setup()`, the serial monitor is initialized, and the I²C bus is started at 400 kHz for faster transfers.

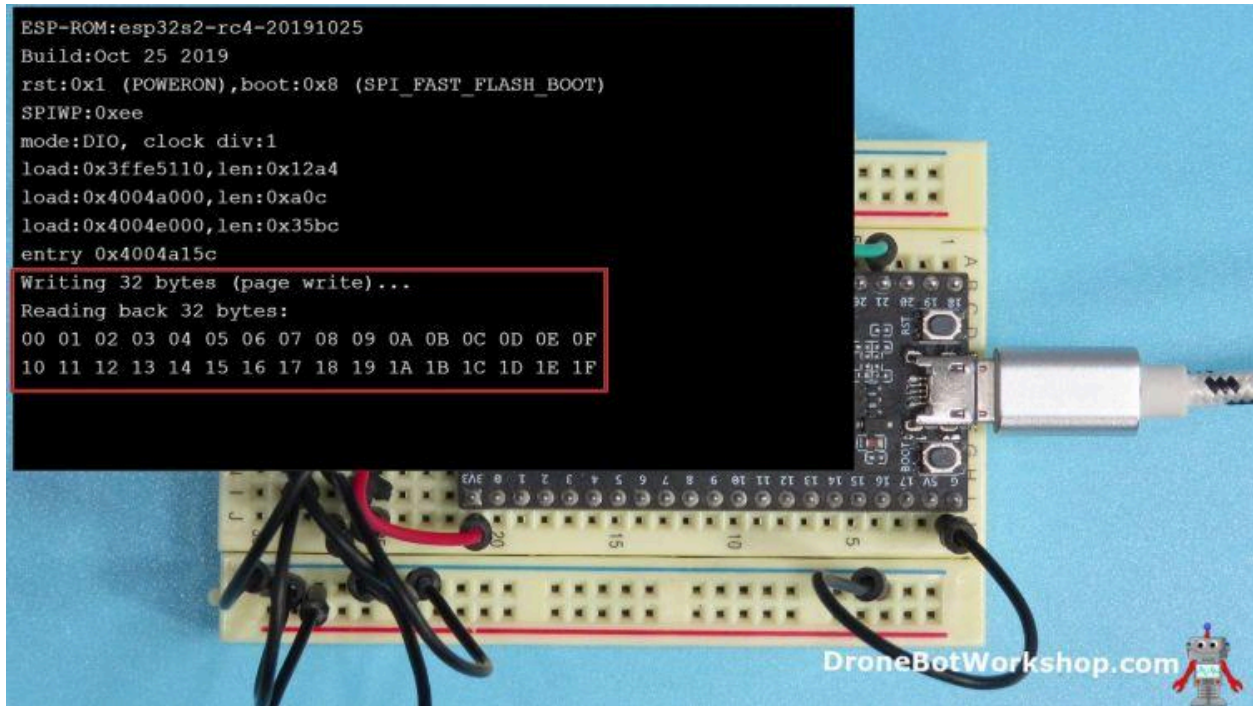
The variable `base` is set to `0x0040` to align the write to a 64-byte page boundary, which is important for efficient EEPROM writes.

An array `tx` of 32 bytes is filled with incremental values (`0x00` to `0x1F`) and written to the EEPROM using `eeepromWritePage()`. The same address range is then read back into the `rx` array with `eeepromReadBlock()`.

Finally, the contents of `rx` are printed in hexadecimal format, 16 bytes per line, allowing the user to verify that the data read matches what was written.

The `loop()` is empty, as this is a one-time demonstration run at startup.

Load the sketch up to your ESP32 board, run it, and watch the serial monitor. You should see the results printed in the serial monitor, with the 32 bytes stored displayed at the end of the list.



Although this experiment may be brief, it remains valuable. The functions we created to read and write EEPROM data can be reused in your own code.

