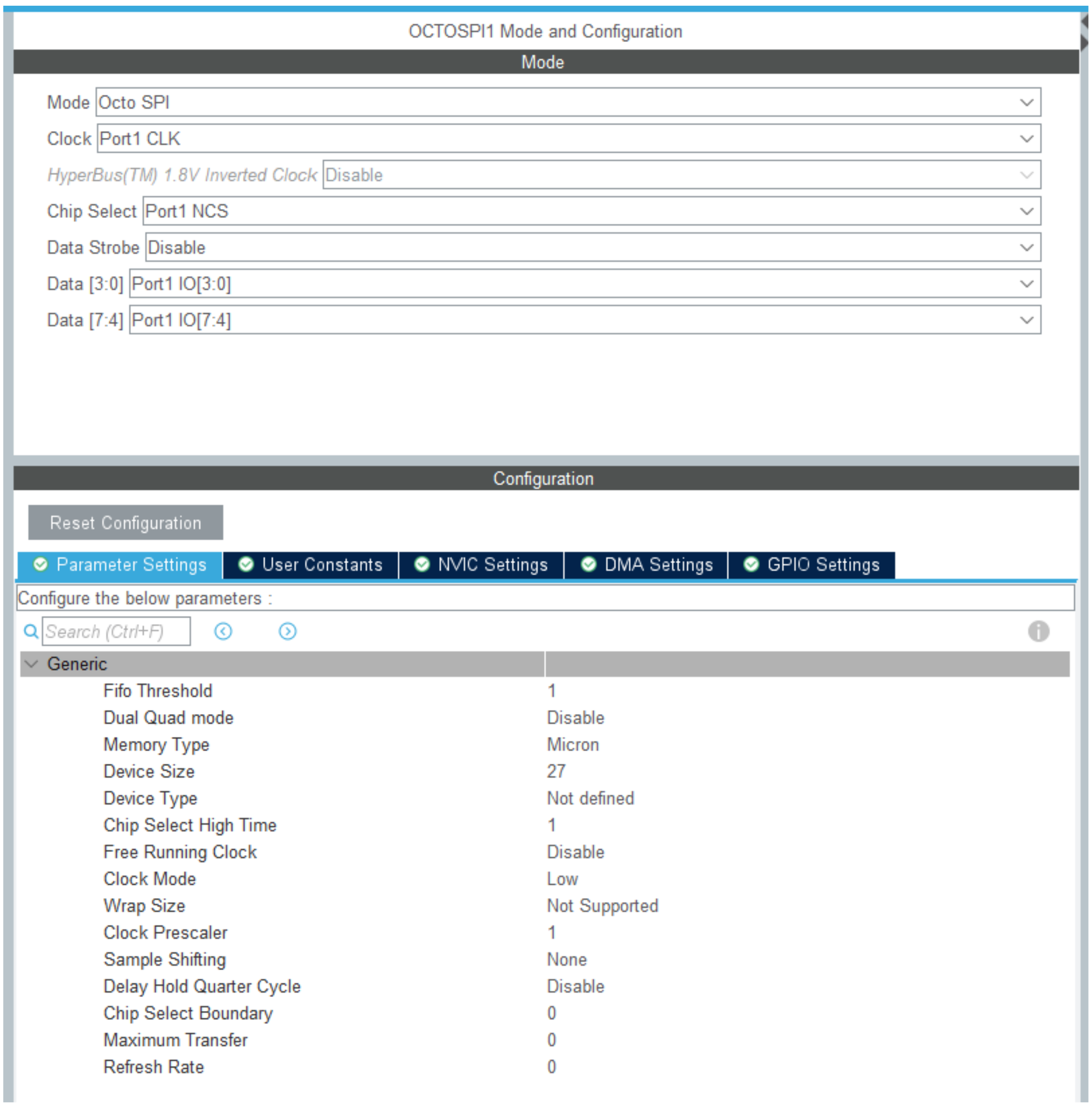
STM32 MRAM driver User guide

This user guide was made to show users how to properly configure and use the Everspin MRAM driver for STM32 Microcontrollers. The driver was tested on STM32U599NJ Microcontroller, using STM32Cube IDE v1.15.0, HAL drivers and OCTOSPI peripheral.

