

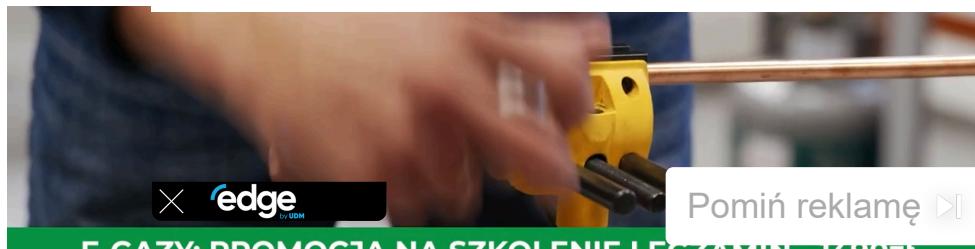


How to program pages in W25Q Flash

This is the 4th tutorial in the [W25Q Flash series](#), and today we will see how to program pages. We will store data into the memory using the page program instruction, and then read the data to confirm if the write has been successful.

The Page Program instruction allows from one byte to 256 bytes (a page) of data to be programmed at previously erased (FFh) memory locations. A Write Enable instruction must be executed before the device will accept the Page Program Instruction.





X edge
by UDN

Pomiń reklamę ▶

F-GAZY: PROMOCJA NA SZKOLENIE I EGZAMIN - 1490 zł

WWW: f-gazy.com.pl / Zadzwoń - +48 660 900 900

The page program instruction (0x02) should be followed by the 24 bit memory address, which should be followed by the data to be stored.

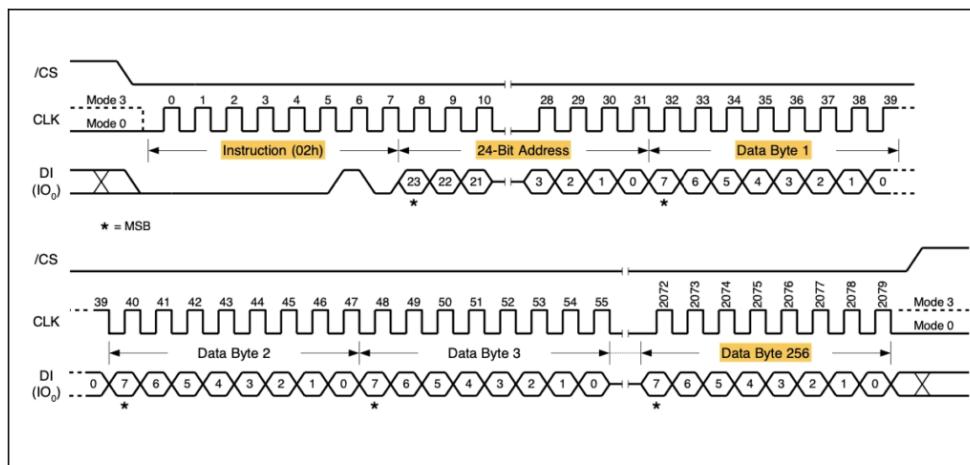


Figure 29. Page Program Instruction

Unlike the Page Read instruction, the page program instruction can not program multiple pages at once. In fact we can only program maximum 1 page at a time. To program another page, we have to send the instruction followed by the memory address again.

We will find another way to program multiple pages together. Let's see the connection first.