SPI Flash

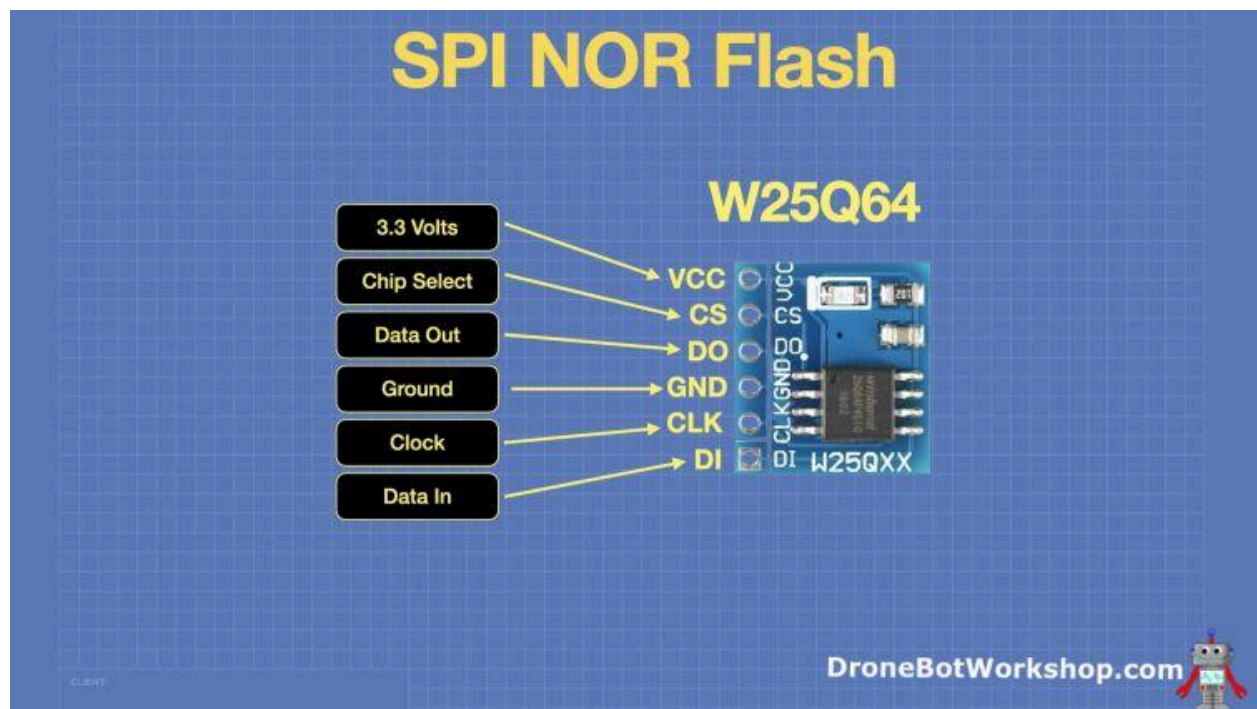
If you need more space than the ESP32's internal flash can provide, external SPI flash is a great option. These chips are similar to the ESP32's onboard flash but connect via the SPI (Serial Peripheral Interface) bus. They're tiny, cheap, and fast to read (the SPI bus is much faster than the I²C bus).

SPI Flash memory utilizes NAND or NOR flash technology to store data in arrays of floating-gate transistors. While it is similar to EEPROM, the organization and access methods differ. The main distinction between the two lies in how data is erased and written. EEPROM allows for byte-level operations, whereas SPI Flash organizes memory into pages (typically 256 bytes) and sectors (generally 4096 bytes). In SPI Flash, entire blocks must be erased before new data can be written.

SPI Flash Experiments

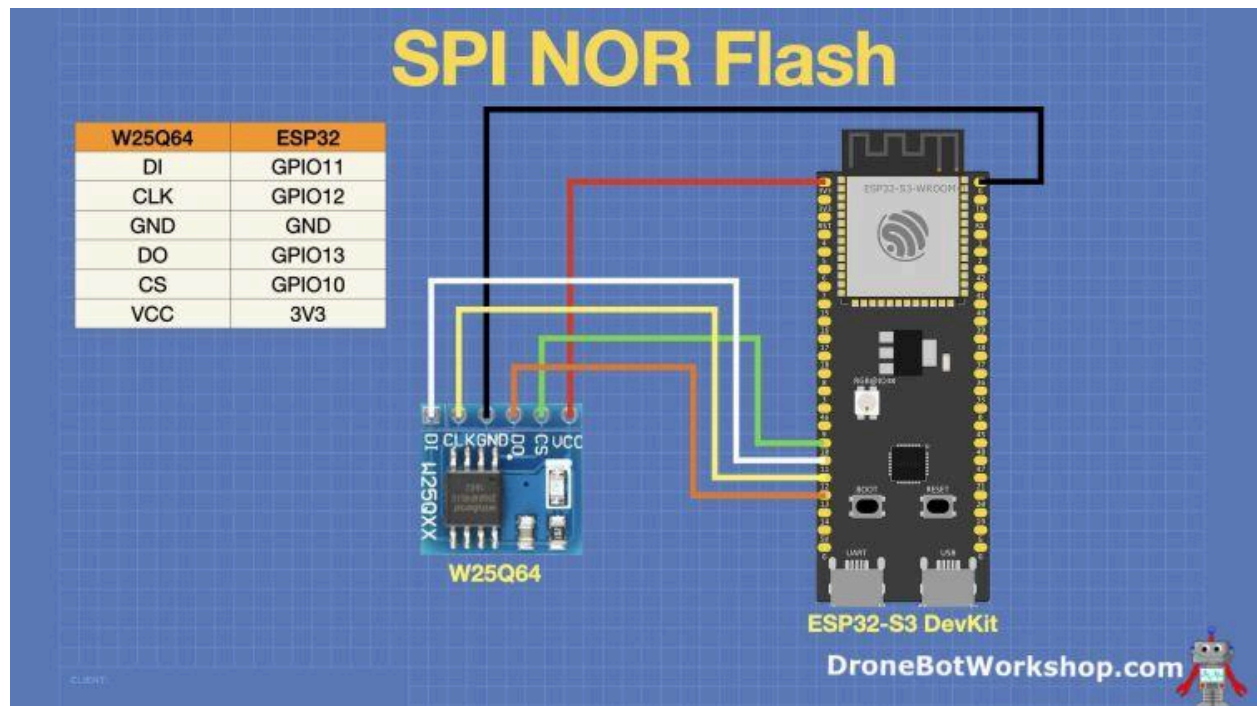
We will be using the [W25Q64 from Winbond](#). This is a popular SPI Flash device and is available on a module. It provides 64 megabits (8 megabytes) of storage. The device operates on a standard 4-wire SPI interface (clock, data in, data out, and chip select), though it also supports advanced modes like Quad-SPI for even higher performance. Operating voltage ranges from 2.7V to 3.6V, making it compatible with 3.3-volt logic.