STM32 Cube MX Setup

OCTOSPI

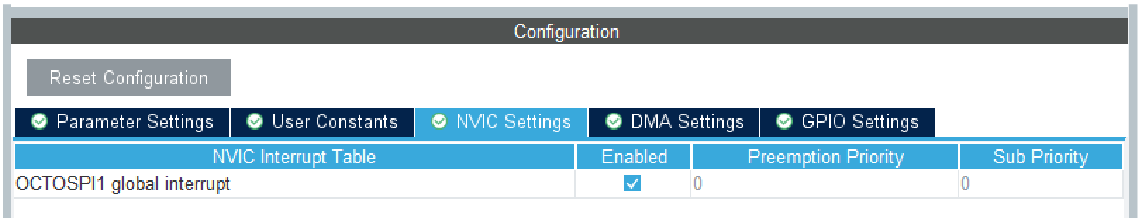
Start by selecting the desired OCTOSPI peripheral. Configure it to the desired mode and enable the clock, chip select and data pins. Select the device size appropriate to your device.



Mode	Octo SPI
Clock	Port1 CLK
HyperBus(TM) 1.8V Inverted Clock	Disable
Chip Select	Port1 NCS
Data Strobe	Disable
Data [3:0]	Port1 IO[3:0]
Data [7:4]	Port1 IO[7:4]

Generic	
Fifo Threshold	1
Dual Quad mode	Disable
Memory Type	Micron
Device Size	27
Device Type	Not defined
Chip Select High Time	1
Free Running Clock	Disable
Clock Mode	Low
Wrap Size	Not Supported
Clock Prescaler	1
Sample Shifting	None
Delay Hold Quarter Cycle	Disable
Chip Select Boundary	0
Maximum Transfer	0
Refresh Rate	0

On the NVIC tab, enable the referred OCTOSPI interrupt.



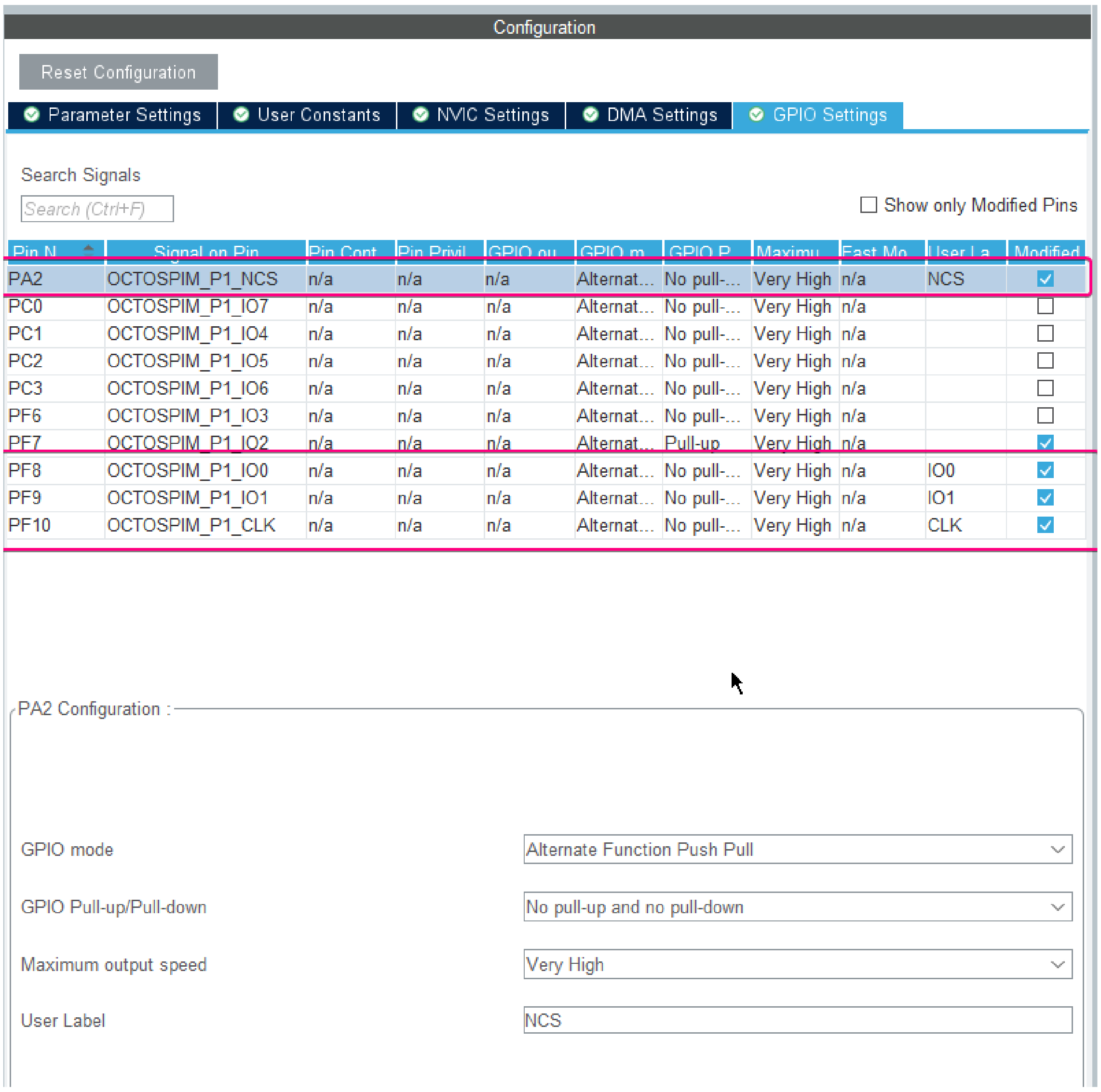
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
OCTOSPI1 global interrupt	✓	0	0

