Logging ADXL Data to FAT32 on STM32F411

A Follow-On to `cmsis_adxl_sdcard`

Project Notes

February 25, 2026

Contents

Chapter 1

From Raw Sectors to Files

This chapter is a continuation of the raw-sector project chapter for `cmsis_adxl_sdcard`. It assumes you already understand:

- SPI transaction framing and chip-select timing,
- SD card SPI initialization (CMD0/CMD8/ACMD41/CMD58), and
- 512-byte block read/write primitives.

What changes here is **where writes are targeted**: not fixed LBAs, but a named file (`0:adxl_log.csv`) managed by FatFs.

1.1 Design Delta from the Raw-Sector Project

The two projects share the same hardware split:

- SPI1 for ADXL345
- SPI2 for microSD
- USART2 TX for live monitoring

The key software delta is layering:

1. SD SPI driver still provides sector-level operations.
2. A `diskio` port maps FatFs requests to those sector operations.
3. Application code uses FatFs file APIs (mount, open, write, sync).

1.2 Application Flow with FatFs

The main loop still reads ADXL and prints UART telemetry, but logging now goes through `f_write()` into a CSV file.

Listing 1.1: Main FatFs logging flow (condensed from src/main.c)

```
1 FATFS fs;
2 FIL file;
```

```
FRESULT fr;

fr = f_mount(&fs, "0:", 1);
if (fr != FR_OK) {
  while (1) {
    printf("f_mount failed: %d\n", (int)fr);
    systick_msec_delay(500);
  }
}

fr = f_open(&file, "0:adxl_log.csv", FA_OPEN_APPEND | FA_WRITE);
if (fr != FR_OK) {
  while (1) {
    printf("f_open failed: %d\n", (int)fr);
    systick_msec_delay(500);
  }
}

if (f_size(&file) == 0U) {
  const char *header = "sample,ax_mg,ay_mg,az_mg\r\n";
  UINT bw = 0U;
  fr = f_write(&file, header, (UINT)strlen(header), &bw);
  if ((fr != FR_OK) || (bw != (UINT)strlen(header))) {
    while (1) {
      printf("header write failed: %d\n", (int)fr);
      systick_msec_delay(500);
    }
  }
  (void)f_sync(&file);
}

while (1) {
  // read ADXL sample and convert to mg (same as prior chapter)
  // ...

  char line[128];
  int n = snprintf(line, sizeof(line), "%lu,%ld,%ld,%ld\r\n", ...);
  if (n > 0) {
    UINT bw = 0U;
    printf("sample=%lu ...\n", ...); // UART remains active

    fr = f_write(&file, line, (UINT)n, &bw);
    if ((fr != FR_OK) || (bw != (UINT)n)) {
      printf("f_write failed: %d\n", (int)fr);
    }

    if ((sample_index % 10U) == 0U) {
      fr = f_sync(&file);
      if (fr != FR_OK) {
        printf("f_sync failed: %d\n", (int)fr);
      }
    }
  }
}
```

1.3 Why f_sync() Every N Samples?

FatFs and SD cards buffer state. If power drops before sync/close, recent data and metadata can be lost.

This project uses a periodic sync policy (every 10 samples):

- lower loss window than syncing only at shutdown,
- less overhead than syncing every single line,
- practical for interactive logging while debugging firmware.

1.4 The Disk I/O Bridge: FatFs to SPI Blocks

FatFs is filesystem logic. It still needs low-level block device hooks. That contract lives in src/fatfs/diskio_port.c

Listing 1.2: Core diskio mapping in this project

```
DSTATUS disk_initialize(BYTE pdrv) {
  if (pdrv != 0U) return STA_NOINIT;
  return sdcard_spi_init() ? 0U : STA_NOINIT;
}

DRESULT disk_read(BYTE pdrv, BYTE *buff, LBA_t sector, UINT count) {
  if ((pdrv != 0U) || (buff == 0) || (count == 0U)) return RES_PARERR;
  for (UINT i = 0; i < count; i++) {
    if (!sdcard_spi_read_block((uint32_t)sector + i, &buff[i * 512U])) {
      return RES_ERROR;
    }
  }
  return RES_OK;
}

DRESULT disk_write(BYTE pdrv, const BYTE *buff, LBA_t sector, UINT count) {
  if ((pdrv != 0U) || (buff == 0) || (count == 0U)) return RES_PARERR;
  for (UINT i = 0; i < count; i++) {
    if (!sdcard_spi_write_block((uint32_t)sector + i, &buff[i * 512U])) {
      return RES_ERROR;
    }
  }
  return RES_OK;
}
```

Conceptually, this layer is the most important addition versus the raw-sector project. It lets FatFs treat your SD SPI driver as a generic block device.

1.5 SD Driver Changes for FatFs

Compared with the raw-sector project, the SD SPI module adds block read support:

- `CMD17` single-block read path,
- wait for start token `0xFE`,
- receive exactly 512 bytes plus 2 CRC bytes.

Listing 1.3: Single-block read in SPI mode (condensed)

```
bool sdcard_spi_read_block(uint32_t lba, uint8_t *data_512) {
  // ... send CMD17 ...

  // Wait for start token 0xFE
  for (uint32_t i = 0; i < 100000U; i++) {
    if (!spi_txrx(0xFFU, &rx)) return false;
    if (rx == 0xFEU) break;
  }
  if (rx != 0xFEU) return false;

  for (uint32_t i = 0; i < 512U; i++) {
    if (!spi_txrx(0xFFU, &data_512[i])) return false;
  }

  // ignore 2 CRC bytes when SPI CRC checking is disabled
  if (!spi_txrx(0xFFU, &rx) || !spi_txrx(0xFFU, &rx)) return false;
  return true;
}
```

1.6 CSV Format and Host Workflow

Because this project writes through FatFs, the card remains host-friendly:

- file: `adxl_log.csv`
- header: `sample,ax_mg,ay_mg,az_mg`
- rows: one sample per line with CRLF endings

That means you can remove the card and open the file directly in spreadsheet or analysis tools, which is the practical advantage over raw fixed-LBA logging.

1.7 Error Behavior

The application handles errors in two categories:

- **startup-fatal**: mount/open/header failures stay in a visible error loop.
- **runtime-nonfatal**: write/sync failures print status over UART and keep sampling.

This preserves observability while exposing storage problems in real time.

1.8 Testing Checklist

1. Format SD card as FAT32.
2. Flash firmware and open UART at 115200 8N1.
3. Confirm boot banner and sample stream.
4. Power cycle once to verify append behavior (file grows, no header duplication).
5. Remove card and inspect `adxl_log.csv` on host.

Appendix: Relationship to the Previous Chapter

Use the raw-sector chapter for SPI/SD protocol grounding:

- `cmsis_adxl_sdcard/docs/cmsis_adxl_sdcard_chapter.pdf`

  Use this chapter for filesystem integration concepts:

- `cmsis_adxl_sdcard_fatfs/docs/cmsis_adxl_sdcard_fatfs_chapter.pdf`