Here are the pinouts for the W25Q64 module:



SPI Flash Hookup

Here is how I hooked up my W25Q64 module to an ESP32-S3 DevKit board. As with all of the other experiments, you could use a different ESP32. The pins used for SPI (except the SC pin) are the defaults for this ESP32 board. If you use a different board, you should determine its default SPI pins.



SPI Flash Code

Our demonstration code will perform the following:

- Read the internal ID from the SPI Flash chip.
- Determine the Flash memory capacity by reading its parameters.
- Erase a 4kB sector.
- Write and Read the Flash Memory.

Here is the code:

```
/*  
  ESP32 External SPI Flash Demo  
  esp32-ext-flash-demo.ino  
  Demonstrates use of W25Q64 Flash memory module  
  Uses ESP32-S3 DevKit1  
  Uses SPIMemory Library
```

```
  1 - Read Flash ID  
  2 - Read Flash Capacity  
  3 - Erase 4kB sector  
  4 - Write and Read
```

```
  DroneBot Workshop 2025  
  https://dronebotworkshop.com
```

```
*/
```

```
// Include Required Libraries
```

```
#include <SPI.h>
```

```
#include <SPIMemory.h>
```

```
// --- Your wiring on ESP32-S3 DevKitC-1 ---
```

```
static const int PIN_SCK = 12;    // CLK
```

```
static const int PIN_MISO = 13;   // DO  -> MISO
```

```
static const int PIN_MOSI = 11;   // DI  -> MOSI
```

```
static const int PIN_CS = 10;     // CS
```

```
SPIFlash flash(PIN_CS);
```

```
void setup() {  
  
    Serial.begin(115200);  
  
    delay(200);  
  
    Serial.println("\n=== W25Q64 Quick Test ===");  
  
  
    SPI.begin(PIN_SCK, PIN_MISO, PIN_MOSI, PIN_CS);  
  
  
    if (!flash.begin()) {  
        Serial.println("Flash NOT detected. Check 3V3, GND, CS, and wiring.");  
        while (1) delay(10);  
    }  
  
  
    // NEW: getJEDECID() returns a 32-bit value like 0xEF4017 for W25Q64  
  
    uint32_t jedec = flash.getJEDECID();  
    uint8_t manufacturer = (jedec >> 16) & 0xFF;  
    uint8_t memType = (jedec >> 8) & 0xFF;  
    uint8_t capacityCode = jedec & 0xFF;  
  
  
    Serial.printf("JEDEC ID: 0x%06lX  (MFG=0x%02X TYPE=0x%02X CAP=0x%02X)\n",  
                  (unsigned long)jedec, manufacturer, memType, capacityCode);  
    Serial.printf("Reported capacity: %lu bytes\n", (unsigned long)flash.getCapacity());  
  
  
    const uint32_t addr = 0x00000; // sector 0  
    Serial.println("Erasing 4KB sector @ 0x000000...");  
    if (!flash.eraseSector(addr)) Serial.println("Erase FAILED");  
  
  
    // NEW: make it non-const for writeCharArray (or use writeAnything)  
  
    char msg[] = "Hello from ESP32-S3 + W25Q64!";  
}
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
bool okW = flash.writeCharArray(addr, msg, sizeof(msg)); // includes '\0'

char buf[sizeof(msg)] = { 0 };

bool okR = flash.readCharArray(addr, buf, sizeof(buf));

Serial.printf("Write: %s Read: %s\n", okW ? "OK" : "FAIL", okR ? "OK" : "FAIL");

Serial.print("Data: ");

Serial.println(buf);

}

void loop() {}
```

In this sketch, the SPI pins for clock (PIN_SCK), data in (PIN_MOSI), data out (PIN_MISO), and chip select (PIN_CS) are explicitly defined to match the pinout of the DevKit I used. You may need to change them to suit your ESP32 board.

