

How to Write/Read Integers floats and 32bit Data

This is the 6th tutorial in the [W25Q Flash series](#), and today we will see how to update data in sectors. In the [previous tutorial](#) we saw how to update sectors, and how to Read/Write a single byte data without having to erase the entire sector for the same.

Today we will use the functions from the previous tutorial to read/write the integers, floats, and also the 32bit data array.

ADVERTISEMENT

Ads by



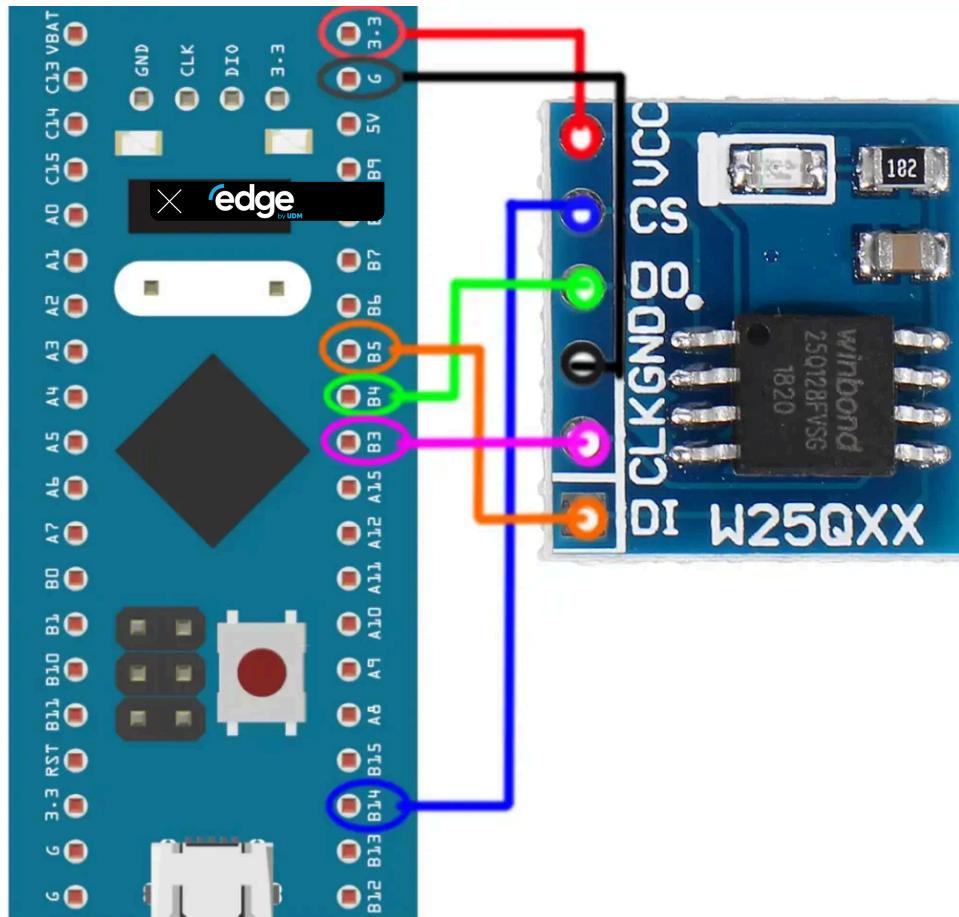


Connection

The W25Q NOR Flash memories can use the SPI in Single/Dual/Quad mode. For the simplicity of the operation, we will stick to the Single SPI mode, where we would need 4 pins to connect the Flash with the controller.

The connection diagram between the Flash module and the STM32 is shown below.



