AN5050
Application note

Getting started with Octo-SPI, Hexadeca-SPI, and XSPI Interface on STM32 MCUs

Introduction

The growing demand for richer graphics, wider range of multimedia and other data-intensive content, drives embedded designers to enable more sophisticated features in embedded applications. These sophisticated features require higher data throughputs and extra demands on the often limited MCU on-chip memory.

External parallel memories have been widely used so far to provide higher data throughput and to extend the MCU on-chip memory, solving the memory size and the performance limitation. However, this action compromises the pin count and implies a need of more complex designs and higher cost.

To meet these requirements, STMicroelectronics offers several MCU products in the market with the new integrated high-throughput Octo/Hexadeca/XSPI interface (see the table below).

The Octo/Hexadeca/XSPI interface enables the connection of the external compact-footprint Octo-SPI/16-bit and the HyperBus™/regular protocol high-speed volatile and non-volatile memories available today in the market. Thanks to its low-pin count, the Octo/Hexadeca/XSPI interface allows easier PCB designs and lower costs. Its high throughput allows in place code execution (XIP) and data storage.

Thanks to the memory-mapped mode, the external memory can be accessed as if it was an internal memory allowing the system masters (such as DMA, LTDC, DMA2D, GFXMMU, SDMMC or GPU2D) to access autonomously even in low-power mode when the CPU is stopped, which is ideal for mobile and wearable applications

This application note describes the OCTOSPI, HSPI, and XSPI peripherals in STM32 MCUs and explains how to configure them in order to write and read external Octo-SPI/16-bit, HyperBus™ and regular protocol memories. This document describes some typical use cases to use the Octo/Hexadeca/XSPI interface and provides some practical examples on how to configure the OCTOSPI/HSPI/XSPI peripheral depending on the targeted memory type.

Note: The XSPI interface can be configured as: SPI for 1 line data transmission, Quad-SPI for 4 lines data transmission, Octo-SPI for 8 lines data transmission, Hexadeca-SPI for 16 lines data transmission.

Table 1. Applicable products

Type	Series or line
Microcontrollers	STM32L4+ series, STM32L5 series, STM32U5 series, STM32N6 series STM32H7A3/7B3, STM32H7B0 value line STM32H723/733, STM32H725/735, STM32H730 value line STM32H562, STM32H563/573 lines STM32H7R7/7S7, STM32H7R3/7S3 lines

Related documents

Available from STMicroelectronics web site *www.st.com*:

- reference manuals and datasheets for STM32 devices
- application note *Quad-SPI interface on STM32 microcontrollers* (AN4760)

Contents

List of tables

List of figures

1 General information

This application note provides an overview of the OCTOSPI, HSPI, and XSPI peripheral availability across the STM32 MCUs listed in *Table 1*, Arm®[a] Cortex® core-based devices.

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Overview of the OCTOSPI, HSPI, and XSPI in STM32 MCUs

2.1 OCTOSPI, HSPI, and XSPI main features

The table below summarizes the OCTOSPI main features.

Table 2. OCTOSPI main features

Feature		STM32L4R/Sxxx	STM32L4P5/Q5xx	STM32L5 series	STM32H562/ 563/573	STM32H7A3/7B3 STM32H7B0	STM32H72x/3x(1)	STM32U5 series: STM32U535/545	STM32U5 series: STM32U575/585	STM32U5 series: STM32U59x/ 5Ax	STM32U5 series: STM32U5Fx/ 5Gx
Number of instances		2		1	1	2		1	2		
Max OCTOSPI speed (MHz)(2)	Regular-command SDR mode	86	92	90	150	140	140	100	100	100	100
	Regular-command DTR mode with DQS HyperBus protocol with single-ended clock (3.3 V)	64(3)	90	76	125	120(4)	100	100	100	100	100
	HyperBus protocol with differential clock (1.8 V)	N/A	66	58	125	120(4)	100(4)	93	93	93	93
OCTOSPI I/O manager arbiter		Available		N/A		Available		N/A	Available		
Multiplexed mode		N/A	Available	N/A				N/A			
Dedicated OTFDEC support (on-the-fly decryption engine)		N/A		Available(5)							
Memory-mapped mode	Max bus frequency access (MHz)	120 (32-bit AHB bus)		110 (32-bit AHB bus)	250 (32-bit AHB bus)	280 (64-bit AXI bus)	275 (64-bit AXI bus)	160 (32-bit AHB bus)			
	Max addressable space (Mbytes)	256									