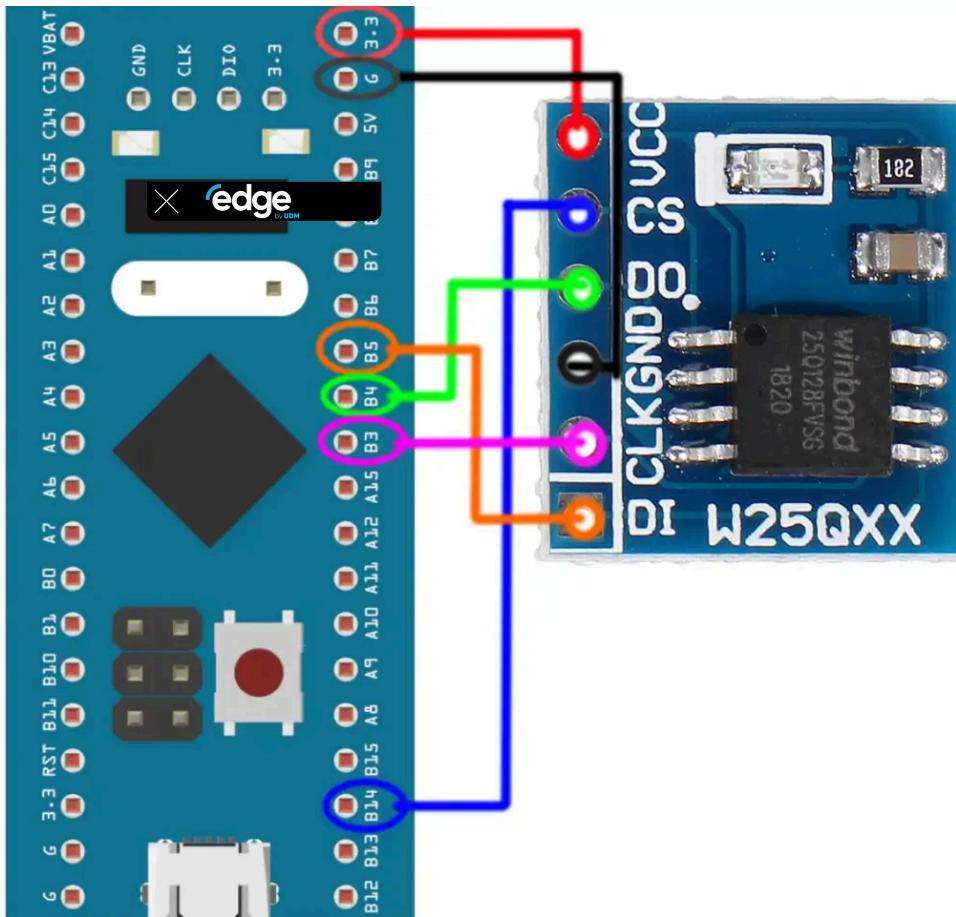
Connection



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.



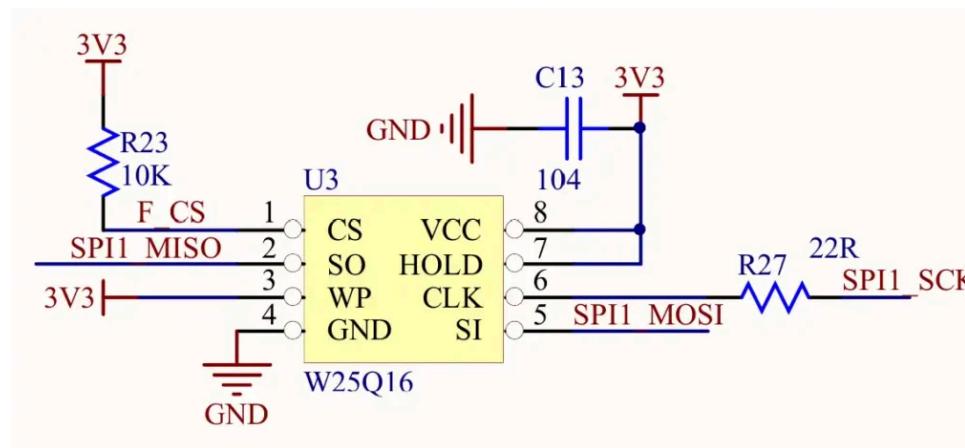


- The **CS** (Chip Select) Pin is connected to the pin PB14. It will be used to select or unselect the slave Device.
- The **DO** (Data Out) pin is connected to the pin PB4 (**MISO**). The device outputs the data on this pin
- The **CLK** (Clock) pin is connected to the pin PB3 (**CLK**). The clock is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