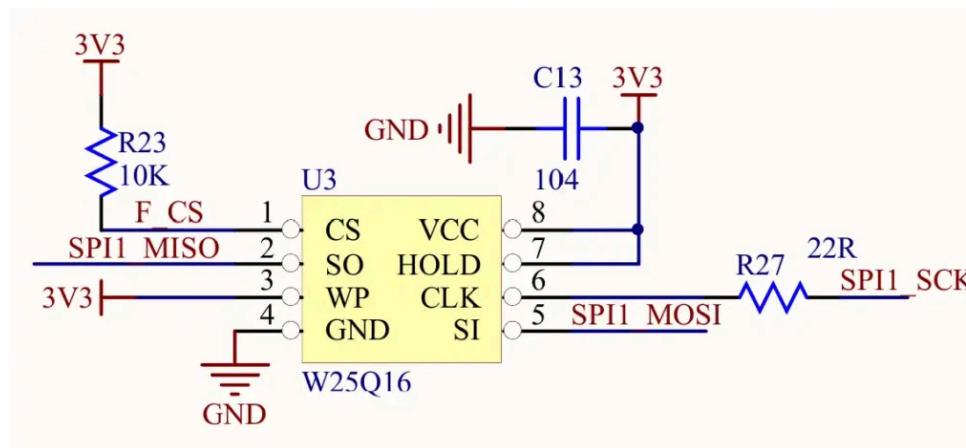


- The **CS** (Chip Select) Pin is connected to the pin PB14. It will be used to select or unselect the slave Device.
- The **DO** (Data Out) pin is connected to the pin PB4 (**MISO**). The device outputs the data on this pin
- The **CLK** (Clock) pin is connected to the pin PB3 (**CLK**). The clock is used to synchronise the master and the slave device.

- The **DI** (Data In) pin is connected to the pin PB5 (**MOSI**). The master send the data to the device on this pin.
- The Device is powered with .3.3v from the MCU itself.



The Module provides 6 pins (including the Vcc and Gnd). But the chip has 8 pins in total. If you have the chip rather than the module, you can connect it as shown below.



Note that the **WP** (Write Protect) pin is active Low pin, so it must be pulled HIGH when you want to modify the flash, and pulled LOW when you want to disable the modification.

The connections shown above are for the Single SPI mode, not for Dual or Quad modes.

CubeMX Setup



We will enable the SPI in **Full Duplex master mode**. The configuration is shown below.

SPI1 Mode and Configuration

Mode

Mode Full-Duplex Master

Hardware NSS Signal Disable

Configuration

Reset Configuration

NVIC Settings DMA Settings GPIO Settings

Parameter Settings User Constants

Configure the below parameters :

Search (Ctrl+F)

Basic Parameters

Frame Format Motorola

Data Size 8 Bits

First Bit MSB First

Clock Parameters

Prescaler (for Baud Rate) 32

Baud Rate 2.625 MBits/s

Clock Polarity (CPOL) Low

Clock Phase (CPHA) 1 Edge

Advanced Parameters

CRC Calculation Disabled

NSS Signal Type Software

MODE 0

The Data width is set to **8 bits** and the data should be transferred as the **MSB first**. The prescaler is set such that the Baud Rate is around **2.5 Mbits/sec**.

According to the datasheet  W25Q Flash, the SPI Mode 0 and Mode 3 are supported.

Below is the image from the datasheet.