An *SPIFlash* object named *flash* is created with the chip select pin, which can be any I/O pin on the ESP32.

In `setup()`, the serial monitor is initialized, and the SPI bus is started with the defined pins.

The *flash.begin()* call attempts to detect the connected W25Q64; if it fails, the program halts with an error message. Once detected, the code retrieves the JEDEC ID via `getJEDECID()`, which encodes the manufacturer, memory type, and capacity code, and prints these values along with the reported total capacity from *getCapacity()*.

It then targets address 0x00000 (sector 0) and erases a 4 KB sector using *eraseSector()* to ensure a clean write area.

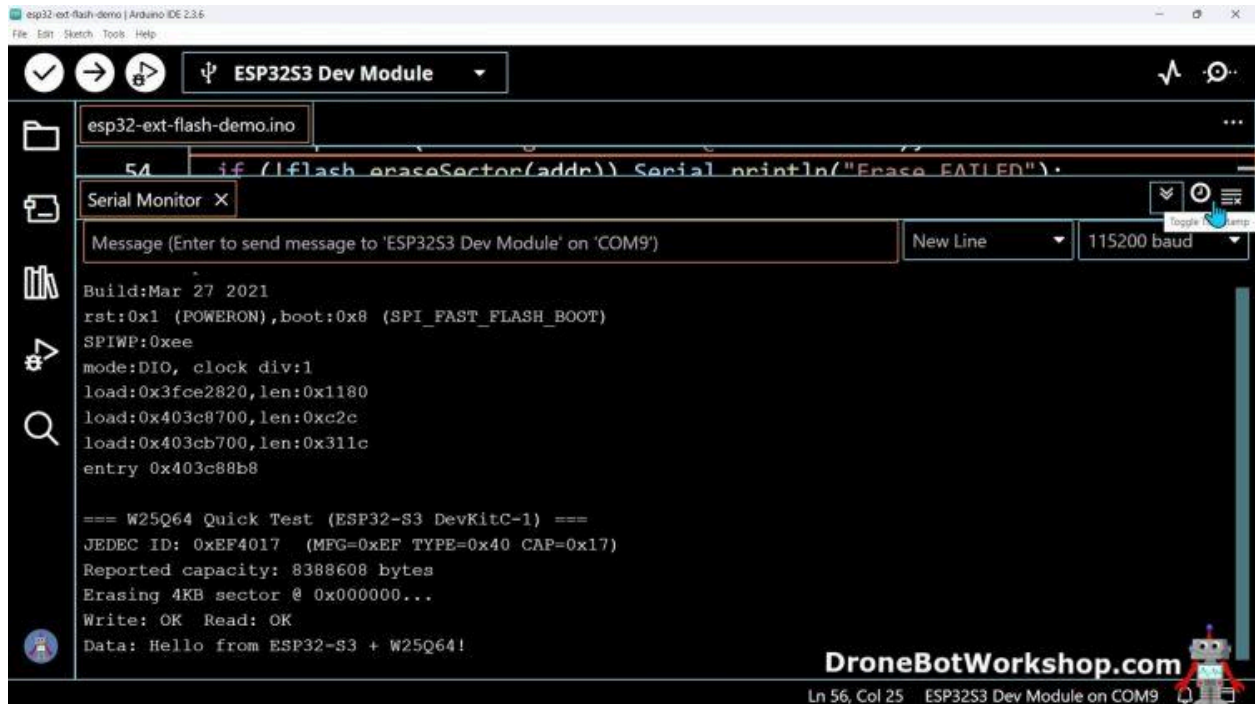
A test message string is stored in the variable *msg* and written to flash at the chosen address using *writeCharArray()*. A buffer *buf* of the same size is then filled by reading

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

back the data with `readCharArray()`. The code reports whether the write and read operations succeeded and prints the retrieved string to the serial monitor.

Once again, the `loop()` function is empty, as this is a one-time demonstration.

Load the code onto the ESP32 and observe the results on the serial monitor. An example from my serial monitor is shown below.



```
esp32-ext-flash-demo | Arduino IDE 2.3.6
File Edit Sketch Tools Help
ESP32S3 Dev Module
esp32-ext-flash-demo.ino
54 if (!flash_eraseSector(addr)) Serial.println("Erase FAILED");

Serial Monitor x
Message (Enter to send message to 'ESP32S3 Dev Module' on 'COM9') New Line 115200 baud

Build:Mar 27 2021
rst:0x1 (POWERON),boot:0x8 (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:1
load:0x3fce2820,len:0x1180
load:0x403c8700,len:0xc2c
load:0x403cb700,len:0x311c
entry 0x403c88b8

=== W25Q64 Quick Test (ESP32-S3 DevKitC-1) ===
JEDEC ID: 0xEF4017 (MFG=0xEF TYPE=0x40 CAP=0x17)
Reported capacity: 8388608 bytes
Erasing 4KB sector @ 0x000000...
Write: OK Read: OK
Data: Hello from ESP32-S3 + W25Q64!

DroneBotWorkshop.com
Ln 56, Col 25 ESP32S3 Dev Module on COM9
```

Note that the capacity is 8388608 bytes, which is correct for an 8 MB device. It also passed the read and write tests.