On the GPIO tab, give the following labels to the following pins:
OCTOSPIM_Px_NCS -> NCS

OCTOSPIM_Px_IO0 -> IO0

OCTOSPIM_Px_IO1 -> IO1

OCTOSPIM_Px_CLK -> CLK



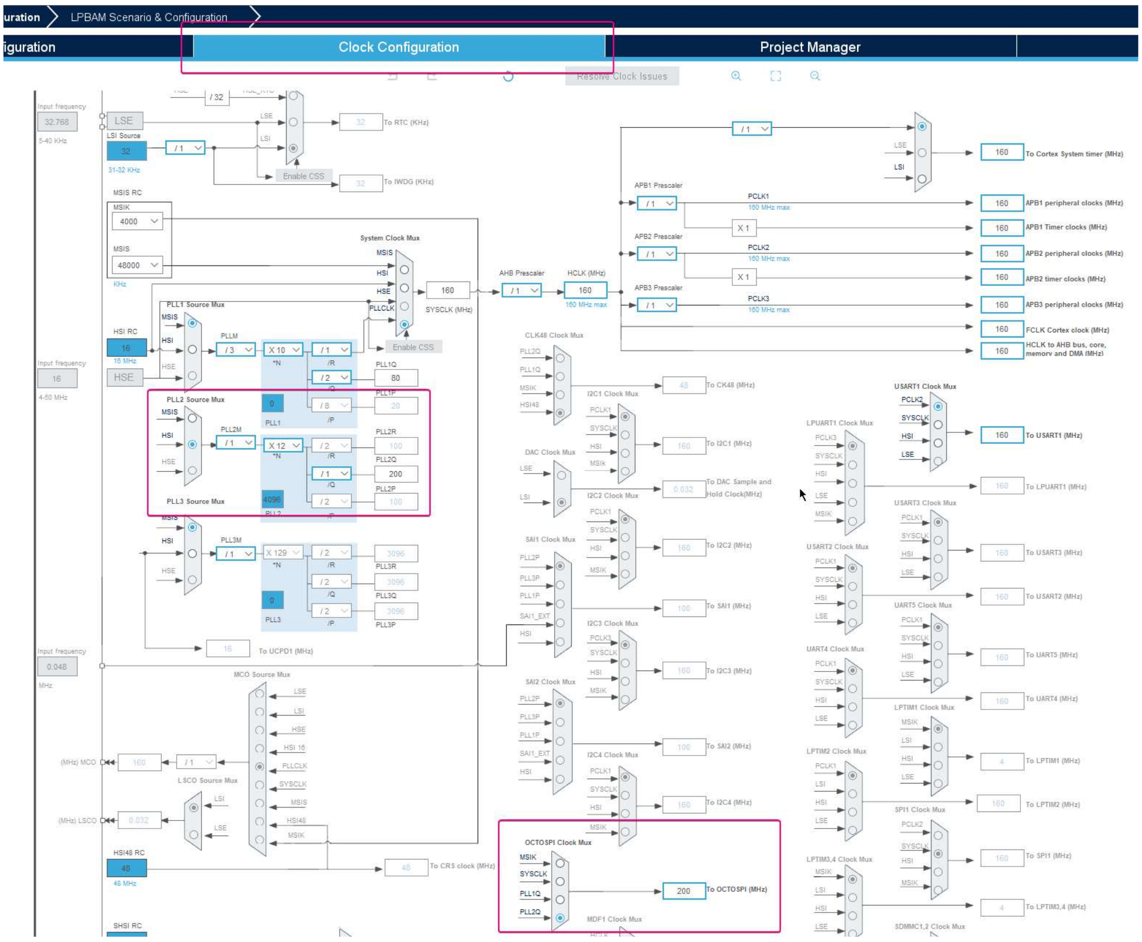
Pin N	Signal on Pin	Pin Cont	Pin Privil	GPIO ou	GPIO m	GPIO P	Maximu	Fast Mo	User La	Modified
PA2	OCTOSPIM_P1_NCS	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a	NCS	✓
PC0	OCTOSPIM_P1_IO7	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a		☐
PC1	OCTOSPIM_P1_IO4	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a		☐
PC2	OCTOSPIM_P1_IO5	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a		☐
PC3	OCTOSPIM_P1_IO6	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a		☐
PF6	OCTOSPIM_P1_IO3	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a		☐
PF7	OCTOSPIM_P1_IO2	n/a	n/a	n/a	Alternat...	Pull-up	Very High	n/a		✓
PF8	OCTOSPIM_P1_IO0	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a	IO0	✓
PF9	OCTOSPIM_P1_IO1	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a	IO1	✓
PF10	OCTOSPIM_P1_CLK	n/a	n/a	n/a	Alternat...	No pull-...	Very High	n/a	CLK	✓