6.1 Standard SPI Instructions

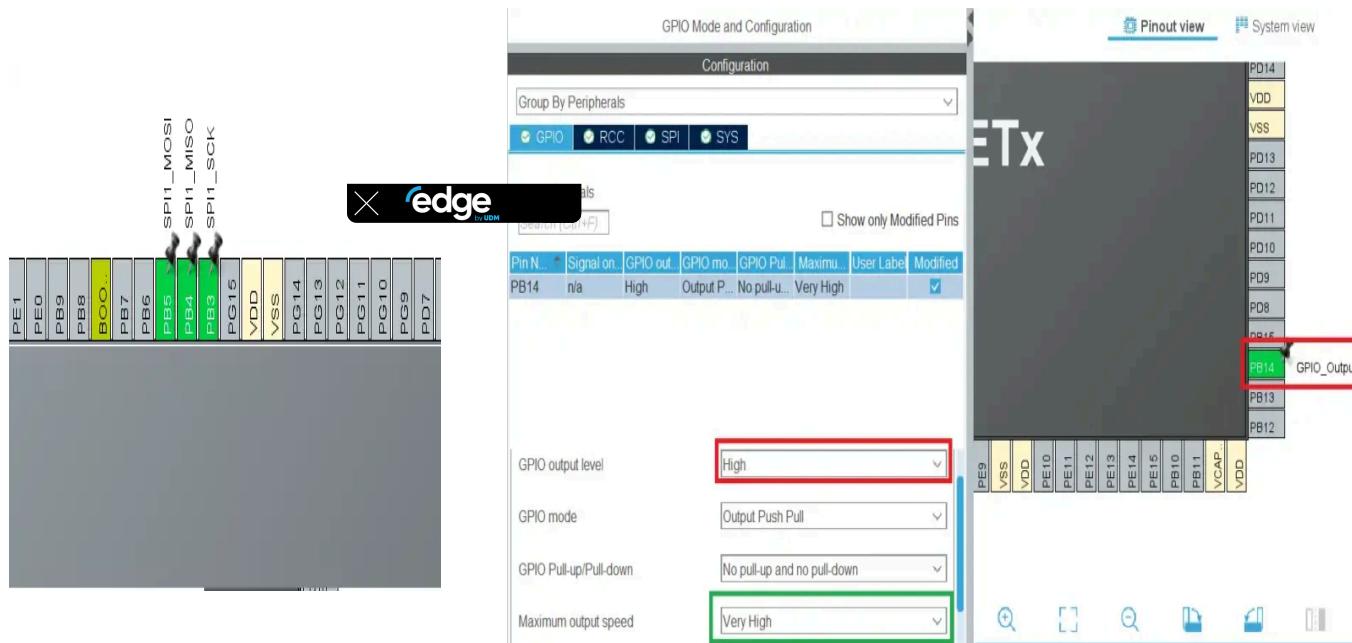
The W25Q16JV is accessed through an SPI compatible bus consisting of four signals: Serial Clock (CLK), Chip Select (/CS), Serial Data Input (DI) and Serial Data Output (DO). Standard SPI instructions use the DI input pin to serially write instructions, addresses or data to the device on the rising edge of CLK. The DO output pin is used to read data or status from the device on the falling edge of CLK.

SPI bus operation Mode 0 (0,0) and 3 (1,1) are supported. The primary difference between Mode 0 and Mode 3 concerns the normal state of the CLK signal when the SPI bus master is in standby and data is not being transferred to the Serial Flash. For Mode 0, the CLK signal is normally low on the falling and rising edges of /CS. For Mode 3, the CLK signal is normally high on the falling and rising edges of /CS.

In the SPI configuration, we keep the Clock Polarity (**CPOL**) **low** and the Clock Phase (**CPHA**) **to 1 edge**. Here 1 edge means that the data will be sampled on the **first edge of the clock**. And when the CPOL is Low, the first edge is the **rising edge**. Basically we are using the **SPI Mode 0**.

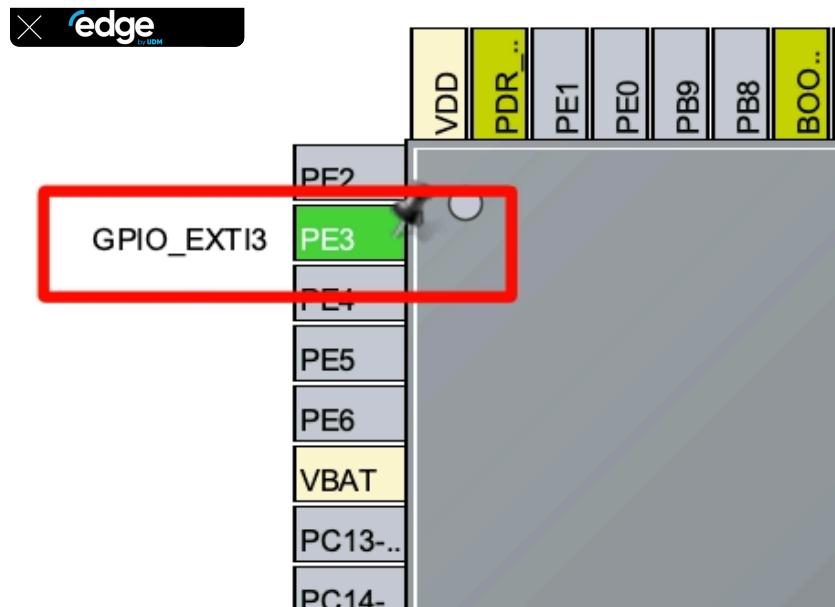
In Full duplex Mode, SPI uses 3 pins, **MOSI**, **MISO** and **CLK**. We need to set one more pin as output so to be used as the Chip Select (**CS**) Pin.





