

hola, estoy aprendiendo como leer una tarjeta SD usando un microcontrolador AVR (en concreto el atmega168pa), he comprado un modulo de lector de tarjetas microsd, me interesa saber como funciona este modulo que he comprado, es un circuito con algunas resistencias y condensadores, y la parte principal entiendo que son dos ICs que tiene, te digo los códigos que tienen escritos encima, me interesaria saber que hacen cada uno de los IC,

uno es: 1117-3.3 XBLW 2517A

y el otro es: SN74HC125 LX45A507

¡Perfecto! Esos dos componentes son muy típicos en los módulos de lector de microSD para microcontroladores como tu ATmega168PA. Te explico cuál es el papel de cada uno:

---

## 1. 1117-3.3 (AMS1117-3.3 o variante)

**Marcado:** 1117-3.3 XBLW 2517A

**Función:** Regulador de voltaje lineal de 3.3 V

### ✓ Qué hace

Convierte la alimentación de 5 V (la que probablemente usas con tu AVR o Arduino) a 3.3 V, que es la tensión necesaria para:

- La tarjeta microSD (siempre funciona a 3.3 V)
- La lógica adicional del módulo

### Características típicas:

- Salida fija: 3.3 V
- Corriente máxima: ~800 mA (depende del fabricante)
- Caída (dropout): ~1.1 V
- Protecciones básicas: térmica y de sobrecorriente

### Por qué está en tu módulo

Las tarjetas SD son **muy sensibles** al voltaje: **no toleran 5 V.**

El regulador se asegura de que **todo el lado de la SD esté alimentado correctamente.**

---

## 2. SN74HC125

**Marcado:** *SN74HC125 LX45A507*

**Función:** Cuádruple buffer/bidireccional line driver con salida tri-state

En un módulo de microSD, este IC normalmente funciona como **adaptador de nivel lógico (5 V → 3.3 V).**

### ✓ Qué hace

El SN74HC125 incluye **4 buffers independientes** que:

- Aceptan señales **de 5 V** en sus entradas
- Generan señales **de 3.3 V** (porque está alimentado con 3.3 V del regulador)
- Tienen salida **tri-state**, lo cual permite desconectar las líneas cuando no se usan

### ¿Para qué líneas?

Típicamente protege/convierte las líneas SPI:

- **MOSI**
- **SCK**
- **CS**

La línea **MISO** suele venir directamente desde la tarjeta (que ya es 3.3 V), así que no necesita conversión.

### Por qué se usa este chip

La SD es **strict 3.3 V.**

Si conectas señales SPI de 5 V directamente desde un AVR, puedes **dañar la tarjeta SD.**

Este buffer evita eso.

---

## ¿Cómo se organiza todo en el módulo?

scss

 Copiar código

5V → AMS1117-3.3 → Alimentación de SD y SN74HC125 (3.3 V)

Señales SPI del microcontrolador (5V)

→ SN74HC125 → SD (3.3V)

Salida MISO de la SD (3.3V)

→ microcontrolador (muchos AVRs lo reconocen como HIGH vía

## 📌 Resumen

Componente	Función	Uso en tu módulo
1117-3.3	Regulador de 3.3 V	Alimentar la tarjeta SD y la lógica
SN74HC125	Adaptador de nivel (buffers tri-state)	Convertir señales SPI de 5 V a 3.3 V

Si quieras, puedo explicarte **cómo inicializar la tarjeta SD en modo SPI**, o incluso ayudarte con **código para el ATmega168PA** (C o AVR-GCC). ¿Quieres avanzar hacia la parte de software?

🔗 🤝 🗞 ⬆️ ...

Si, explícame como haría para leer el contenido de la SD, me vale una lectura RAW, no necesito utilizar un sistema de archivos, con que me expliques como leer un byte (el primero mismamente) ya me vale.

Perfecto, te explico **cómo leer un solo byte RAW de la tarjeta SD usando SPI** con un ATmega168PA, *sin sistema de archivos*.

Leer RAW significa simplemente pedirle a la tarjeta que te envíe un bloque de 512 bytes (sector) y tú luego coges el primer byte.