Once again, this is a simple experiment, but the elements used in the code can be repurposed in your own code.

FRAM

FRAM (Ferroelectric RAM) is a modern technology that merges the speed of Static RAM (SRAM) with the non-volatile characteristics of Flash or EEPROM. It saves data by polarizing a ferroelectric layer, which can be switched billions or even trillions of times without suffering wear. Unlike traditional storage technologies that have slow write operations and limited durability, FRAM offers instantaneous writes with virtually unlimited write cycles.

Write operations in FRAM occur at speeds comparable to RAM, typically completing in nanoseconds instead of the milliseconds required by EEPROM or flash memory. This rapid performance enables applications that were previously not feasible, such as high-frequency data logging, real-time buffering of critical data, and immediate storage of system state information.

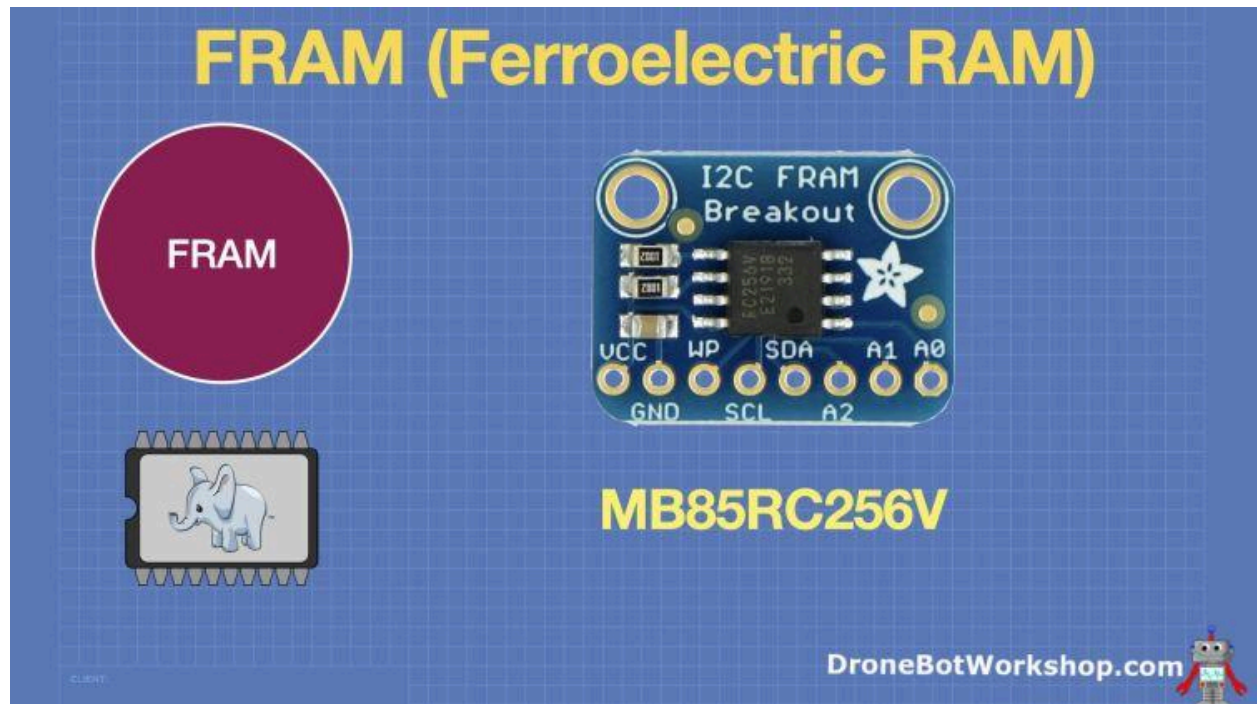
If you need to store data that changes rapidly, or if you need to play back stored data at a quick pace, then FRAM is the memory for you! It does have a few disadvantages, however. It is more expensive than the other storage options we examined today (although the module we are using is very cost-effective). It has a higher density, meaning that you can't pack as much of it onto a chip as with other memory technologies like Flash or SRAM.

