STM32F411 Project Documentation Book

Project Notes

February 25, 2026

Contents

Chapter 1

From Sensor Sample to SD Card Sector

This chapter walks through the `cmsis_adxl_sdcard` project as a systems programmer would: one peripheral at a time, with emphasis on transaction boundaries, failure behavior, and data format.

1.1 What This Project Does

The firmware performs three jobs in a loop:

1. Read acceleration data from an ADXL345 using **SPI1**.
2. Print each sample to the console using **USART2 TX**.
3. Append the same line to a RAM buffer and periodically write 512-byte sectors to a microSD card using **SPI2**.

1.2 Hardware Partitioning

A key design choice is bus separation:

- ADXL345 remains on SPI1 (PA4–PA7).
- microSD uses SPI2 (PB12–PB15).
- UART telemetry uses USART2 TX on PA2.

This prevents SD traffic from disturbing sensor transfers and lets serial monitoring continue while SD logging is active.

1.3 Important Constraint: Raw Sectors, Not FAT

This project writes raw blocks with `CMD24`. It does **not** implement FAT32 metadata (file allocation table, directory entries, file size updates). In practice, this means:

- The card is useful for low-level learning and hex-dump inspection.
- The written data is not immediately a normal mountable text file.
- You should use a dedicated test card and carefully choose logical block addresses (LBAs).

1.4 Main Control Flow

The top-level loop is intentionally simple: initialize peripherals, read ADXL samples, format a text line, send to UART, and append/flush to SD.

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

```c
#define LOG_START_LBA 32768U
#define LOG_SECTOR_SIZE 512U

static uint8_t adxl_raw[6];
static uint8_t log_sector[LOG_SECTOR_SIZE];
static uint32_t log_sector_offset = 0U;
static uint32_t next_log_lba = LOG_START_LBA;
static bool sd_available = false;

int main(void) {
  system_init();
  led_init();
  uart_init();

  if (!adxl_init()) {
    while (1) {
      printf("ADXL345 init failed\\n");
      systick_msec_delay(500);
    }
  }

  sd_available = sdcard_spi_init();

  uint32_t sample_index = 0U;
  while (1) {
    led_toggle();

    if (!adxl_read(ADXL345_REG_DATA_START, adxl_raw)) {
      printf("ADXL345 read timeout\\n");
      systick_msec_delay(100);
      continue;
    }

    int16_t ax = (int16_t)(((uint16_t)adxl_raw[1] << 8) | adxl_raw[0]);
    int16_t ay = (int16_t)(((uint16_t)adxl_raw[3] << 8) | adxl_raw[2]);
    int16_t az = (int16_t)(((uint16_t)adxl_raw[5] << 8) | adxl_raw[4]);

    int32_t ax_mg = (int32_t)ax * 39 / 10;
    int32_t ay_mg = (int32_t)ay * 39 / 10;
    int32_t az_mg = (int32_t)az * 39 / 10;

    char line[128];
    int n = snprintf(line, sizeof(line),
      "sample=%lu, ax=%ldmg, ay=%ldmg, az=%ldmg\\n",
      (unsigned long)sample_index,
      (long)ax_mg, (long)ay_mg, (long)az_mg);

    if (n > 0) {
      printf("%s", line);                    // live monitoring over UART
      if (sd_available && !log_append_line(line)) {
        printf("microSD append failed; disabling SD logging\\n");
        sd_available = false;
      }
    }

    if (sd_available && (sample_index % 16U == 15U) && (log_sector_offset > 0U
      )) {
      if (!log_flush_sector()) {
        printf("microSD flush failed; disabling SD logging\\n");
        sd_available = false;
      }
    }

    sample_index++;
    systick_msec_delay(100);
  }
}
```

Why the 16-sample periodic flush?

Without flushing, a partial sector stays in RAM until full. Periodic flushing gives earlier on-card visibility, which is useful during bring-up and debugging.

1.5 SPI2 Transport Layer for microSD

`src/spi2_sd.c` is the byte-transport layer. It owns GPIO muxing and SPI2 timing.

Pin and mode setup

- PB13/PB14/PB15 configured AF5 for SCK/MISO/MOSI.
- PB12 configured as software-controlled chip-select.
- SPI mode 0 (CPOL=0, CPHA=0), 8-bit frames, master mode.
- Start at very low speed (BR=/256), then increase after card init.

Listing 1.2: SPI2 byte transfer with timeout guards

```
bool spi2_sd_transfer(uint8_t tx, uint8_t *rx) {
  if (rx == 0) return false;

  if (!spi2_wait_set(&SPI2->SR, SPI_SR_TXE)) return false;
  *(__IO uint8_t *)&SPI2->DR = tx;

  if (!spi2_wait_set(&SPI2->SR, SPI_SR_RXNE)) return false;
  *rx = *(__IO uint8_t *)&SPI2->DR;

  if (!spi2_wait_set(&SPI2->SR, SPI_SR_TXE)) return false;
  if (!spi2_wait_clear(&SPI2->SR, SPI_SR_BSY)) return false;

  return true;
}
```

The final TXE/BSY checks are important: they keep chip-select transitions aligned with real wire activity.

1.6 SD Card SPI Protocol Layer

`src/sdcard_spi.c` builds SD commands on top of `spi2_sd_transfer()`.