## ★ RESUMEN RÁPIDO DEL PROCESO

1. Inicializar SPI (modo 0, baja velocidad)
2. Mandar CMD0 → poner SD en modo SPI
3. Mandar CMD8 → comprobar voltaje
4. Mandar ACMD41 → inicialización
5. Mandar CMD17 → leer un bloque de 512 bytes
6. Recibir token 0xFE
7. Leer 512 bytes → el *primer byte* ya es tu dato
8. Ignorar CRC (2 bytes)

### 1. Inicializar SPI (modo SPI a baja velocidad)

Durante la inicialización la SD requiere **frecuencia ≤ 400 kHz**.

C

[Copiar código](#)

```
void spi_init() {
    // MOSI, SCK, SS como salida
    DDRB |= (1<<PB3) | (1<<PB5) | (1<<PB2);
    // MISO como entrada
    DDRB &= ~(1<<PB4);

    // SPI enable, master, clock = f/128 (para ~125kHz a 16 MHz)
    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1) | (1<<SPR0);
}
```

Enviar y recibir un byte:

C

[Copiar código](#)

```
uint8_t spi_transfer(uint8_t data) {
    SPDR = data;
    while (!(SPSR & (1<<SPIF)));
    return SPDR;
}
```

### 2. Función para enviar comandos SD

Un comando SPI siempre tiene la forma:

- Byte 1: 0x40 | comando
- Byte 2-5: argumento (4 bytes)
- Byte 6: CRC (solo importa para CMD0 y CMD8)

Respuesta: un byte tipo R1 (bit7=0 cuando el byte es válido)

C

[Copiar código](#)

```
uint8_t sd_cmd(uint8_t cmd, uint32_t arg, uint8_t crc)
{
    spi_transfer(0x40 | cmd);
    spi_transfer(arg >> 24);
    spi_transfer(arg >> 16);
    spi_transfer(arg >> 8);
    spi_transfer(arg);

    spi_transfer(crc);

    // La SD puede responder durante varios bytes, esperar hasta que bit7 :
    uint8_t r;
    for (uint8_t i=0; i<10; i++) {
        r = spi_transfer(0xFF);
        if (!(r & 0x80)) break;
    }
    return r;
}
```



### 3. Rutina de inicialización SD en modo SPI

#### CMD0 – reset

C

[Copiar código](#)

```
uint8_t r = sd_cmd(0, 0, 0x95);
```

Debe devolver **0x01** (idle).

#### CMD8 – chequeo de voltaje

C

[Copiar código](#)

```
r = sd_cmd(8, 0x000001AA, 0x87);
```

## ACMD41 - arrancar

Es un comando especial: primero se manda **CMD55**, luego **CMD41**.

C

 Copiar código

```
do {
    sd_cmd(55, 0, 0x65);
    r = sd_cmd(41, 0x40000000, 0x77); // HCS=1
} while (r != 0x00);
```

Cuando responde **0x00**, la SD está inicializada.

## CMD58 - leer OCR (opcional pero recomendable)

C

 Copiar código

```
sd_cmd(58, 0, 0xFD);
spi_transfer(0xFF);
spi_transfer(0xFF);
spi_transfer(0xFF);
spi_transfer(0xFF);
```

## 4. Leer el primer sector (LBA = 0) con CMD17

C

 Copiar código

```
uint8_t read_sector_0(uint8_t *buffer) {
    uint8_t r = sd_cmd(17, 0, 0xFF); // sector 0

    if (r != 0x00) return 1; // error

    // Esperar token 0xFE
    uint8_t token;
    do {
        token = spi_transfer(0xFF);
    } while (token == 0xFF);
```

```

if (token != 0xFE) return 2; // token incorrecto

// Leer 512 bytes
for (int i=0; i<512; i++) {
    buffer[i] = spi_transfer(0xFF);
}

// Ignorar CRC
spi_transfer(0xFF);
spi_transfer(0xFF);

return 0; // OK
}

```

---

## ★ 5. Leer el primer byte

Imagina que tienes:

c Copiar código

```

uint8_t sector[512];

if (read_sector_0(sector) == 0) {
    uint8_t first_byte = sector[0];
    // Aquí ya tienes el primer byte RAW de la SD
}

```

Ese **primer byte** es exactamente el byte número 0 del sector lógico 0 de la tarjeta microSD.