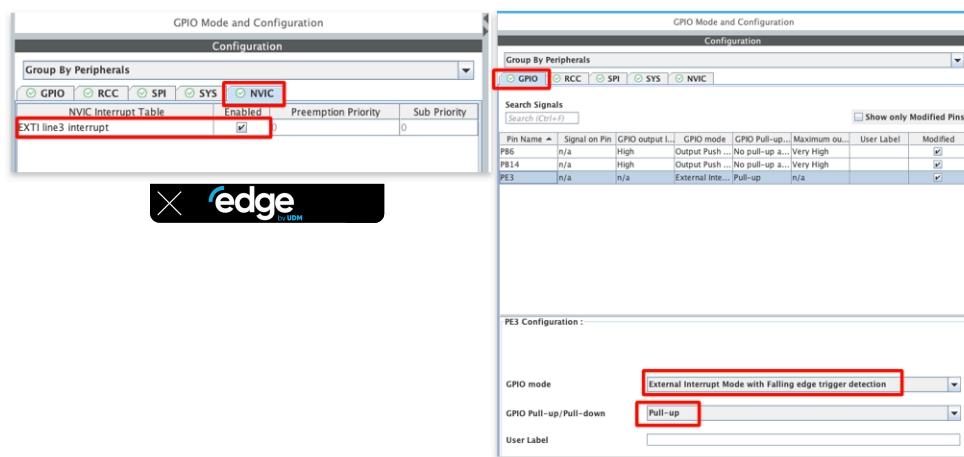
The Pin **PB14** is set as the CS pin. The initial output level is set to **HIGH**. This is because the pin needs to be pulled low in order to select the slave device, so we keep it high initially to make sure the slave device isn't selected. The Output speed is set to **Very High** because we might need to select and unselect the slave at higher rate.

We will also add a button to the project. Whenever the button is pressed, the data will be stored in the flash memory. The board I am using already has a user button, Its connection is shown below.



The pin PE3 is set as the EXTI (External Interrupt) pin.

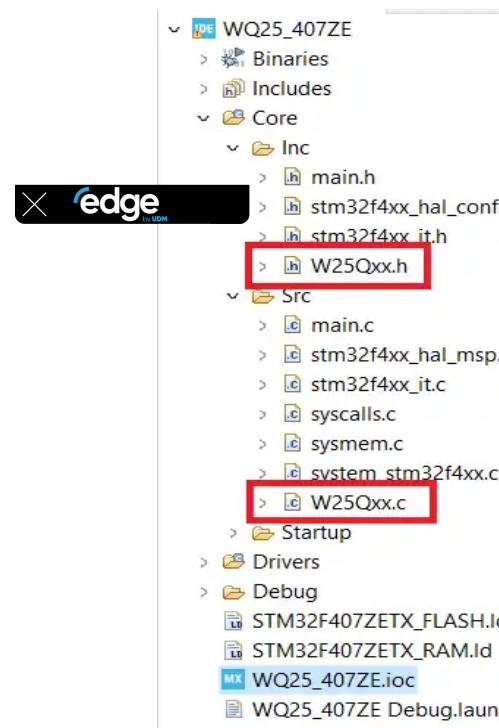
In the GPIO configuration, I have enabled the Interrupt for the EXTI line 3.