Table 2. OCTOSPI main features (continued)

Feature		STM32L4R/Sxxx	STM32L4P5/Q5xx	STM32L5 series	STM32H562/563/573	STM32H7A3/7B3 STM32H7B0	STM32H72x/3x(1)	STM32U5 series			
								STM32U535/545	STM32U575/585	STM32U59x/5Ax	STM32U5Fx/5Gx
Indirect mode	Max bus frequency access (MHz)	120 (32-bit AHB bus)		110 (32-bit AHB bus)	250 (32-bit AHB bus)	280 (32-bit AHB bus)	275 (32-bit AHB bus)	160 (32-bit AHB bus)			
	Max addressable space (Gbytes)	4									

1. Devices belonging to STM32H723/733, STM32H725/735 and STM32H730 Value line.
2. For the maximum frequency reached, refer to each product datasheet.
3. PSRAM memories are not supported.
4. Using PC2, PI11, PF0 or PF1 I/O in the data bus adds 3.5 ns to this timing value. For more details, refer to the specific product datasheet.
5. OTFDEC not supported on STM32H7A3, STM32H72x, STM32H56x, and STM32U575 devices.

The HSPI and XSPI main features are summarized in *Table 3* and *Table 4*.

Table 3. HSPI main features

Feature		STM32U59x/5Ax STM32U5Fx/5Gx
Number of instances		1
OCTOSPI I/O manager arbiter		N/A
Multiplexed mode		N/A
Dedicated OTFDEC support (on-the-fly decryption engine)		N/A
Memory-mapped mode	Max bus frequency access (MHz)	160 (32-bit AHB bus)
Indirect mode		

Table 4. XSPI main features

Feature		STM32H7Rx/Sx
Number of instances		2
XSPI I/O manager arbiter		Available
Multiplexed mode		Available
Dedicated MCE support		Available
Memory-mapped mode	Dedicated bus access	AXI
Indirect mode		

2.2 OCTOSPI, HSPI, and XSPI in a smart architecture

The OCTOSPI is an AHB/AXI slave mapped on a dedicated AHB/AXI layer. The HSPI is an AHB slave mapped on a dedicated AHB layer. The XSPI is a slave AXI mapped on a dedicated layer. This type of mapping allows the OCTOSPI, HSPI, and XSPI to be accessible as if it was an internal memory thanks to Memory-mapped mode.

In addition, the OCTOSPI, HSPI, and XSPI peripherals are integrated in a smart architecture that enables the following:

- All masters can access autonomously to the external memory in Memory-mapped mode, without any CPU intervention.
- Masters can read/write data from/to memory in Sleep mode when the CPU is stopped.
- The CPU, as a master, can access the OCTOSPI, HSPI, and XSPI and then execute code from the memory, with support of wrap operation, to enable "critical word first" access and hence improve performance in case of cache line refill.
- The DMA can do transfers to/from the OCTOSPI, HSPI, and XSPI to/from other internal or external memories.
- The graphical DMA2D can directly build framebuffer using graphic primitives from the connected Octo-SPI/16-bit flash or HyperFlash™ memory.
- The DMA2D can directly build framebuffer in Octo-SPI/16-bit SRAM, HyperFlash™, or HyperRAM™ memory.
- The GFXMMU as a master can autonomously access the OCTOSPI/HSPI/XSPI.
- The LTDC can fetch framebuffer directly from the memory that is connected to the OCTOSPI/HSPI/XSPI.
- The SDMMC master interface can transfer data between the OCTOSPI/HSPI/XSPI and SD/MMC/SDIO cards without any CPU intervention.
- The GPU2D master interface can load/store data from/to the HSPI/XSPI memory.