---

## 🔌 Conexión de pines (SPI)

SD pin	Nombre	AVR pin
1	CS	PB2 (SS)
2	MOSI	PB3 (MOSI)
5	SCK	PB5 (SCK)
7	MISO	PB4 (MISO)

SD pin	Nombre	AVR pin
4	VDD	3.3 V
6	GND	GND

## 📎 ¿Quieres que prepare un proyecto completo AVR-GCC listo para compilar?

Puedo darte:

- Makefile
- main.c completo con inicialización + lectura
- Código optimizado
- Ejemplo de cómo imprimir el byte por UART

Solo dime qué usas: **AVR-GCC + avrdude**, Arduino IDE, PlatformIO, etc.



si, prepara el proyecto, uso avr-gcc + avrdude, pero no hagas que el byte se imprima por UART, simplemente haz que salga por el registro DDRD para que yo conecte 8 leds en ese registro y vea el propio byte como leds iluminados

### ⌚ Avr Sd Raw Project

Listo, tu proyecto AVR-GCC ya está generado en el canvas.

Siquieres, puedo añadir:

- Un **diagrama de conexiones**
- Una versión que lea **cualquier sector**
- Una rutina para **parpadear los LEDs** según el byte leído
- Una versión optimizada en **ensamblador AVR**

Solo dime.



Normalmente como se haría la lectura?? en bloques de 512 bytes o se puede cambiar para hacer lecturas mas grandes?

En las tarjetas SD **la unidad mínima e inamovible de lectura/escritura es siempre el bloque de 512 bytes.**

Esto viene definido en la especificación SD y **no se puede cambiar**, independientemente del tamaño de la tarjeta o del modo (SPI o SDIO).

---

## ✓ ¿Cómo se leen normalmente los datos?

**Siempre leyendo bloques de 512 bytes (sector lógico).**

Cuando envías un comando para leer, la tarjeta **solo puede devolver 512 bytes**:

- **CMD17** → leer 1 bloque (512 bytes)
  - **CMD18** → leer múltiples bloques (múltiples × 512 bytes)
- 

## ? ¿Puedo pedir un tamaño mayor, como 1024 o 4096 bytes?

No directamente.

Lo que sí puedes hacer es:

### ✓ 1. Leer varios bloques seguidos

Ejemplo: leer 4096 bytes = leer 8 bloques de 512.

Puedes usar:

### **CMD18 (Read Multiple Blocks)**

Ventaja:

- Más rápido: la SD no sale del modo lectura entre bloques.

Desventaja:

- Tienes que enviar **CMD12** al terminar la secuencia.

## 🔍 ¿Existen tarjetas con bloques físicos más grandes?

Sí, internamente las tarjetas SD modernas (SDHC/SDXC) tienen páginas de 4 KB o más, pero:

- Eso no afecta a la interfaz SPI
- La interfaz siempre expone sectores de 512 bytes

Incluso si el empaquetado interno es de 4096 bytes, la tarjeta traduce todo automáticamente.

## 🔧 ¿Puedo cambiar el tamaño del bloque con CMD16?

Históricamente existía:

- **CMD16 (SET\_BLOCKLEN)**: cambiar el tamaño del bloque

Pero:

👉 En tarjetas SDHC y SDXC CMD16 está deshabilitado

Siempre devuelven error.

Las tarjetas modernas ignoran cualquier valor que no sea 512.

Solo tarjetas **SDSC antiguas (<=2GB)** permitían cambiar el tamaño, pero hoy prácticamente ya no se encuentran.

## 🧠 Resumen final

Pregunta	Respuesta
¿Cuál es el tamaño mínimo de lectura?	<b>512 bytes</b>
¿Se puede leer más de 512?	Sí, múltiples de 512 usando CMD18
¿Se puede cambiar el tamaño del bloque?	No en SDHC/SDXC modernas
¿Qué tamaño usar en proyectos reales?	Siempre 512 bytes

Siquieres, te puedo añadir al proyecto existente:

- **Lectura usando CMD18 (múltiples bloques)**
- Un buffer circular para leer secuencias largas
- Un ejemplo de lectura RAW continua estilo *stream*

Solo dime qué necesitas.

🔗 🤝 🗔 ⬆️ ...

Cual es el tiempo normal de lectura? cada llamada que hago a "sd\_read\_sector\_0" me tarda como unas 4 decimas de segundo, es normal?

No, 0,4 segundos por leer un solo sector (512 bytes) NO es normal.

Eso indica claramente que estás leyendo muchísimo más lento de lo que debería.