PA2 Configuration :

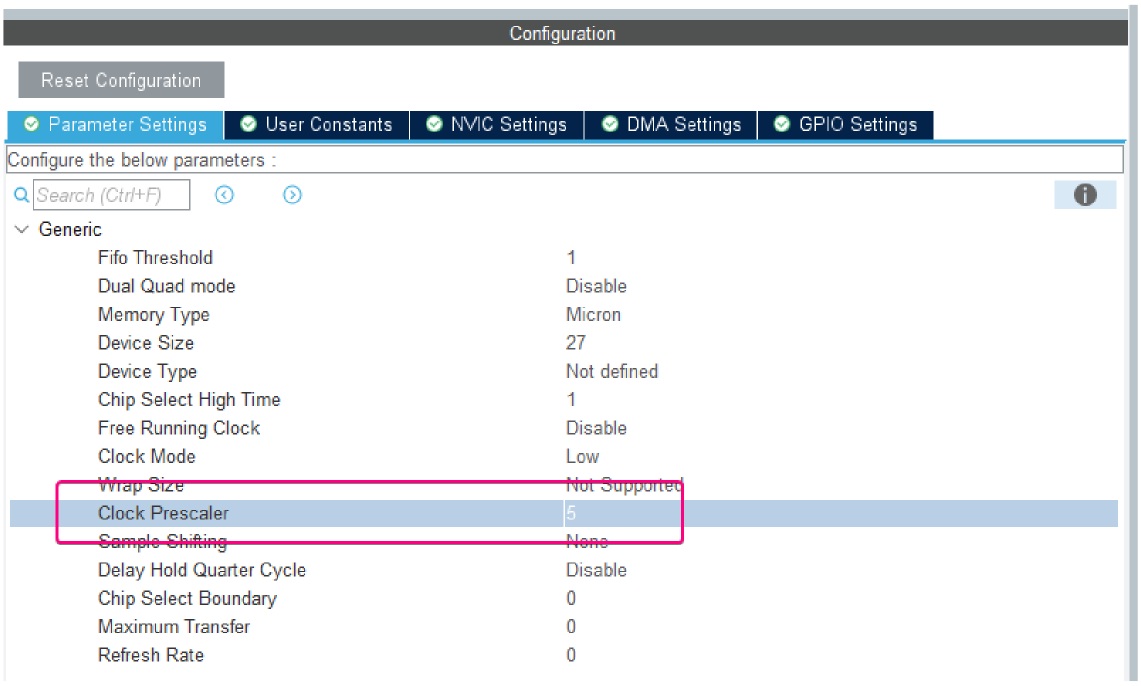
GPIO mode	Alternate Function Push Pull
GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	Very High
User Label	NCS

Clock Configuration

On the Clock Configuration tab, choose a clock source and value for the device.