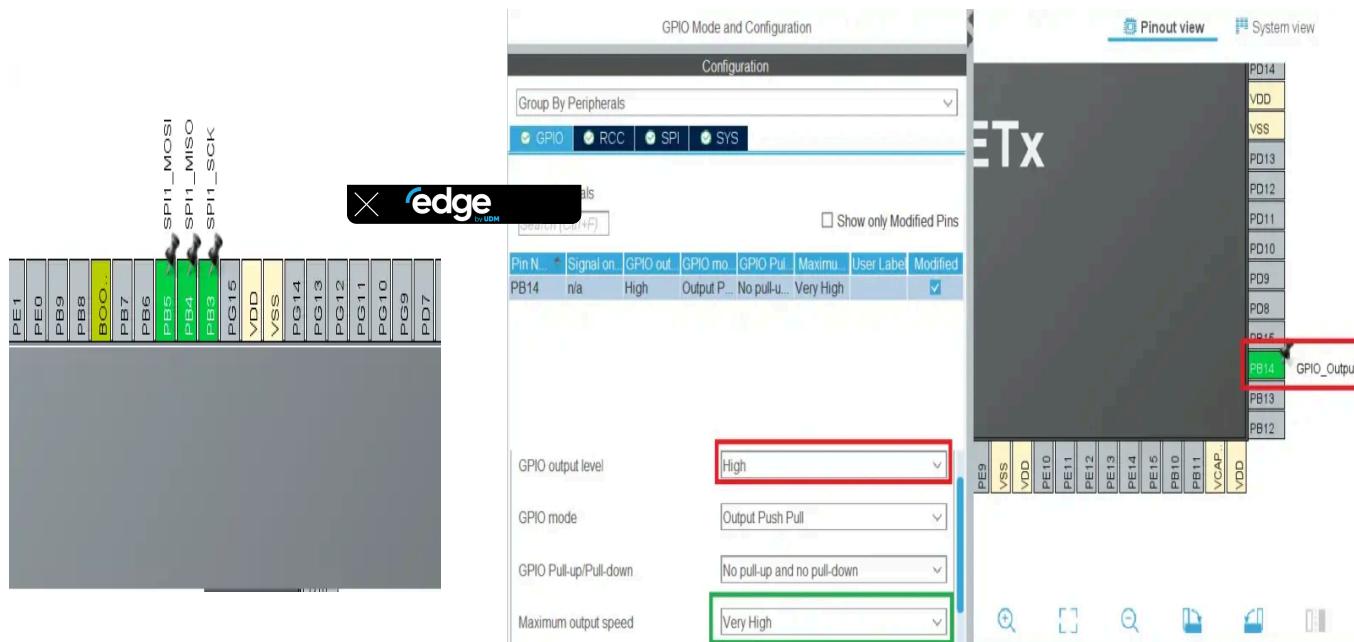
The W25Q16JV is accessed through an SPI compatible bus consisting of four signals: Serial Clock (CLK), Chip Select (/CS), Serial Data Input (DI) and Serial Data Output (DO). Standard SPI instructions use the DI input pin to serially write instructions, addresses or data to the device on the rising edge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

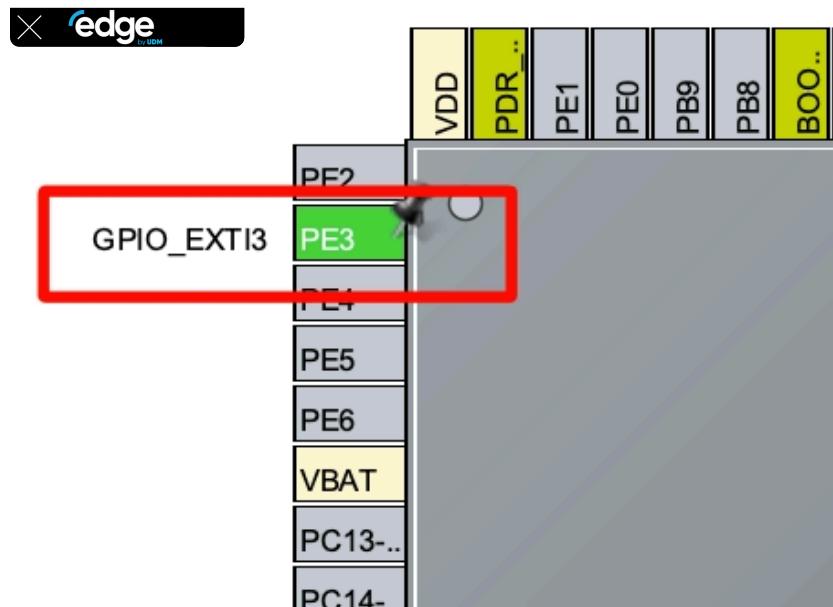
In Full duplex Mode, SPI uses 3 pins, **MOSI**, **MISO** and **CLK**. We need to set one more pin as output so to be used as the Chip Select (**CS**) Pin.