Vamos a ver cuánto debería tardar realmente:

---

## ✓ Tiempo normal de lectura de un sector SD (512 bytes)

Hay dos factores:

---

### 1. Frecuencia SPI durante lectura

Después de la inicialización, se debe aumentar el reloj SPI a:

- **4 MHz**
- **8 MHz**
- Incluso **20-25 MHz** si el cableado es corto y limpio (límite de muchas SD)

Ejemplo con **8 MHz** (muy típico):

yaml

🔗 Copiar código

```
512 bytes × 8 bits = 4096 bits  
4096 bits / 8e6 = 0.000512 s = 512 microsegundos
```

Tiempo típico:

## 👉 Lectura de un sector ≈ 0,5 ms a 1 ms

Incluso con ineficiencia, esperas:

### 👉 Menos de 5 ms

---

## ✖️ 0,4 segundos = 400 ms → 100 veces más lento

Esto no es normal de ninguna manera.

---

## 🔍 ¿Qué está causando tu lentitud?

Casi seguro es una de estas causas:

---

### ⭐ 1. NO subiste la velocidad del SPI después de la inicialización

En tu código actual (el que está en el canvas), SPI queda fijo en:

nginx

 Copiar código

`F_CPU / 128 → 125 kHz`

Ese valor está bien para **inicializar la SD**, pero NO para leer datos.

La lectura quedará **800 veces más lenta** que con SPI a 100 MHz, y unas **40-80 veces más lenta** que a 5-10 MHz.

💥 Resultado práctico:

**≈ 0,3 a 0,5 segundos por sector**  
que es justo lo que estás viendo.

---

### ⭐ 2. Tienes que cambiar el SPI a velocidad alta tras `sd_init()`

Después de inicializar la SD, añade:

c

 Copiar código

```
void spi_set_fast() {
    // SPI = F_CPU / 2  (SPI2X = 1)
```

```

SPSR |= (1<<SPI2X);
SPCR &= ~((1<<SPR1) | (1<<SPR0)); // SPR1=SPR0=0 → F/4
// Con SPI2X esto da F/2 → 8 MHz si tu AVR va a 16 MHz
}

```

Y en main():

c

 Copiar código

```

sd_init();
spi_set_fast();

```

Esto baja el tiempo de lectura a:

👉 1 ms aprox.

---

## ⭐ 3. El módulo de la SD podría estar usando buffers lentos (poco probable)

Los módulos con 74HC125 y AMS1117 son los más típicos y normales.

Así que no es el problema.

---

## ⭐ 4. Retrasos excesivos esperando token 0xFE

Si la SD está vieja o lenta, puede tardar un poco, pero **nunca 400 ms**.

Tiempos típicos de espera:

- 1–5 ms en SDHC modernas
- 10–20 ms en tarjetas viejas
- 100 ms máximo en casos extremos

Nunca 400 ms continuos por cada lectura.

---

## 📌 Conclusión

ruby

 Copiar código

Lectura normal de sector: 0.5–3 ms

Tu lectura actual: 400 ms

=> SPI demasiado lento

### ✓ Solución:

- Cambiar SPI de F\_CPU/128 a F\_CPU/2 al terminar la inicialización

Si quieres, puedo actualizar **automáticamente el proyecto en el canvas** con:

- spi\_set\_fast()
- Mejoras generales
- Temporización más precisa
- Depuración por LED si falla un comando

Solo dímelos.



Puedes modificar la función "sd\_read\_sector\_0" para pasarle el parámetro del sector que quiero leer?

Avr Sd Raw Project

Copiar Editar Descargar

# Proyecto AVR: Lectura RAW de MicroSD y salida del primer byte por DDRD

A continuación tienes un proyecto mínimo para AVR-GCC que:

1. Inicializa SPI
2. Inicializa la tarjeta SD en modo SPI
3. Lee el sector 0 (LBA 0)
4. Muestra el primer byte leído en PORTD, configurando DDRD = 0xFF

Incluye:

- Makefile
- main.c

## Makefile

```
MCU=atmega168p
F_CPU=16000000UL
CC=avr-gcc
CFLAGS=-std=c11 -Os -DF_CPU=$(F_CPU) -mmcu=$(MCU)
OBJCOPY=avr-objcopy
TARGET=main

all: $(TARGET).hex

$(TARGET).elf: main.c
    $(CC) $(CFLAGS) -o $@ $^

$(TARGET).hex: $(TARGET).elf
    $(OBJCOPY) -O ihex -R .eeprom $< $@

flash: all
    avrdude -p $(MCU) -c usbasp -U flash:w:$(TARGET).hex

clean:
    rm -f *.elf *.hex
```