The tested maximum frequency when setting up the device configuration is 40MHz. Go back to the OCTOSPI peripheral configuration and setup a prescaler value that makes fCLK <= 40MHZ



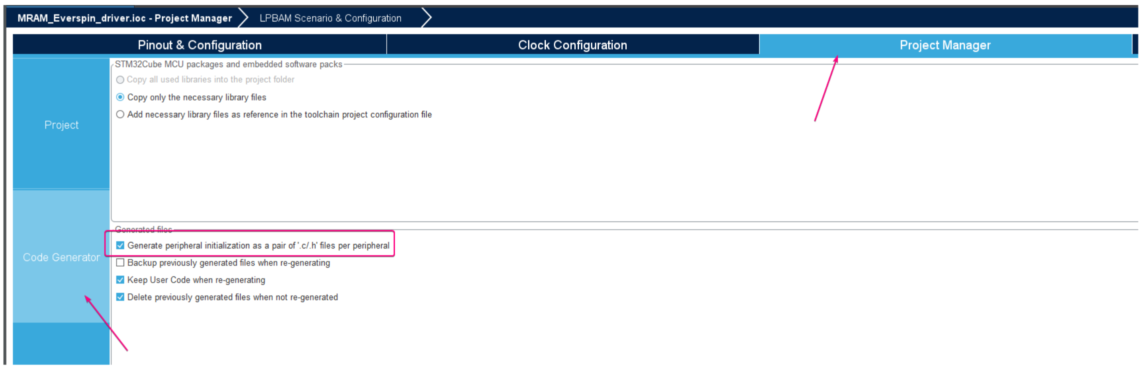
Configure the below parameters :

Generic

Fifo Threshold	1
Dual Quad mode	Disable
Memory Type	Micron
Device Size	27
Device Type	Not defined
Chip Select High Time	1
Free Running Clock	Disable
Clock Mode	Low
Wrap Size	Not Supported
Clock Prescaler	5
Sample Shifting	None
Delay Hold Quarter Cycle	Disable
Chip Select Boundary	0
Maximum Transfer	0
Refresh Rate	0

Project configuration

On the project manager tab, go to Code generator and tick the “Generate peripheral initialization as a pair of ‘.c/.h’ files per peripheral”.



STM32Cube MCU packages and embedded software packs

- ○ Copy all used libraries into the project folder
- ● Copy only the necessary library files
- ○ Add necessary library files as reference in the toolchain project configuration file

Generated files

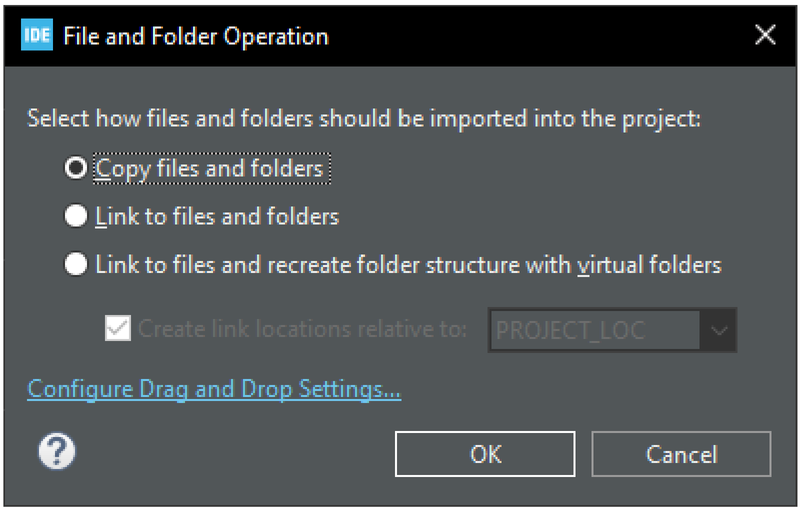
- ✓ Generate peripheral initialization as a pair of '.c/.h' files per peripheral
- ☐ Backup previously generated files when re-generating
- ✓ Keep User Code when re-generating
- ✓ Delete previously generated files when not re-generated

After this, the microcontroller is properly configured to receive the driver.

STM32 CubeIDE Project configuration

Importing

To import the driver into a project, drag and drop it from the file system to the project. On the pop-up that appears you can choose either if you want to have the driver to modify it, or only link to it (not making any modifications).