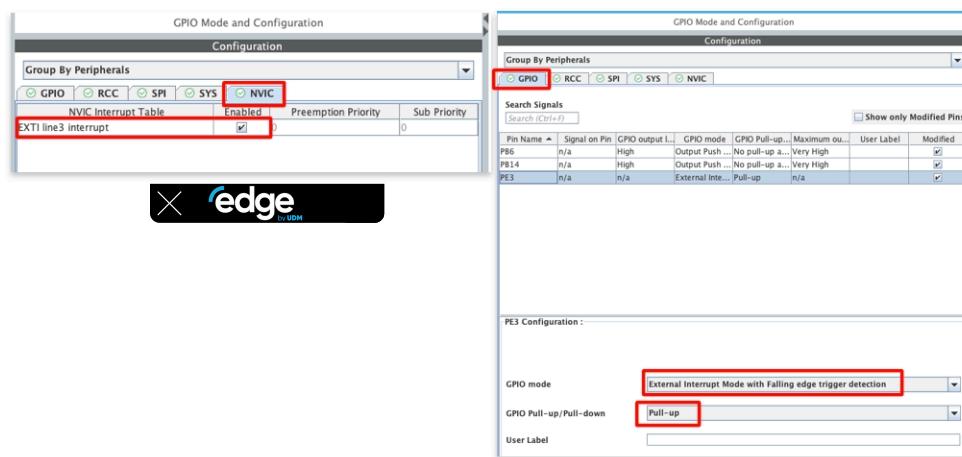
The Pin **PB14** is set as the CS pin. The initial output level is set to **HIGH**. This is because the pin needs to be pulled low in order to select the slave device, so we keep it high initially to make sure the slave device isn't selected. The Output speed is set to **Very High** because we might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

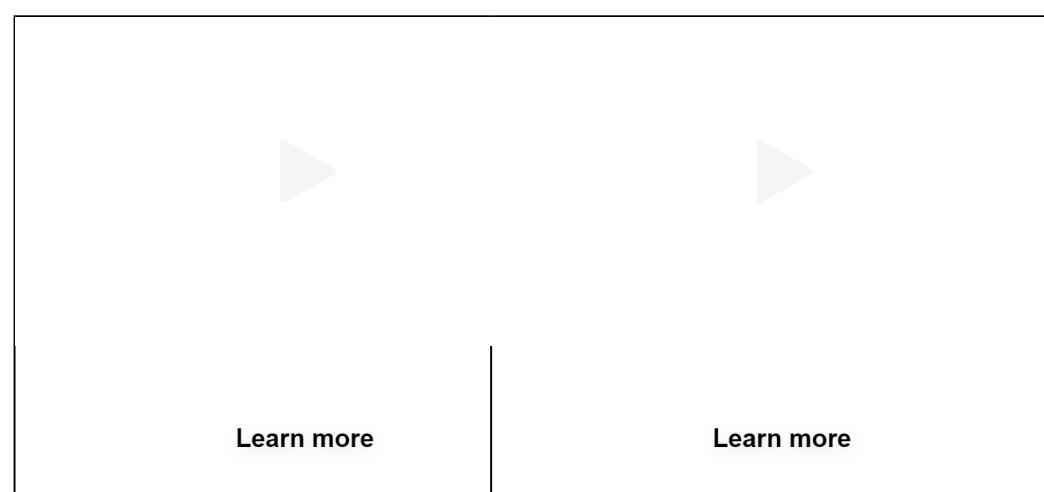
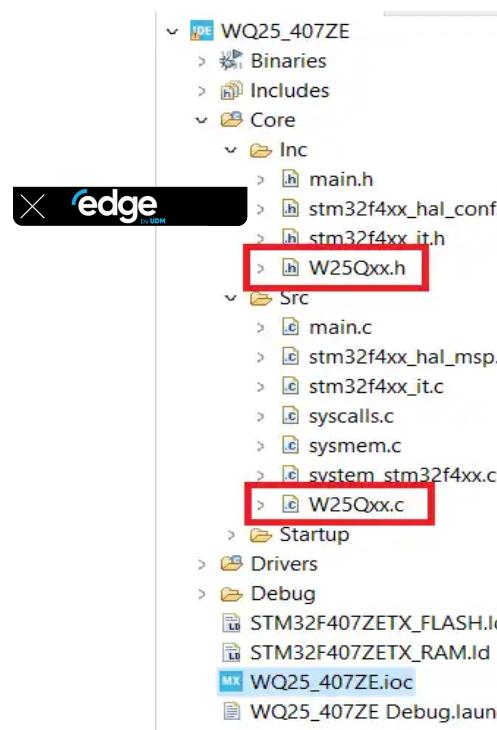
In the GPIO configuration, I have enabled the Interrupt for the EXTI line 3.



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.



Let's Write the Code



I have already covered the functions to **enable the write** and to **erase the sectors** in the previous tutorial, so I will only focus on the page program part in this tutorial.