The GPIO mode is set to falling edge detection. Basically when the button is pressed, the pin goes LOW. On the detection of the falling edge, the interrupt will be triggered.

We created separate library files for the device. Today we will simply update the code in these files itself.

The files are: **W25Qxx.c** file in the **src** directory and **W25Qxx.h** file in the **Inc** directory.



Write/Read Integers and Floats



Since the flash can only store data in bytes, we can't directly store the numbers into it. We need to first convert the numbers to the bytes using the union and store these bytes into the memory.

After reading the byte data from the memory, they are converted back to the numbers using union again.

We will first see the enums used for converting **float to bytes** and **bytes to float**. I am using float as it can be used to store both the integers and the floats itself.

The conversion functions

```
void float2Bytes(uint8_t * ftoa_bytes_temp, float float_variable)
{
    union {
        float a;
        uint8_t bytes[4];
    } thing;
    thing.a = float_variable;
```



```
for (uint8_t i = 0; i < 4; i++) {  
    ftoa_bytes_temp[i] = thing.bytes[i];  
}
```

```
}
```



```
float Bytes2float(uint8_t * ftoa_bytes_temp)  
{  
    union {  
        float a;  
        uint8_t bytes[4];  
    } thing;  
  
    for (uint8_t i = 0; i < 4; i++) {
```

Now we have the functions to convert the data, let's write another function to write the float into the memory.

The write function

```
uint8_t tempBytes[4];  
void W25Q_Write_NUM (uint32_t page, uint16_t offset, float data)  
{  
    float2Bytes(tempBytes, data);  
  
    /* Write using sector update function */  
    W25Q_Write(page, offset, 4, tempBytes);  
}
```



Each float variable takes 4 bytes in the memory, so we need to define an array of 4 bytes to store the converted float data.

The function takes the  the parameter, along with the **@page** and the **@offset** where this value needs to be stored.

Inside the function we will first convert the float value to the bytes.

Then using the Write function, we will store these 4 data bytes into the provided memory location.

Let's see how to read the float data back from the memory.

The read function

```
float W25Q_Read_NUM (uint32_t page, uint16_t offset)
{
    uint8_t rData[4];
    W25Q_Read(page, offset, 4, rData);
    return (Bytes2float(rData));
}
```

The function **W25Q_Read_NUM** takes the parameters as the **@page** and the **@offset** where the number needs to be read from. It **returns** the float value in the end.

Inside the function first we read the 4 bytes of data from the given page and offset.

This data is in bytes format and we convert it to the float using the function `Bytes2float`.

And finally the float value is returned.



To test these functions, we will define some values in the main file and then write them to different locations in the memory.

The main function

```
float tNum1 = 100.58;  
float rNum1 = 0;  
  
float tNum2 = 12345678;  
float rNum2 = 0;  
  
float tNum3 = -1209;  
float rNum3 = 0;
```

Here I have defined a float value, an integer value and a negative integer value. We will write all these 3 values in the memory and then read them.

```
if (isPressed == 1)  
{  
    W25Q_Write_NUM(0, 10, tNum1);  
    rNum1 = W25Q_Read_NUM(0, 10);
```



```
W25Q_Write_NUM(0, 12, tNum2);
rNum2 = W25Q_Read_NUM(0, 12);
```

```
W25Q_Write_NUM(0, 14, Num3);
rNum3 = W25Q_Read_NUM(0, 14);
```

```
    isPressed = 0;
}
```

When the button is pressed, we will write the values to different offsets on page 0. After writing each value, we will read it back to verify if the write was successful.

Note that each float occupies 4 bytes in the memory. So if you don't want to read the values back, make sure there is a gap of 4 bytes between two successive writes.

Let's see the output of the above code.



Expression	Type	Value
(x)=rNum1	float	100.580002
(x)=rNum2	float	12345678
(x)=rNum3	float	-1209

+ Add new expression

edge

```
float tNum1 = 100.58;
float rNum1 = 0;

float tNum2 = 12345678;
float rNum2 = 0;

float tNum3 = -1209;
float rNum3 = 0;
```

As you can see we got the same values we stored in the memory. So the writing and reading numbers works fine.