FRAM Experiments

The [MB85RC256V from Fujitsu](#) is an ideal FRAM for ESP32 projects. This device provides 256 kilobits (32 kilobytes) of FRAM storage with an I²C interface.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

The device operates from 1.8V to 5.5V, so it can be used with both 3.3V and 5V logic. It is available on a popular breakout module, with the following pinout:

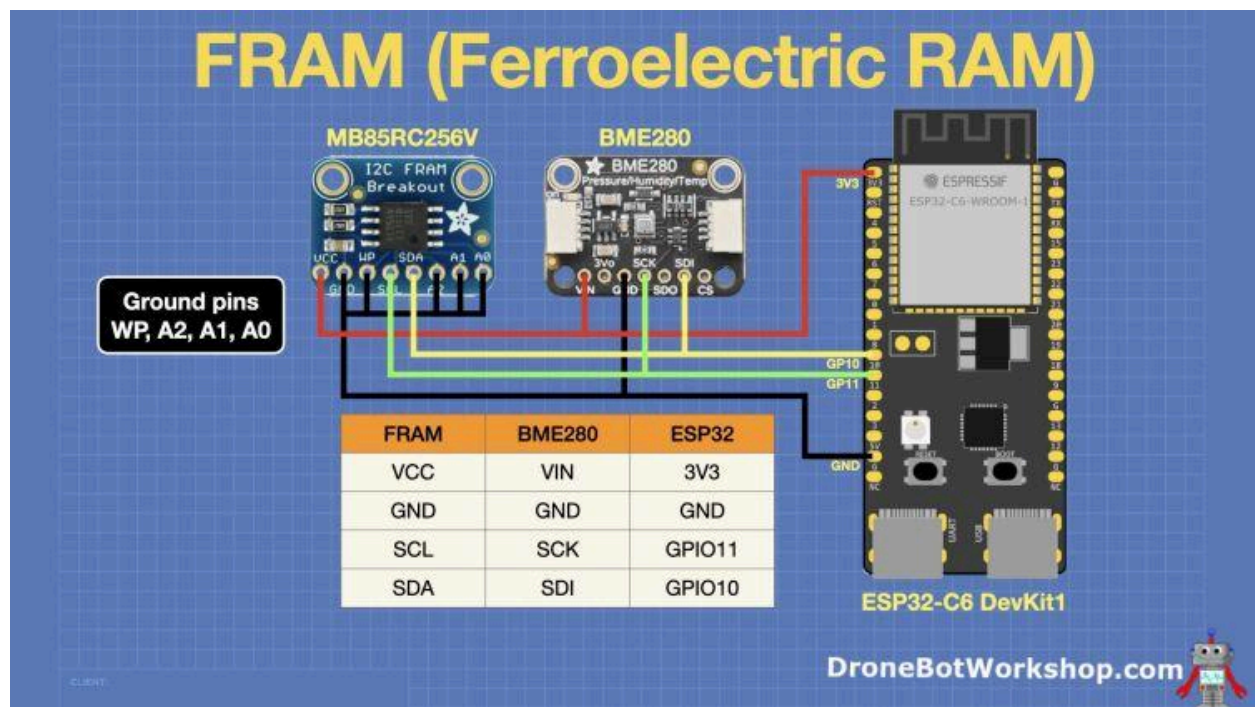


As with the EEPROM we used earlier, the module has a series of address pins that can be used to configure its I²C address.