Page Write Function

```
void W25Q_Write_Page (uint32_t page, uint16_t offset, uint32_t size, uint8_t *dat
{
    uint8_t tData[266];
    uint32_t startPage = page;
    uint32_t endPage  = startPage + ((size+offset-1)/256);
    uint32_t numPages = endPage-startPage+1;
```



The W25Q_Write_Page function takes the following parameters

- **@page** is the start page, where the write will start from.
- **@offset** is the offset on the first page. This can vary between 0 to 255.
- **@size** is the size of data we want to write.



- **@data** is the pointer to the data that we want to write.

We will first define an array of 266 bytes. This is to accommodate the **instruction**, the **memory address**, and  256 bytes of **data**.

- Then assign the value to the **startPage**, which is same as the page variable. This is the page number, where we will start writing the data from.
- Next calculate the **endPage**. This is the page number, where the data writing will end.
- Based on startPage and endPage values, calculate the number of pages (**numPages**) to be written.

The data can be only written to the location which were previously erased (have the value 0xFF). So to make sure that the memory does not have any previous data in those specific locations, we will erase the sectors.

```
uint16_t startSector = startPage/16;
uint16_t endSector = endPage/16;
uint16_t numSectors = endSector-startSector+1;
for (uint16_t i=0; i<numSectors; i++)
{
    W25Q_Erase_Sector(startSector++);
}
```



To erase the sector/sectors, we first need to calculate the **startSector** and **endSector**, where we are writing the data. Each sector contains 16 pages, so to calculate the startSector and endSector I am just dividing the pages by 16.



Then we calculate the number of sectors to be erased, and finally erase the sectors.

After erasing the sectors, we will start preparing for the write operation.

```
uint32_t dataPosition = 0;

// write the data
for (uint32_t i=0; i<numPages; i++)
{
    uint32_t memAddr = (startPage*256)+offset;
    uint16_t bytesremaining = bytestowrite(size, offset);
```

The variable **dataPosition** will keep track of the position of the data in the data pointer. We will update this variable after every page write, so that the offset in the data pointer also gets updated accordingly.

Since we can only write 1 page at a time, we have to repeat the loop as many times as the number of pages we are writing.



- Inside the loop, we first calculate the **memory address** of the page based on the page number and the offset.

- Next calculate the **bytesremaining**. This is equal to the number of data bytes we can send to the current page. The **bytestowrite** function is explained below.



```
uint32_t bytestowrite (uint32_t size, uint16_t offset)
{
    if ((size+offset)<256) return size;
    else return 256-offset;
}
```

bytestowrite function returns the number of bytes available to be written onto the page. It takes the parameters **@size** and **@offset**.

We continue the write function and now we will prepare the data to be sent to the device. We will use different page program instructions depending on if the chip size is lesser than 256Mb or not. For the devices with size 256Mb or more, we will always use the 32bit memory address, and hence a different version of page program instruction will be used.

```
write_enable();
    uint32_t indx = 0;
if (numBLOCK<512) // Chip Size<256Mb
{
    tData[0] = 0x02; // page program
    tData[1] = (memAddr>>16)&0xFF; // MSB of the memory Address
    tData[2] = (memAddr>>8)&0xFF;
    tData[3] = (memAddr)&0xFF; // LSB of the memory Address
```

```
    idx = 4;  
}
```

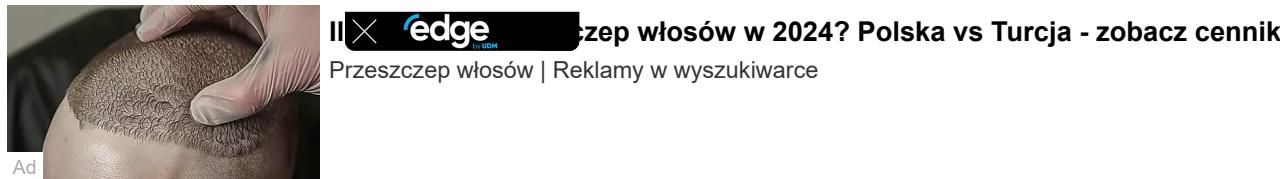
Before writing the data,  enable the write. The **idx** variable is used to keep track of the number of bytes in the **tData** buffer. The value of **idx** variable will change based on whether the memory size is 24bit or 32bit.