2.2.1 STM32L4+ series system architecture as an OCTOSPI interface example

The STM32L4+ series system architecture consists mainly of a 32-bit multilayer AHB bus matrix that interconnects multiple masters and multiple slaves.

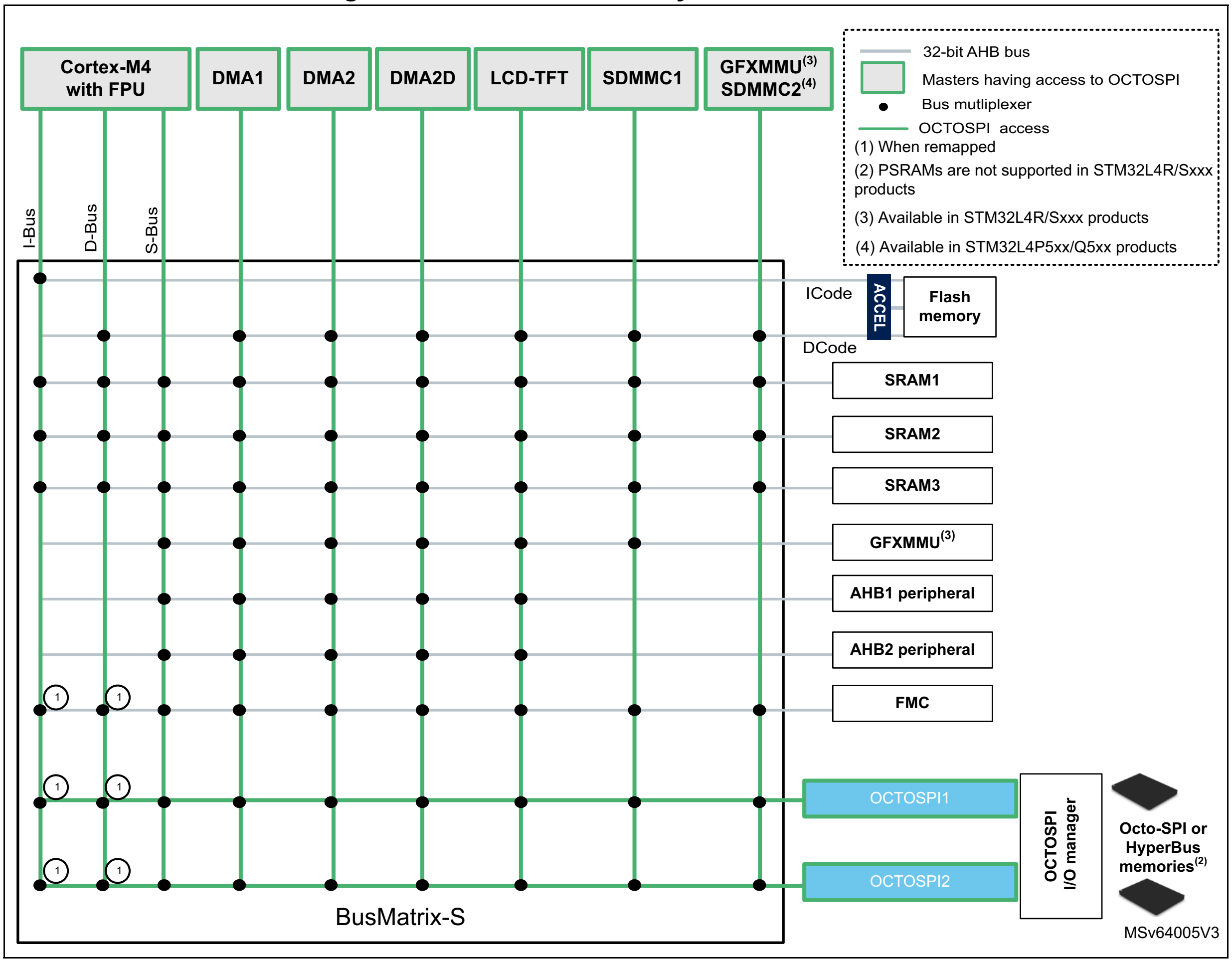
These devices integrate the OCTOSPI peripherals as described below:

