VIDEO#36: Electronic Basics #36: SPI and how to use it

SPI Communication with Arduino and DS3234 RTC

Introduction to Communication Protocols

In a previous discussion, we explored the **I2C (Inter-Integrated Circuit) protocol**, which allows a master device to communicate with multiple slave devices using two communication lines: **SCL (Serial Clock Line) and SDA (Serial Data Line)**. However, not all ICs use I2C. One such example is the **DS3234 Real-Time Clock (RTC)**, which instead utilizes the **SPI (Serial Peripheral Interface) protocol**.

Features of DS3234 RTC

The **DS3234 IC** is a real-time clock that tracks **seconds, minutes, hours, days, date, month, and year**. It also offers programmable **alarms and square wave signal outputs**. Unlike I2C-based RTCs, the DS3234 lacks SCL and SDA pins, instead featuring **CLK, MISO, MOSI, and SS pins**, indicating it operates using SPI communication.

Understanding SPI Communication

SPI is a high-speed serial communication protocol consisting of four main lines:

1. **CLK (Clock)** – A square wave signal generated by the master device to synchronize communication.
2. **MOSI (Master Out Slave In)** – Carries data from the master to the slave.
3. **MISO (Master In Slave Out)** – Carries data from the slave to the master.
4. **SS (Slave Select)** – Enables communication with a specific slave device.

SPI allows multiple slave devices but requires **one SS line per device**, making it less scalable compared to I2C.

Wiring the DS3234 RTC with Arduino

The **Arduino microcontroller** supports SPI functionality on the following pins:

- **PB4, PB3, PB5, and PB2**, which map to **digital pins 13, 12, 11, and 10**.

- The **SS (Slave Select) pin** can be any digital pin, but commonly **pin 10 is used**.

The DS3234 IC is wired to the **Arduino using simple hookup wires**:

- **CLK → Pin 13 (SCK)**
- **MISO → Pin 12**
- **MOSI → Pin 11**
- **SS → Pin 10 (or another digital pin)**

Including the SPI Library and Initializing Communication

To begin programming the RTC, the **SPI library** must be included:

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

Next, the **chip select (CS) pin is assigned**:

```
#define CS_PIN 10
```

The CS pin is **pulled low to start SPI communication** and **pulled high to end it**.

Initializing SPI Communication

The **RTC_init function** is used to initialize SPI communication by:

1. Declaring the **CS pin as an output**.
2. Setting **bit order to MSB first** (Most Significant Bit first) as per the DS3234 datasheet.

3. Defining **SPI mode 1** for proper data transmission.

SPI Modes

SPI operates in **four modes** based on clock polarity and phase:

- **Mode 0**: Clock normally low, data read on rising edge.
- **Mode 1**: Clock normally low, data read on falling edge (**Used in DS3234**).
- **Mode 2**: Clock normally high, data read on falling edge.
- **Mode 3**: Clock normally high, data read on rising edge.

Since the **DS3234 datasheet specifies SPI Mode 1**, the configuration is set accordingly.

Sending Data via SPI

When sending data to the DS3234:

1. The **CS pin is pulled low** to start communication.
2. The address register to be modified is sent (e.g., **0x8E** for control register).
3. Data to be written is sent (e.g., **0x60** to activate square wave output and temperature conversion).
4. The **CS pin is pulled high** to end communication.

Example:

SPI.transfer(0x8E);

SPI.transfer(0x60);

Verifying SPI Communication

To verify successful communication, a **square wave signal output** from the RTC is checked using an **oscilloscope**. When configured correctly, the DS3234 outputs a precise **8.192 kHz signal**.

Setting Date and Time

To set the **date and time** on the DS3234, the entered values (day, month, year, hour, minute, second) are converted into **binary-coded decimal (BCD)** format and written to the corresponding registers using a **for loop**.

Registers involved:

- **0x80** → Seconds

- **0x81** → Minutes
- **0x82** → Hours
- **0x84** → Day
- **0x85** → Month
- **0x86** → Year

Reading Date and Time

To read the date and time:

1. The **register address** is sent over SPI.
2. The received data is stored in an **integer variable**.
3. The data is **converted to human-readable format** using bit manipulation.

Comparing SPI and I2C

Comparing SPI and I2C

Feature	SPI	I2C
Number of wires	4 (MISO, MOSI, SCK, SS)	2 (SDA, SCL)
Speed	Up to **4 MHz** (fast)	Typically **100 kHz**
Scalability	Limited (one SS per slave)	High (multiple addresses)
Complexity	Moderate	Lower
Use case	High-speed communication	Multiple devices

Using SPI with Arduino enables **high-speed communication** with ICs like the **DS3234 RTC**. While **I2C allows multiple devices with fewer wires**, SPI provides **higher data rates** and is **ideal for applications requiring fast communication**, such as writing to **SD cards** or **real-time clock synchronization**.