We will copy the page program instruction (0x02) and the 24 bit memory address in 3 separate bytes. And set the **idx** value to 4 indicating that 4 bytes of the **tData** buffer as been occupied.

```
else  
{  
    tData[0] = 0x12; // page program with 4-Byte Address  
    tData[1] = (memAddr>>24)&0xFF; // MSB of the memory Address  
    tData[2] = (memAddr>>16)&0xFF;  
    tData[3] = (memAddr>>8)&0xFF;  
    tData[4] = (memAddr)&0xFF; // LSB of the memory Address  
  
    idx = 5;  
}
```

If the device size is 256Mb or more, we can use the instruction page program with 4-Byte Address (**0x12**). This instruction will always require the 32bit memory size. 

The index value will be set to 5 indicating that 5 bytes of the tData buffer as been occupied.



Now we will copy the actual data in the tData buffer.

```
uint16_t bytestosend = bytesremaining + indx;  
  
for (uint16_t i=0; i<bytesremaining; i++)  
{  
    tData[indx++] = data[i+dataPosition];  
}
```

Here first we calculate how many bytes we have to send. The value of the bytestosend variable will be equal to the indx value + the bytesremaining.

We will copy the data starting from the indx position in the tData buffer. And the data position in the data pointer will be updated after writing every page. In the beginning it is set to 0 so that we start copying from the beginning of the data pointer.

Now the data has been copied, so we will send it to the device. The bytestosend variable is the actual data length we are sending.

```
csLOW();
SPI_Write(tData, bytestosend);
csHIGH();
```



After sending the data, we need to update the variables to prepare for the next transfer. As I mentioned before, we can only write a maximum of 1 page at a time, so we need to update variables as many times as the number of pages we are writing.

```
startPage++;
offset = 0;
size = size-bytesremaining;
dataPosition = dataPosition+bytesremaining;
```

Here the startPage will increment and the offset will be set to 0. This will make the next write to start from the beginning of the next page.

We have already copied the “**bytesremaining**” amount of data into the memory. So the **size** variable will be reduced by this amount and the **dataPosition** will increment by this amount. The dataPosition is the offset in the data pointer, so we have to increment to avoid copying the same data again.

After programming 1 page, we will wait for the write to finish.



```
W25Q_Delay(5);  
write_disable();  
  
}  
}
```



The 5ms is the delay for the page program to finish. As per the datasheet, the maximum time for the page program is 3ms, but I just added a little more time.

We will finally disable the write. It will be disabled automatically after the page program is finished, but this is just added for precaution.

The main function

```
uint8_t TxDATA[30];  
  
int indx = 0;  
  
int isPressed = 0;  
  
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)  
{
```



```
    isPressed = 1;  
}
```

The TxData array will be used to store the data to be sent to the memory. The idx variable will be used to update this data, so to demonstrate that the new data is being stored.

When the button will be pressed, the EXTI Callback will be triggered, and here we will set the variable **isPressed** to 1.

We will write the rest of the code in the while loop.

```
while (1)  
{  
    if (isPressed == 1)  
    {  
        sprintf (TxData, "Hello from W25Q -- %d", idx++);  
        W25Q_Write_Page(0, 250, strlen(TxData), TxData);  
        isPressed = 0;  
    }  
  
    W25Q_Read(0, 250, 20, RxData);  
  
    HAL_Delay(500);  
}
```

Here we will continuously read the memory location every 500ms.

If the button is pressed, the **isPressed** variable will be set to 1, and then only the write operation will be performed.