- two OCTOSPI slaves (OCTOSPI1 and OCTOSPI2): each of them is mapped on a dedicated AHB layer.
- OCTOSPI slaves are completely independent from each other. Each OCTOSPI slave can be configured independently.
- Each OCTOSPI slave is independently accessible by all the masters on the AHB bus matrix.
- When the MCU is in Sleep or Low-power sleep mode, the connected memories are still accessible by the masters.
- In Memory-mapped mode:
  - OCTOSPI1 addressable space is from 0x9000 0000 to 0x9FFF FFFF.
  - OCTOSPI2 addressable space is from 0x7000 0000 to 0x7FFF FFFF.
- In a graphical application, the LTDC can autonomously fetch pixels data from the connected memory.
- The external memory connected to OCTOSPI1 or OCTOSPI2 can be accessed (for code execution or data) by the Cortex-M4 either through S-Bus, or through I-bus and D-bus when physical remap is enabled.

For main feature differences between OCTOSPIs in STM32L4+ series devices, refer to *Table 2*.

The figure below shows the OCTOSPI1 and OCTOSPI2 slaves interconnection in the STM32L4+ series system architecture.

Figure 1. STM32L4+ series system architecture



2.2.2 STM32U5 series system architecture as an HSPI example

The STM32U5 series system architecture consists mainly of a 32-bit multilayer AHB bus matrix that interconnects multiple masters and multiple slaves.

These devices integrate the OCTOSPI/HSPI peripherals as described below:

- The two OCTOSPI slave (OCTOSPI1/2) and HSPI slave are mapped on a dedicated AHB layer and accessible independently by all the masters connected to the AHB bus matrix.[a]
- When the MCU is in Sleep or Low-power sleep mode, the connected memories are still accessible by the masters.
- In Memory-mapped mode:
  - OCTOSPI1 addressable space is from 0x9000 0000 to 0x9FFF FFFF.
  - OCTOSPI2 addressable space is from 0x7000 0000 to 0x7FFF FFFF.[b]
  - HSPI addressable space is from 0xA000 0000 to 0xAFFF FFFF.
- The CPU can benefit from the ICACHE for code execution when accessing the OCTOSPI1/2 or HSPI by remap. Thanks to the ICACHE, the CoreMark® execution from the external memory can reach a highly close score to the internal flash memory.
- The CPU can also benefit from the DCACHE1 and the GPU from DCACHE2 for data transactions when accessing the OCTOSPI1/2 or HSPI.
- The CPU can profit as well from the GPU2D master interface for data load through the OCTOSPI1/2 or HSPI.

For main feature differences between OCTOSPIs in STM32U5 series devices, refer to *Table 5*.