FRAM Hookup

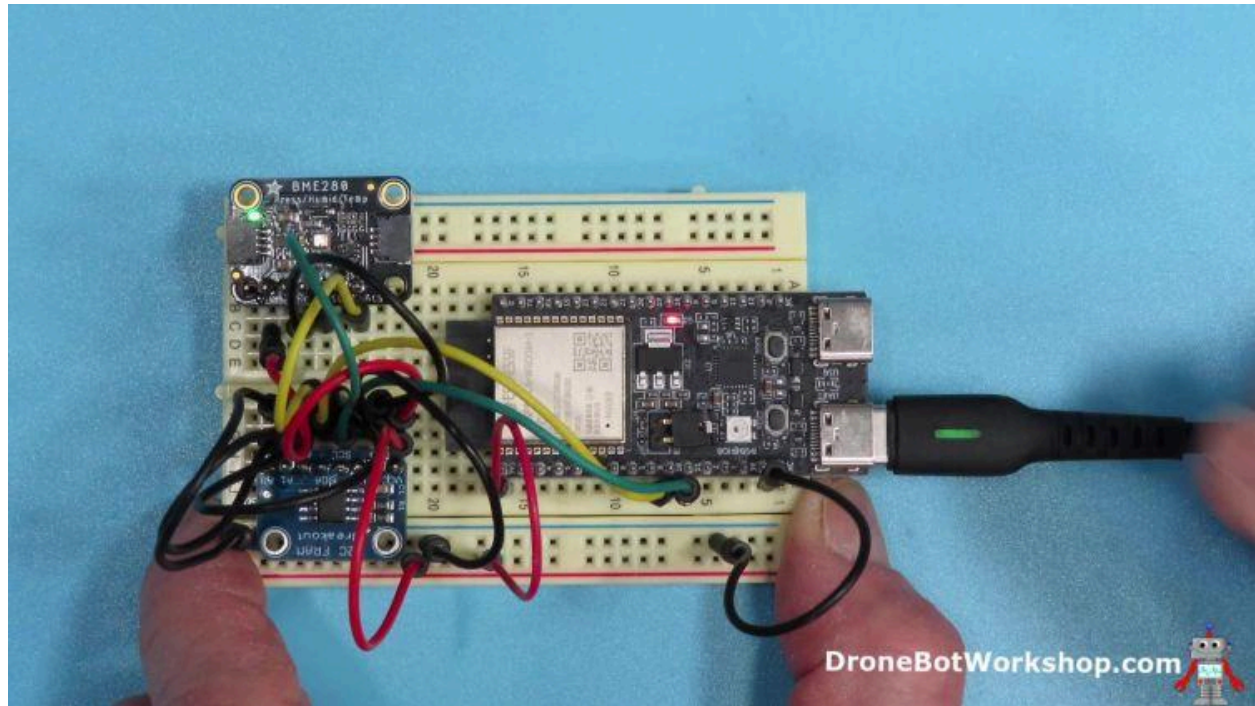
To demonstrate the operation of the FRAM, we'll be using it with an ESP32 and a BME280. The BME280 is a Pressure/humidity & Temperature sensor that can operate on either the I²C or SPI bus. We will use it on the same I²C bus as the MB85RC256V FRAM module.

The ESP32 I chose is an ESP32-C6 DevKit. Once again, any ESP32 module will work for this experiment.



Try to keep the wires short. Pull-up resistors are not required, as the modules both have them.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



Here it is, wired onto a solderless breadboard. Now, let's look at some code we can run to test it out.

<https://dronebotworkshop.com>

FRAM Code

Our code will take temperature readings from the BME280 every 200 milliseconds and record them to the FRAM module. This is a rapid pace of data that would not be practical with the other memory types we have looked at here:

- Flash (internal or external) would burn out in about 60 hours, as it's only good for about a million cycles.
- EEPROM would be too slow to record data at this pace. It would also burn out after a few weeks.

FRAM is very fast and can last trillions of cycles. You can run this demo for the remainder of the 21st century!

Here is the code. It pauses when it starts up, so you can read the last entry.

```
/*  
  
  ESP32 FRAM Demo  
  
  esp32-fram-demo.ino  
  
  Writes web page to external Flash Memory  
  
  Uses MB85RC256V FRAM module  
  
  Requires Adafruit_FRAM_I2C Library  
  Requires Adafruit_BME280 Library  
  
  
  How it works:  
  
  1. In setup(), it tries to read the last saved temperature from a known  
     address in the FRAM. If valid data is found, it reports this  
     "recovered" value to the Serial Monitor.  
  
  2. In the loop(), it reads the current temperature from the BME280  
     sensor every 200 milliseconds.  
  
  3. It immediately writes this new temperature value to the FRAM.  
  
  
  The Demo:  
  
  - Let the sketch run for a few seconds.  
  - Unplug the ESP32-C6 to simulate a power failure.  
  - Plug it back in. Observe the "Recovered last known temperature"  
    message in the Serial Monitor, proving the data survived.  
  
  
  DroneBot Workshop 2025  
  
  https://dronebotworkshop.com  
  
*/  
  
  
// Include Required Libraries  
  
#include <Wire.h>
```

```
#include <Adafruit_FRAM_I2C.h>

#include <Adafruit_Sensor.h>

#include <Adafruit_BME280.h>


// I2C Pin Configuration (change if required)

const int I2C_SDA_PIN = 10;

const int I2C_SCL_PIN = 11;


// Create the FRAM and BME280 objects

Adafruit_FRAM_I2C fram = Adafruit_FRAM_I2C();

Adafruit_BME280 bme;


// We'll store the temperature (a float, which is 4 bytes)

// at the very beginning of the FRAM memory, address 0.

#define TEMP_ADDRESS 0


void setup() {

  Serial.begin(115200);

  while (!Serial) {

    delay(10);

  }

  delay(1000);

  Serial.println("\n--- FRAM Power-Loss-Proof Logger ---");


  // Initialize the I2C bus with our defined pins.

  Wire.begin(I2C_SDA_PIN, I2C_SCL_PIN);


  // Initialize FRAM
```

```
if (!fram.begin()) {  
    Serial.println("Could not find a valid FRAM chip. Check wiring!");  
    while (1)  
        ;  
}  
  
Serial.println("FRAM chip initialized.");  
  
// Initialize BME280 at the detected address 0x77  
if (!bme.begin(0x77)) {  
    Serial.println("Could not find a valid BME280 sensor. Check wiring!");  
    while (1)  
        ;  
}  
  
Serial.println("BME280 sensor initialized.");  
  
// --- POWER-ON RECOVERY ---  
// Read the 4 bytes starting at TEMP_ADDRESS and interpret them as a float.  
float lastTemp = 0.0;  
fram.read(TEMP_ADDRESS, (uint8_t*)&lastTemp, sizeof(float));  
  
// isnan() checks for "Not a Number". Uninitialized FRAM often has this  
if (!isnan(lastTemp) && lastTemp > -50.0 && lastTemp < 100.0) {  
    Serial.println("-----");  
    Serial.print(">> Recovered last known temperature: ");  
    Serial.print(lastTemp);  
    Serial.println(" C");  
    Serial.println("-----");  
} else {
```



```
    Serial.println("No valid previous data found in FRAM.");
}

// Slight Delay
delay(1500);

Serial.println("Starting real-time logging...");
}

void loop() {

    // Read the current temperature from the BME280
    float currentTemp = bme.readTemperature();

    // Check if the read was successful
    if (isnan(currentTemp)) {
        Serial.println("Failed to read from BME sensor!");
        return;
    }

    // Print the current temperature to the serial monitor
    Serial.print("Logging temperature: ");
    Serial.print(currentTemp);
    Serial.println(" C");

    // Write the new temperature value to FRAM, overwriting the old one.
    // This happens on every loop, demonstrating high write endurance.
    fram.write(TEMP_ADDRESS, (uint8_t*)&currentTemp, sizeof(float));
}
```