The video is at the end of this post.



Prefabrykowane domy na zgłoszenie do 70m - Cena może cię zaskoczyć

Prefabrykowane domy | Reklamy w wyszukiwarce

Write/Read 32 bit data

Just like numbers, writing 32 bit data in the memory is not directly permitted, but this does not mean that we can't write it. We can store a 32bit variable, by splitting it into the 4 bytes.

And to read the data back ~~we will read those 4 bytes, and then combine them to make a 32bit variable.~~



Let's start with the writing part first.

Write 32bit data

```
void W25Q_Write_32B (uint32_t page, uint16_t offset, uint32_t size, uint32_t *dat
{
    uint8_t data8[size*4];
    uint32_t indx = 0;

    for (uint32_t i=0; i<size; i++)
    {
        data8[indx++] = data[i]&0xFF;      // extract LSB
        data8[indx++] = (data[i]>>8)&0xFF;
        data8[indx++] = (data[i]>>16)&0xFF;
        data8[indx++] = (data[i]>>24)&0xFF;
    }

    W25Q_Write(page, offset, indx, data8);
}
```

The function takes the parameters as follows:

- **@page, @offset** is the memory location where the write will start from.
- **@size** is the number of elements in the 32bit array.
- **@data** is the pointer to the 32bit array that we want to write.

Since each 32bit variable occupies 4 bytes, we first create a buffer (**data8**), whose size is 4 times the parameter **@size**.

The **indx** variable will keep track of how many bytes has been stored in the **data8** array.

Then the for loop will run as many times as there are elements in the data array. For each run, we will split the 32 bit variable in 4 bytes and store them in the **data8** array.

Once all the elements has been converted to bytes, we will write the bytes array (**data8**) to the specified memory location.

The reading 32bit variable is similar to write function. Here we will read the data from the memory, and then combine the 4 bytes to make a 32bit variable.

Read 32bit variable

```
void W25Q_Read_32B (uint32_t page, uint16_t offset, uint32_t size, uint32_t *data
{
    uint8_t data8[size*4];
    uint32_t indx = 0;

    W25Q_FastRead(page, offset, size*4, data8);
```

```
for (uint32_t i=0; i<size; i++)  
{  
    data[i] = (data8[indx++]) | (data8[indx++]<<8) | (data8[indx++]<<16) | (d  
}  
}
```



The array to store the 32bit variable is passed in the parameter itself. The for loop runs as many times as the number of 32bit variables we have to read, and for each run we combine the 4 bytes to make a 32bit variable.

In the main function we will define a 32 bit array to write to the memory.



The main function

```
uint32_t tData32[4] = {0x12, 0x1234, 0x12345678, 0xABCD};  
uint32_t rData32[4];
```

I defined an array of 4 elements. It contains a byte, a 16bit variable, a 32bit variable and a 24bit variable. This is to test if the function is able to handle all sort of values or not.

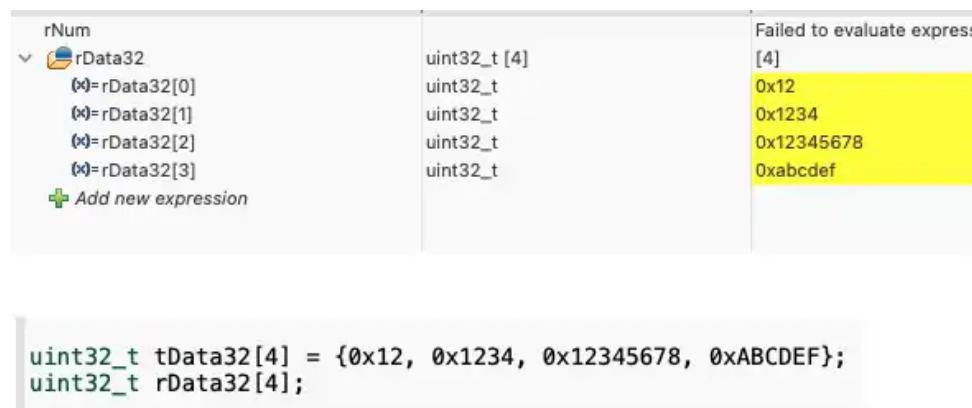
When the button is pressed, will write this array to the memory.



```
if (isPressed == 1)  
{
```

```
W25Q_Write_32B(0, 250, 4, tData32);  
W25Q_Read_32B(0, 250, 4, rData32);  
  
isPressed = 0;  
}  
X edge
```