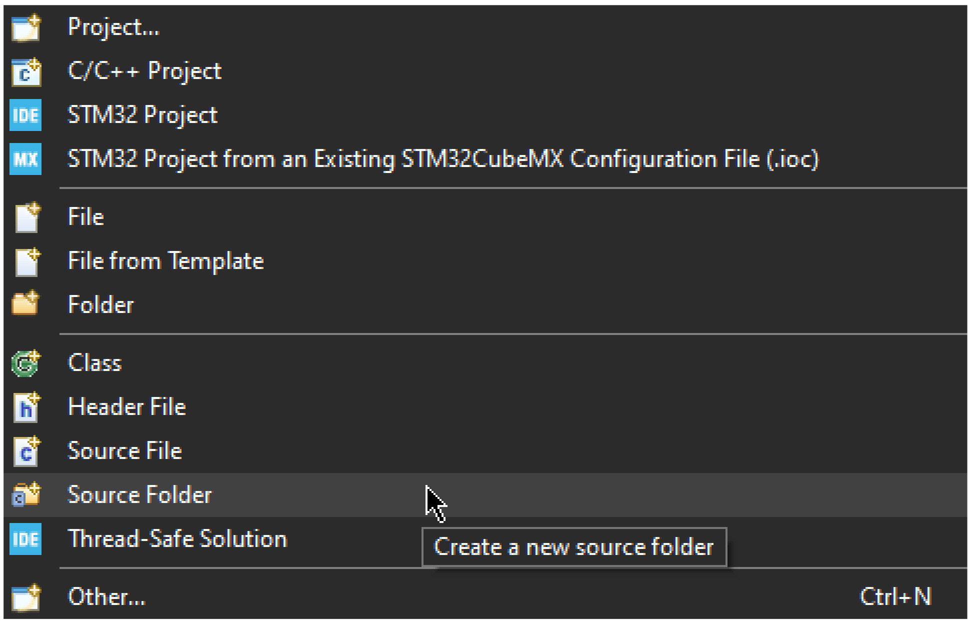
Select how files and folders should be imported into the project:

- ● Copy files and folders
- ○ Link to files and folders
- ○ Link to files and recreate folder structure with virtual folders

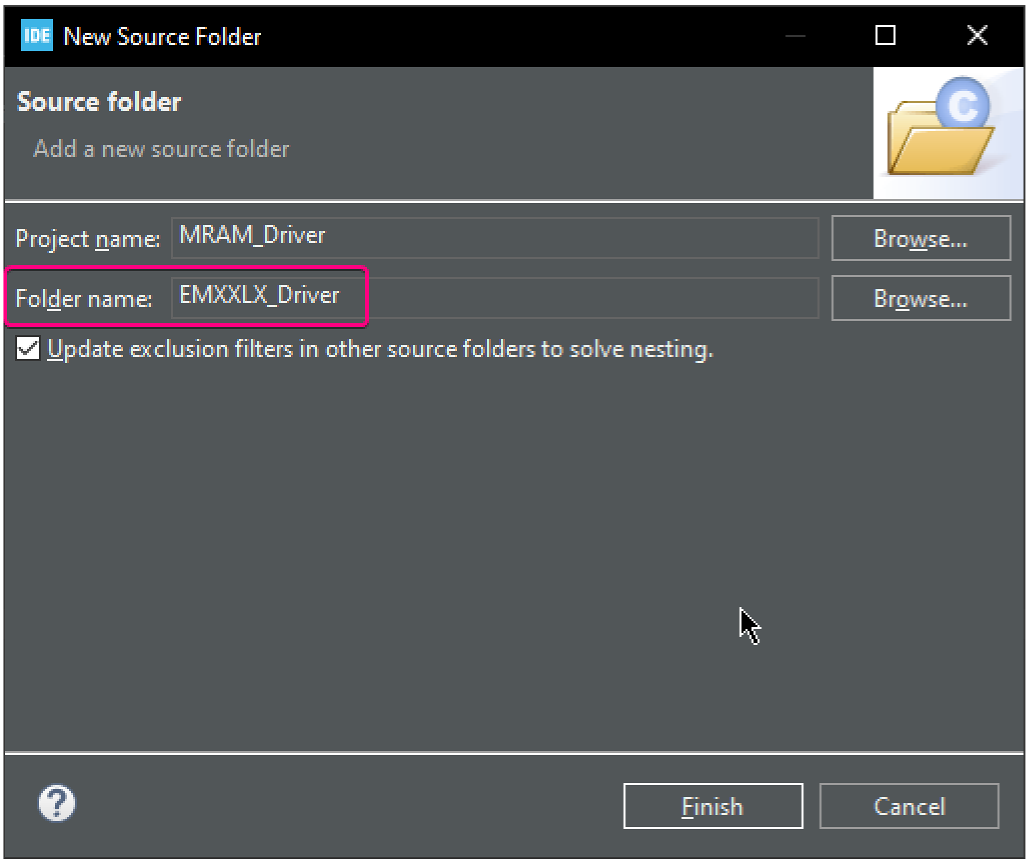
✓ Create link locations relative to: PROJECT_LOC