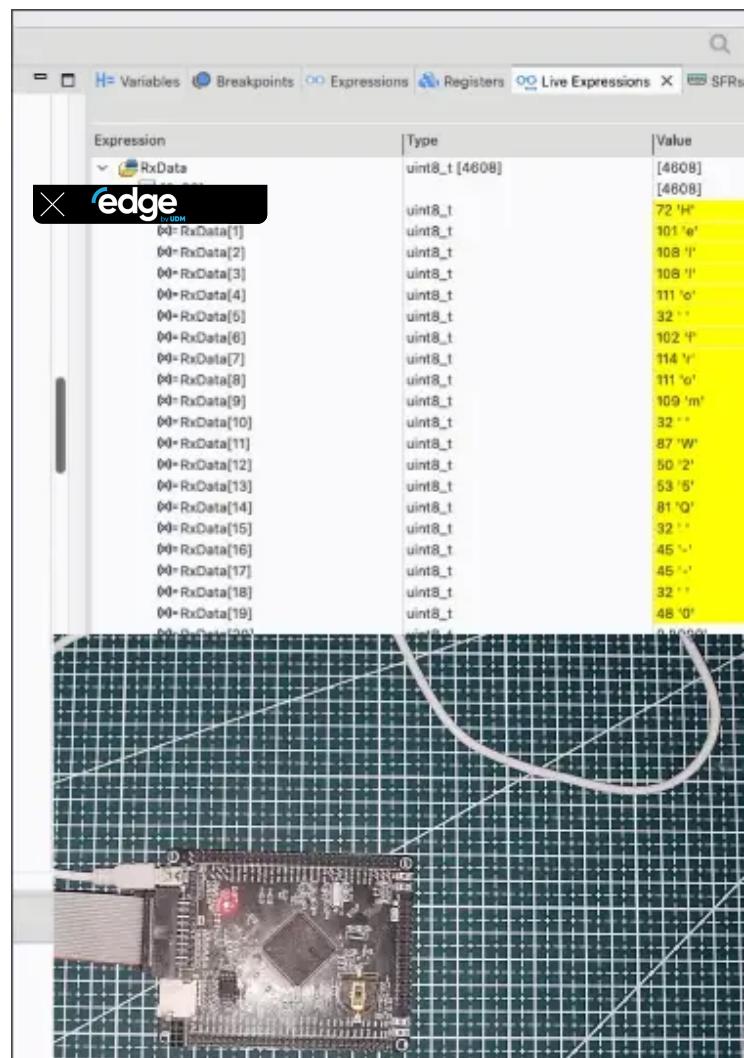
We will first copy the up  **TxData** array, and then write the data into the page. I am writing the data into the page 0 at an offset of 250. I chose this offset intentionally, just to demonstrate that write is successful across the page boundary.

The Read is also being performed from the same location, so we should see the data we sent in the **RxData** buffer.

Result

Below is the gif showing the working.





As shown in the gif above, when the button is pressed a new data is updated in the live expression. Actually this is the data that we read from the memory, so the update data must have been stored.



 Check out the Video Below

W25Q FLASH Memory || Part 4 || How to Program Pages



PART 3

PART 5



Informacja  X



Wypróbuj Pracuj.pl za 1 z

Zobacz teraz

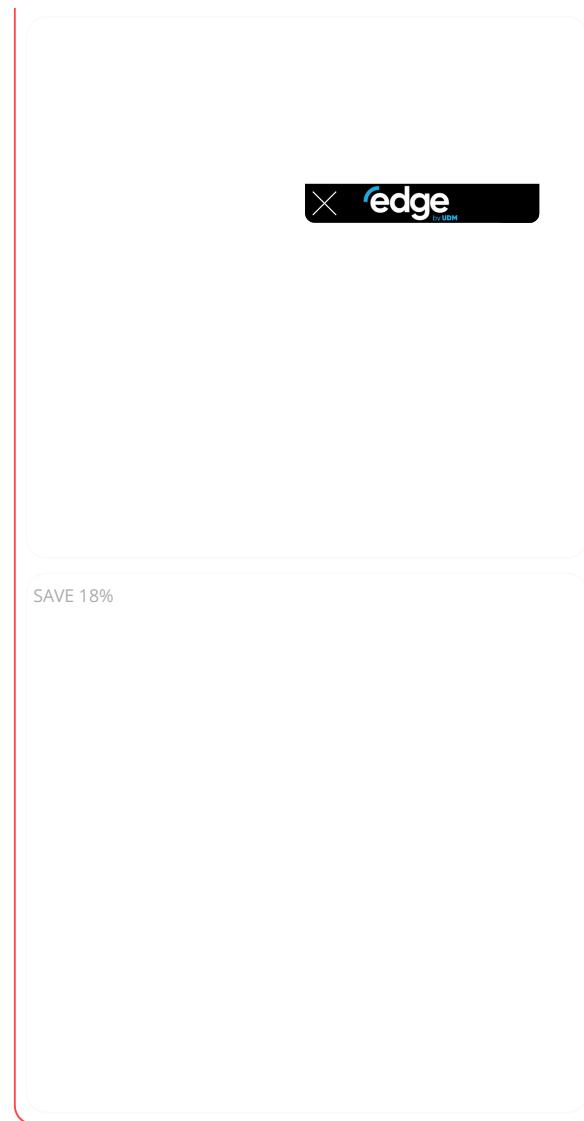


Learn more



An advertisement for mBank. At the top, there's a horizontal bar with colored segments (red, orange, yellow, green) and the word "zob" in white. Below this is the mBank logo, which consists of three squares in red, orange, and green with the word "mBank" in white. The main text reads "weź kredyt gotówkowy!" in bold black font, followed by "z kodem BIEG 0% prowizji za udzielenie (RRSO 13,23%)". To the right is a small image of a person's legs running on a track. At the bottom left is a red button with the word "sprawdź" in white. In the bottom right corner of the ad area is a black square with a white "X" icon.