Here I am writing the array to the page 0 at an offset of 250. Then we read from the same memory location. The output of the code is shown below.



As you can see in the image above, we get the same data in the output which we wrote to the memory. The function can handle all sort of data sizes.



Check out the Video Below

W25Q FLASH Memory || Part 6 || How to Write/Read Integers, floats and 32 b...



PART 5



X



Geekbuying



Proscenic P11
Smart Cordless
Vacuum Cleaner

€ 129.99

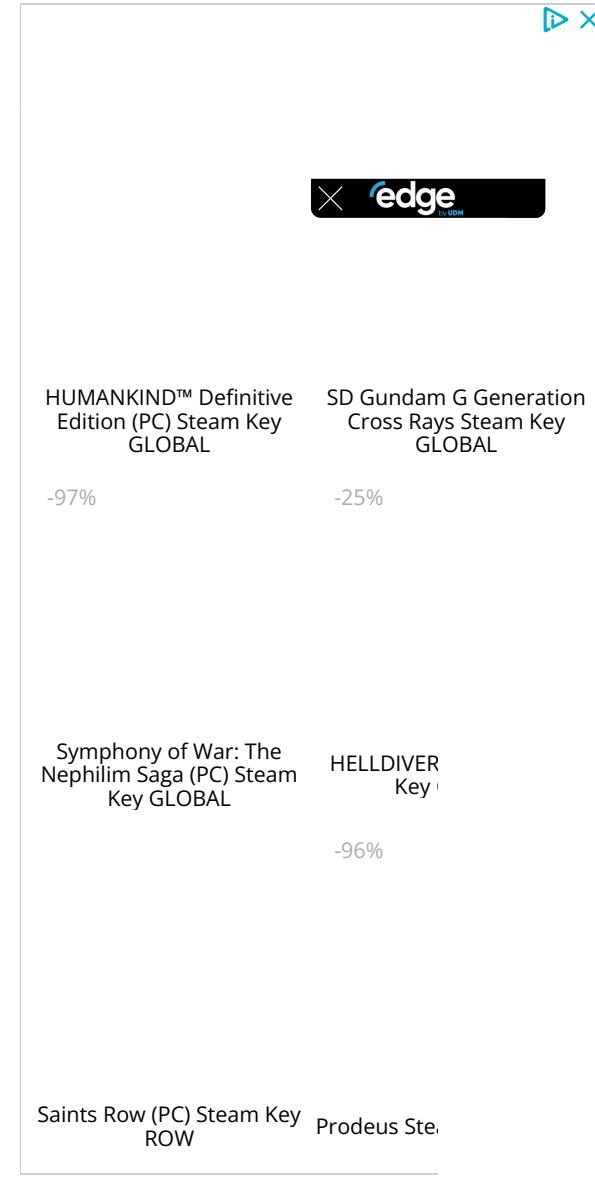
[Buy Now](#)



Wanbo T2 Max
NEW LCD
Projector - Blue

€ 145.00

[Buy Now](#)



X

DOWNLOAD SECTION



Info

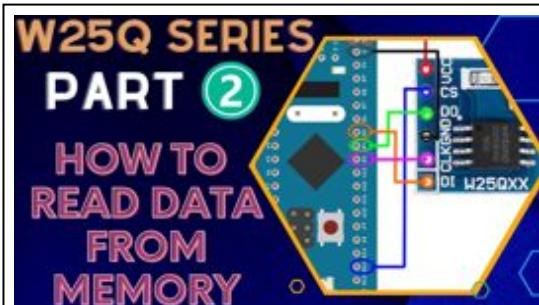
You can help with the development by DONATING