Initialization sequence

Initialization follows the standard SPI-mode progression for SD v2 cards:

1. Send idle clocks with CS high.
2. Assert CS, issue `CMD0` to enter idle state.
3. Issue `CMD8` and validate echo pattern `0x1AA`.
4. Loop `CMD55 + ACMD41(HCS)` until R1 becomes `0x00`.
5. Issue `CMD58` and inspect OCR for high-capacity support.
6. If not high capacity, issue `CMD16` for 512-byte blocks.
7. Deassert CS, send one extra idle byte, then raise SPI clock.

Listing 1.3: Representative command framing in SPI mode

```
static bool sd_send_cmd(uint8_t cmd, uint32_t arg, uint8_t crc, uint8_t *r1) {
  uint8_t pkt[6];
  pkt[0] = (uint8_t)(0x40U | cmd);
  pkt[1] = (uint8_t)(arg >> 24);
  pkt[2] = (uint8_t)(arg >> 16);
  pkt[3] = (uint8_t)(arg >> 8);
  pkt[4] = (uint8_t)(arg >> 0);
  pkt[5] = crc;

  // one idle byte before command
  uint8_t rx = 0U;
  if (!spi_txrx(0xFFU, &rx)) return false;

  for (uint32_t i = 0; i < 6U; i++) {
    if (!spi_txrx(pkt[i], &rx)) return false;
  }

  return sd_wait_r1(r1, 16U);
}
```

Single-block write path (CMD24)

For each 512-byte payload:

1. Convert LBA to byte address only on SDSC cards.
2. Send `CMD24` and require R1=0.
3. Send start token `0xFE`.
4. Send 512 data bytes.
5. Send two dummy CRC bytes.
6. Verify data response token accepted.
7. Poll busy state until card releases MISO (returns `0xFF`).

This maps cleanly to how SD cards internally program NAND pages while host-side SPI clocks continue.

1.7 Log Buffering Strategy

A RAM sector buffer collects variable-length text lines and writes full blocks.

Listing 1.4: Append and flush behavior

```c
static bool log_append_line(const char *line) {
  size_t len = strlen(line);
  size_t i = 0U;

  while (i < len) {
    if (log_sector_offset >= LOG_SECTOR_SIZE) {
      if (!log_flush_sector()) return false;
    }

    uint32_t remaining = LOG_SECTOR_SIZE - log_sector_offset;
    size_t chunk = len - i;
    if (chunk > remaining) chunk = remaining;

    memcpy(&log_sector[log_sector_offset], &line[i], chunk);
    log_sector_offset += (uint32_t)chunk;
    i += chunk;
  }

  return true;
}
```

The design keeps write calls aligned to SD native block size, avoiding partial-block protocol complexity.

1.8 ADXL345 Read Path on SPI1

The ADXL driver performs register-level operations and multi-byte reads:

- Set read bit and multibyte bit in command address.
- Assert CS, transmit address, then read 6 bytes (X/Y/Z LSB/MSB pairs).
- Deassert CS.

Listing 1.5: ADXL345 burst read of XYZ registers

```c
bool adxl_read(uint8_t address, uint8_t *rxdata) {
  if (rxdata == 0U) return false;

  address |= ADXL345_READ_OPERATION;
  address |= ADXL345_MULTI_BYTE_ENABLE;

  cs_enable();
  if (!spi1_transmit(&address, 1)) {
    cs_disable();
    return false;
  }
  if (!spi1_receive(rxdata, 6)) {
    cs_disable();
    return false;
  }
  cs_disable();
  return true;
}
```

1.9 USART Output for Live Monitoring

The project redirects `printf` through `_write()` into USART2 TX. Every sample is printed even if SD logging later fails. This is a deliberate diagnostics choice: telemetry survives partial storage failures.

1.10 Failure Handling Philosophy

The code uses conservative fail-safe behavior:

- If ADXL init fails, firmware stays in a visible error loop.
- If SD init fails, firmware continues with UART only.
- If SD append/flush fails later, SD logging is disabled, UART continues.
- SPI/UART low-level waits include timeout guards to avoid deadlock.

1.11 What You Learn from This Project

This project is a strong first step for STM32 storage work because it isolates core concerns:

- Distinct SPI buses for distinct devices.
- Deterministic command framing and chip-select control.
- Sector-oriented buffering.
- Robustness with timeout and graceful degradation.

1.12 Natural Next Step: FAT-Aware Logging

Once this raw-sector flow is clear, the next step is filesystem-aware writing (FatFs), where file creation, append semantics, and directory updates are handled for you.

Appendix: Build This Chapter PDF

From the repository root:

```
cd docs
pdflatex cmsis_adxl_sdcard_chapter.tex
pdflatex cmsis_adxl_sdcard_chapter.tex
```

Two passes are typical so the table of contents resolves page numbers.

Chapter 2

From Raw Sectors to Files

This chapter is a continuation of Chapter 1 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.

2.1 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).

2.2 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 2.1: Main FatFs logging flow (condensed from src/main.c)

```
FATFS fs;
FIL file;
```

```c
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);
      }
    }
  }
}
```

2.3 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.

2.4 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 2.2: Core diskio mapping in this project

```c
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.

2.5 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 2.3: Single-block read in SPI mode (condensed)

```c
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;
}
```

2.6 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.

2.7 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.

2.8 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 Chapter 1 for SPI/SD protocol grounding:

- Raw-sector chapter in this combined book (label: `ch:raw-sd`)

  Use Chapter 2 for filesystem integration concepts:

- FatFs chapter in this combined book (label: `ch:fatfs-followon`)

Cross-Chapter References

This combined book supports cross-references between chapters. For example, the raw-sector chapter is Chapter 1, and the FatFs follow-on chapter is Chapter 2.