X

DOWNLOAD SECTION



Info

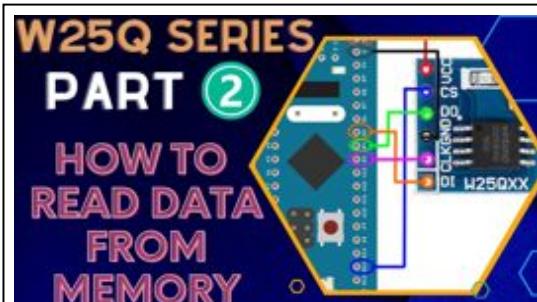
You can help with the development by DONATING

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

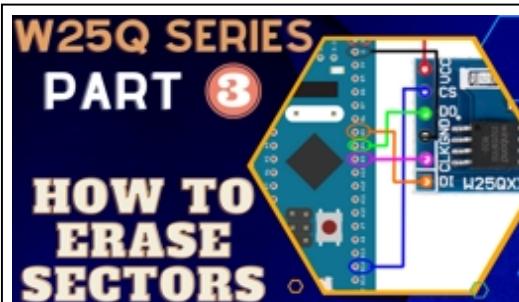


[**DOWNLOAD**](#)[**DONATE**](#)

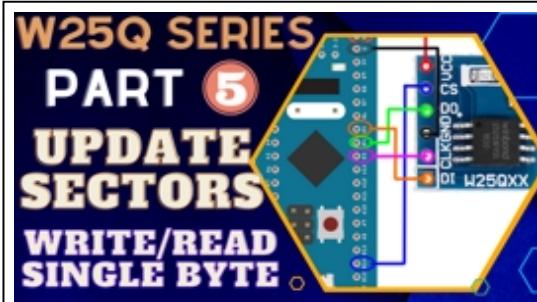
Related Posts:



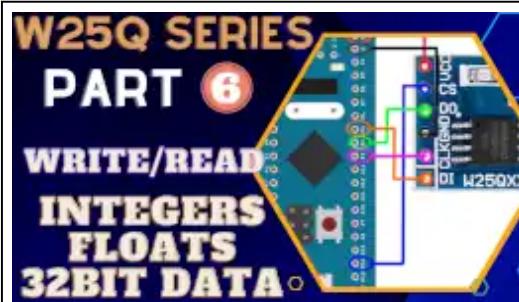
[W25Q Flash Series || Part 2 || Read Data from Device](#)



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

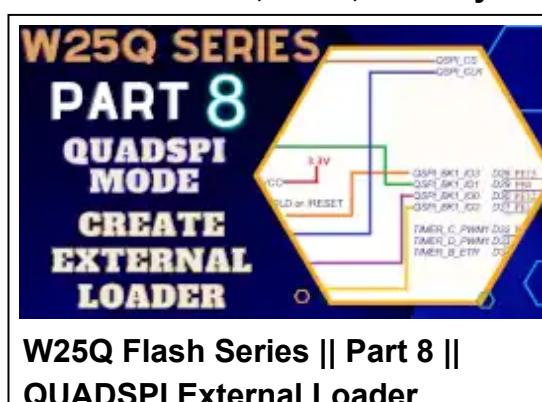
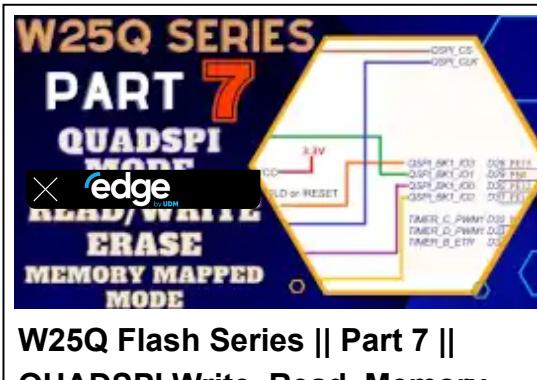


[W25Q Flash Series || Part 5 || how to update sectors](#)



[W25Q Flash Series || Part 6 || Integers floats and 32bit Data](#)





1 Comment. Leave new

Hieule

August 21, 2023 7:40 PM

Thank you. It's very help full for me

Reply



Leave a Reply

Your email address will  be d. Required fields are marked *

Comment *

Name *

Email *

Post Comment

[Privacy Policy](#)