***To download the code, click DOWNLOAD button and view the Ad. The project
will download after the Ad is finished.***

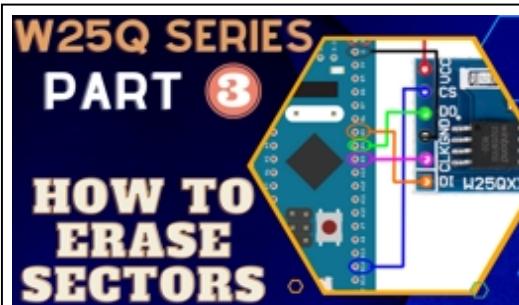


[**DOWNLOAD**](#)[**DONATE**](#)

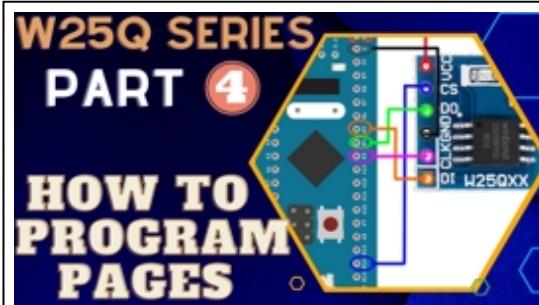
Related Posts:



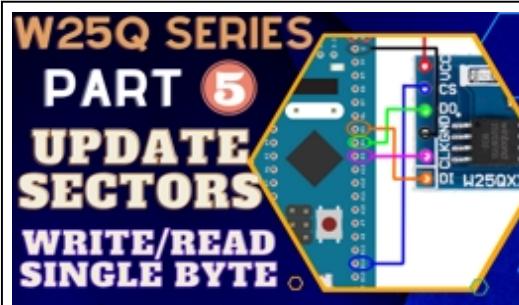
[W25Q Flash Series || Part 2 || Read Data from Device](#)



[W25Q Flash Series || Part 3 || How to Erase Sectors](#)



[W25Q Flash Series || Part 4 || How to Program Pages](#)

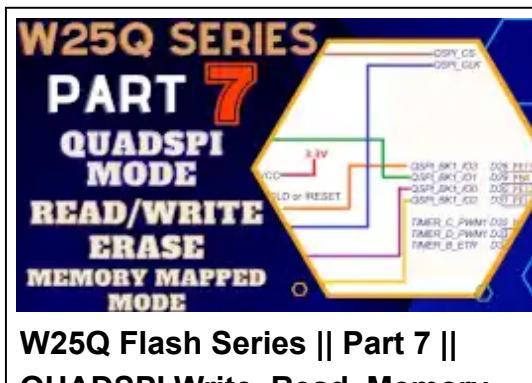


[W25Q Flash Series || Part 5 || how to update sectors](#)





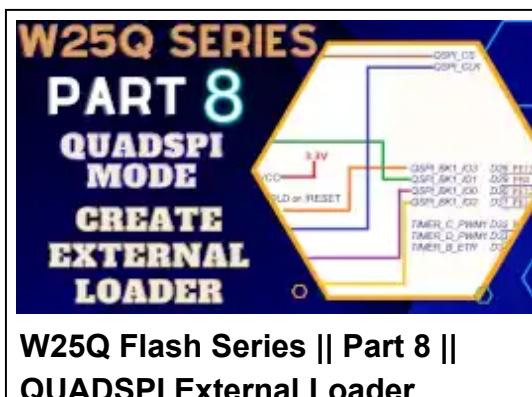
W25Q Flash Series || Part 1 || Read ID



W25Q Flash Series || Part 7 || QUADSPI Write, Read, Memory



WS2812 LEDs using SPI



W25Q Flash Series || Part 8 || QUADSPI External Loader

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *



Name * Email ***Post Comment**[Privacy Policy](#)