# Flash Save System Documentation

Your Name

November 8, 2025

## Contents

# 1 Overview

This document describes a flash-based data save system for microcontrollers using SPI NOR flash. The system is designed to store sequential messages without overwriting existing data, using a metadata tally to track free and used blocks. It is suitable for short mission logging or small payload storage.

## 1.1 File Structure

| File | Purpose |
|---|---|
| flash.h | Defines SPI flash interface, constants, and function prototypes |
| flash.cpp | Implements SPI communication, read/write operations, and sector erase |
| save.h | Declares save system API, metadata helpers, and constants |
| save.cpp | Implements sequential save logic using a metadata tally |
| main.cpp | Example/test program demonstrating usage of the system |

# 2 Flash Chip Layout

## 2.1 Memory Partitioning

The 16 MB flash chip is divided into two regions:

| Region | Address Range | Purpose |
|---|---|---|
| Metadata sector | 0x000000 – 0x000FFF (4 KB) | Tracks which user data blocks are free/used |
| User data region | 0x001000 – 0xFFFFFF | Stores actual saved data sequentially |

## 2.2 Metadata Tally Mapping

- Metadata sector: 4 KB = 4096 bytes = 32,768 bits
- Each bit represents one user data block of 512 bytes
- Bit value: 1 = free, 0 = used

**Example: First Metadata Byte (8 blocks)**

```
Bit:   7 6 5 4 3 2 1 0
Value: 1 1 1 1 1 1 1 1  <- all blocks free
```

After writing the first block:

```
Bit:   7 6 5 4 3 2 1 0
Value: 0 1 1 1 1 1 1 1  <- first block used
```

—

# 3 Operation Flow

## 3.1 Initialization

1. Call `flashInit()` to configure SPI and CS pin.
2. Metadata sector is read; if needed, it is erased to all 1s.

## 3.2 Saving Data

1. Check data size ($\leq$ FLASH_BLOCK_SIZE and not empty).
2. Find next free block using `flashGetNextFreeAddr()`.
3. Write data with `flashWrite()`.
4. Wait for write completion using `flashIsBusy()`.
5. Update metadata using `flashAdvanceNextFreeAddr()`.
6. Return flash address of the saved block.

## 3.3 Reading Data

Use `flashRead(address, buffer, length)`:

```
uint8_t buffer[64];
flashRead(addr, buffer, len);
buffer[len] = '\0';
Serial.println((char*)buffer);
```

—

# 4 Design Decisions

| Decision | Reasoning |
|---|---|
| Tally-based metadata | Allows sequential writes without overwriting, simple and efficient |
| Block size (512 B) | Matches typical payloads, aligns with 256 B page, balances RAM usage |
| $1 \to 0$ bit marking | Matches NOR/EEPROM behavior: write $1 \to 0$, erase for $0 \to 1$ |
| Metadata erase | Minimizes wear, only frequent writes are to metadata sector |
| Sequential writes | Simplifies logic, avoids complex wear leveling |
| Byte-level read/write for metadata | Reduces RAM usage, compatible with small MCUs |

—

# 5 Usage Instructions

## 5.1 Saving Data

```
const char msg[] = "HELLO WORLD";
uint32_t addr = saveState(msg, strlen(msg));
if(addr == 0) Serial.println("Save failed!");
```

## 5.2 Reading Data

```
uint8_t buffer[64];
flashRead(addr, buffer, strlen(msg));
buffer[strlen(msg)] = '\0';
Serial.println((char*)buffer);
```

—

# 6 Limitations and Considerations

- Metadata sector wear may accumulate in long missions.
- Maximum write per block = FLASH_BLOCK_SIZE (512 bytes by default).
- Power loss mid-write may cause inconsistent state.
- Sequential writes only; old blocks are not erased automatically.
- Ensure SPI clock is compatible with flash chip.

—

# 7 Example Sequence

1. Initialize flash: `flashInit()`.
2. Save first message: `saveState("Message1", 11)`.
3. Save second message: `saveState("Message2", 14)`.
4. Read back messages with `flashRead()`.
5. Metadata shows used/free blocks accordingly.

—

# 8 Conclusion

The system provides a simple, robust sequential flash storage mechanism:

- Tracks free blocks via a metadata tally.
- Writes sequentially without overwriting.
- Handles flash-specific write behavior ($1 \rightarrow 0$ vs sector erase).
- Provides a clean API: `saveState()` and `flashRead()`.
- Suitable for short missions, small messages, or logging.