Configure Drag and Drop Settings...

Left click the project and press new and source folder



Write the exact same folder name into the folder name space in the pop-up window

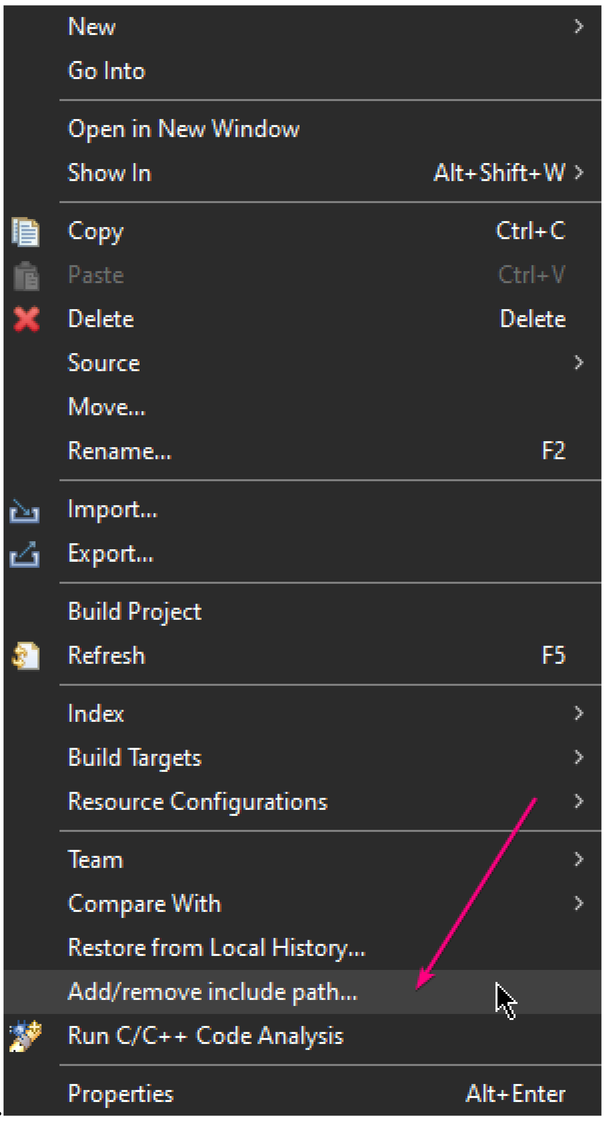


Project name: MRAM_Driver

Folder name: EMXXLX_Driver

☑ Update exclusion filters in other source folders to solve nesting.

Next, right click the newly imported and now converted to source folder and add it to the



include path.

After this, the driver is imported and ready to be used.

Coding

Start by including the mram.h header file

```
/* USER CODE BEGIN Includes */
#include "mram.h"
/* USER CODE END Includes */
```

Create a structure based on the EMXXLX_ConfigurationTypeDef Type and initialize all its values based on your preferences. CTRL+ Left mouse click the structure or opening the main.h file will show all the structure parameters. All required values are defined in the main.h.