## main.c

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdint.h>

// =====
// SPI
// =====
void spi_init() {
    DDRB |= (1<<PB3) | (1<<PB5) | (1<<PB2); // MOSI, SCK,
    DDRB &= ~(1<<PB4); // MISO

    SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1) | (1<<SPR0); /
}

uint8_t spi_transfer(uint8_t data) {
```

```
SPDR = data;
while (!(SPSR & (1<<SPIF)));
return SPDR;
}

// =====
// SD Card (modo SPI)
// =====
uint8_t sd_cmd(uint8_t cmd, uint32_t arg, uint8_t crc) {
    spi_transfer(0x40 | cmd);
    spi_transfer(arg >> 24);
    spi_transfer(arg >> 16);
    spi_transfer(arg >> 8);
    spi_transfer(arg);
    spi_transfer(crc);

    uint8_t r;
    for (uint8_t i = 0; i < 10; i++) {
        r = spi_transfer(0xFF);
        if (!(r & 0x80)) break;
    }
    return r;
}

void sd_init() {
    // SD requiere >74 pulsos de reloj con CS alto
    PORTB |= (1<<PB2);
    for (int i=0; i<10; i++) spi_transfer(0xFF);

    // CMD0
    uint8_t r;
    do {
        PORTB &= ~(1<<PB2);
        r = sd_cmd(0, 0, 0x95);
        PORTB |= (1<<PB2);
    } while (r != 0x01);

    // CMD8
    PORTB &= ~(1<<PB2);
    sd_cmd(8, 0x000001AA, 0x87);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    spi_transfer(0xFF);
    PORTB |= (1<<PB2);
```

```
// ACMD41
do {
    PORTB &= ~(1<<PB2);
    sd_cmd(55, 0, 0x65);
    r = sd_cmd(41, 0x40000000, 0x77);
    PORTB |= (1<<PB2);
} while (r != 0x00);

// CMD58 (descartar OCR)
PORTB &= ~(1<<PB2);
sd_cmd(58, 0, 0xFF);
spi_transfer(0xFF);
spi_transfer(0xFF);
spi_transfer(0xFF);
spi_transfer(0xFF);
PORTB |= (1<<PB2);
}

uint8_t sd_read_sector(uint32_t lba, uint8_t *buf) {
    uint8_t r;

    PORTB &= ~(1<<PB2);
    r = sd_cmd(17, lba, 0xFF);
    if (r != 0x00) { PORTB |= (1<<PB2); return 1; }

    uint8_t token;
    do {
        token = spi_transfer(0xFF);
    } while (token == 0xFF);

    if (token != 0xFE) { PORTB |= (1<<PB2); return 2; }

    for (int i=0; i<512; i++)
        buf[i] = spi_transfer(0xFF);

    spi_transfer(0xFF);
    spi_transfer(0xFF);

    PORTB |= (1<<PB2);

    return 0;
}

uint8_t token;
do {
    token = spi_transfer(0xFF);
```

```
    } while (token == 0xFF);

    if (token != 0xFE) { PORTB |= (1<<PB2); return 2; }

    for (int i=0; i<512; i++)
        buf[i] = spi_transfer(0xFF);

    spi_transfer(0xFF);
    spi_transfer(0xFF);

    PORTB |= (1<<PB2);

    return 0;
}

// =====
// Main
// =====
int main() {
    uint8_t sector[512];

    DDRD = 0xFF; // LEDs en PORTD

    spi_init();
    _delay_ms(50);

    sd_init();

    uint8_t err = sd_read_sector(0, sector);

    if (err == 0) {
        PORTD = sector[0];
    } else {
        PORTD = 0xF0; // señal de error
    }
}
```

Listo, la función `sd_read_sector_0` ha sido reemplazada por la más general:

cpp

 Copiar código

`sd_read_sector(uint32_t lba, uint8_t *buf)`

y también ajusté la llamada en `main()` para usar el LBA 0 como ejemplo.

Si quieras, puedo añadir también:

- Versión que lea **múltiples sectores (CMD18)**