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>

```
// Wait a short amount of time before the next reading.  
  
delay(200);  
}
```

This sketch uses the *Adafruit_FRAM_I2C* library to communicate with the FRAM over I²C and the *Adafruit_BME280* library to read temperature data from the BME280 environmental sensor.

The I²C pins are defined as *I2C_SDA_PIN* and *I2C_SCL_PIN*, and the constant *TEMP_ADDRESS* specifies the FRAM memory location (address 0) where the temperature value will be stored as a 4-byte float.

In *setup()*, the code initializes the serial monitor, the I²C bus, the FRAM chip, and the BME280 sensor.

It then attempts a power-on recovery by reading the last stored temperature from FRAM using *fram.read()*. If the retrieved value is valid (not NaN and within a reasonable range), it prints the recovered temperature to the serial monitor. If no valid data is found, it reports that as well.

The *loop()* runs continuously, reading the current temperature from the BME280 with *bme.readTemperature()*. If the reading is valid, it prints the value to the serial monitor and immediately writes it to FRAM using *fram.write()*, overwriting the previous value. This happens every 200 ms.

Load the sketch and run it. Watch the temperature as it logs it to the FRAM.

Now disconnect the board for a while and then plug it back in. Note how it retained the last reading and continues to populate the log with temperature readings.

For more projects and tutorials visit the DroneBot Workshop - <https://dronebotworkshop.com>



```
esp32-fram-demo.ino
1  /*
2  ESP32 FRAM Demo

Serial Monitor X
Message (Enter to send message to 'ESP32C6 Dev Module' on 'COM25') New Line 115200 baud
--- FRAM Power-Loss-Proof Logger ---
FRAM chip initialized.
BME280 sensor initialized.
-----
>> Recovered last known temperature: 24.79 C
-----
Starting real-time logging...
Logging temperature: 25.06 C
Logging temperature: 25.06 C
Logging temperature: 25.07 C
Logging temperature: 25.07 C
Logging temperature: 25.08 C
Logging temperature: 25.08 C
Logging temperature: 25.08 C
Logging temperature: 25.08 C
```

DroneBotWorkshop.com
Ln 104, Col 27 ESP32C6 Dev Module on COM25

At the rate this is logging data, it would burn out a Flash memory (internal or external) in less than three days—no worries with a FRAM, which can keep going on for decades.

For data that changes rapidly or that needs to be retained for a very long time, FRAM is an excellent choice.

<https://dronebotworkshop.com>

Conclusion

As we have seen, there are several non-volatile storage options for the ESP32.

The choice between internal NVS, EEPROM, SPI Flash, and FRAM isn't simply about picking the "best" option – it's about matching the proper storage technology to your specific application requirements and project goals.

- **NVS / LittleFS** – Best for small to medium data stored in the ESP32's built-in flash.
- **EEPROM (24LC256)** – Simple, cheap, and great for occasional writes.
- **SPI Flash (W25Q64)** – Large, fast, and perfect for web and log files.
- **FRAM (MB85RC256V)** – Best for high-frequency updates.

All of these options will allow you to create an "ESP32 that never forgets". By understanding these powerful memory technologies, you can take your projects to the next level.