Initialization code example

uint8_t ID[3];

EMXXLX_ConfigurationTypeDef MemConfig = { 0 };

```
MemConfig.SpiInterfaceMode = MRAM_OSPI_W_DS;
MemConfig.DummyCycles = MRAM_DEFAULT_DC;
MemConfig.DriverStrenght = MRAM_50_DRIVER_STR;
MemConfig.AddedDsDelay = MRAM_0_ADDED_DELAY;
MemConfig.AddressMode = MRAM_ADDRESS_BYTES_4;
MemConfig.XIPConfiguration = MRAM_XIP_DISABLE;
MemConfig.WrapConfiguration = MRAM_CONTINUOUS_WRAP;
MemConfig.EraseBitValue = MRAM_ERASE_VALUE_1;
MemConfig.ResetPinEnable = MRAM_RESET_ENABLE;
MemConfig.WriteMode = MRAM_NONVOLATILE;
MemConfig.OtpLockEnable = MRAM_OTPLOCK_ENABLE;
if (EMXXLX_Init(&hospi1, MemConfig, 8) != HAL_OK)
        Error_Handler();
EMXXLX_Read_ID(&hospi1, ID);
```

The code above configures the following aspects of the MRAM device, respectively:

OctoSPI interface with data strobe

Default amount of dummy cycles

50 ohms driver strength

No added delay

Address mode set to 32 bits or 4 bytes

Execute in place mode disabled

Warping configuration set to continuous

Erased bit value is 1

Reset pin enabled

Non-volatile memory mode

OTP lock is enabled.

EMXXLX_Init function takes the following arguments:

1st argument: the structure handling the peripheral tied to the device. **This is always the first argument of all functions in this driver.**

2nd argument: The structure containing all the settings chosen for the device in this application.

3rd argument: The number of data lines configured in the first space of the configuration structure, so if the structure configures quadSPI mode this number should be 4. The same logic applies to single and duo SPI modes.

The EMXXLX_Read_ID function takes the global 1st argument, while the second argument must be an array of 3 unsigned 8bit integers. This array will receive all 3 bytes forming the device's ID.

Memory mapped mode code example

Memory mapped mode is the most usual mode of operation for NOR devices, as it allows the core to access the external device as if it was internal flash, usually with added latency but with a huge density benefit. Graphics and AI applications rely on this mode to have access to large chunks of data that don't fit in the micro's internal memory.

Considering the initialization example was followed correctly, entering memory mapped mode should be as easy as flipping the write enable latch and calling the memory mapped function:

EMXXLX_Write_Enable(&hospi1);

EMXXLX_MemoryMapped_Config(&hospi1);

Both functions take only the mandatory handler argument. One thing to be noted is that memory mapped mode only works when memory addressing is set to 32 bits.

To test memory mapped mode, the following code is used:

```
        /*memory mapped variables */
uint8_t data[] =
                " ****Memory-mapped OSPI communication****   ****Memory-mapped OSPI communication****   ****Memory-mapped OSPI communication****   ****Memory-mapped OSPI communication****   ****Memory-mapped OSPI communication****   ****Memory-mapped OSPI communication****";
uint8_t read_data[sizeof(data) - 1];
uint32_t BUFFERSIZE = sizeof(data) - 1;


        __IO uint8_t *mem_addr;
        uint32_t address = 0;
        uint16_t index;


        /* Writing Sequence ----------------------------------------------- */
```

```
address = 0;
while(address <= (sizeof(data) * 32768)){
mem_addr = (uint8_t*) (OCTOSPI1_BASE + address);
for (index = 0; index < BUFFERSIZE; index++) {
        *mem_addr = data[index];
        mem_addr++;
}
address += sizeof(data);
}
/* Reading Sequence ----------------------------------------------- */
while(address <= (sizeof(data) * 24)){
mem_addr = (uint8_t*) (OCTOSPI1_BASE + address);
for (index = 0; index < BUFFERSIZE; index++) {
        read_data[index] = *mem_addr;
        mem_addr++;
}
address += sizeof(data);
}
```

To exit memory mapped mode, use the following HAL function

```
/* Abort OctoSPI driver to stop the memory-mapped mode ------------ */
if (HAL_OSPI_Abort(&hospi1) != HAL_OK) {
        Error_Handler();
}
```

Direct R/W mode

Direct access mode is used mostly when there's need to write into the external device, as you can use commands to pool for a finished write operation.

As the device is initialized, it is ready for direct access operation. The following code is an example of how to read/write data using this mode

```
uint8_t read_data[sizeof(data) - 1];
unsigned char data[] = "STM32 + MRAM";
 EMXXLX_Write_Enable(&hospi1);
```

```
  EMXXLX_Write(&hospi1, 0, data, sizeof(data));


  EMXXLX_Read(&hospi1, 0, read_data, sizeof(data));
  if (memcmp(payload, read_data, sizeof(payload))){
//Success case
  } else {
//Failure case
}
```