Table 5. Instances on STM32U5 series devices

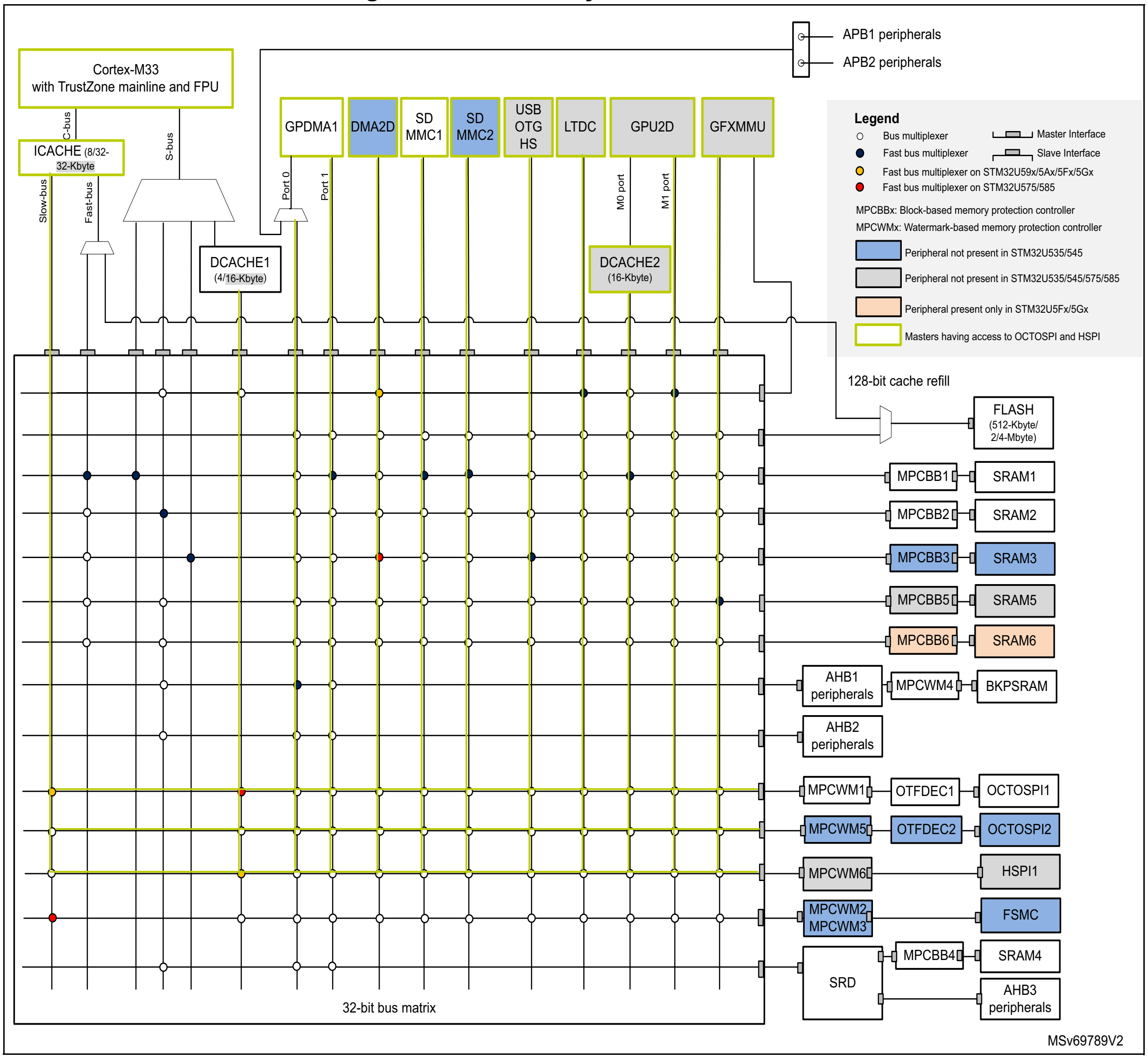
Devices	OCTOSPI1	OCTOSPI2	OCTOSPIM	HSPI1
STM32U535/545	X	-	-	-
STM32U575/585	X	X	X	-
STM32U59x/5Ax	X	X	X	X
STM32U5Fx/5Gx	X	X	X	X

The figure below shows the OCTOSPI and HSPI slaves interconnection in the STM32U5 series system architecture.

a. Only one OCTOSPI instance for STM32U535/545

b. For STM32U575/585, STM32U59x/5Ax, STM32U5Fx/5Gx

Figure 2. STM32U5 system architecture



2.2.3 STM32H7Rx/Sx system architecture as an XSPI example

The STM32H7Rx/Sx series system architecture consists of a 64-bit AXI and 32-bit multilayer AHB bus matrix and bus bridges that allow interconnecting bus masters with bus slaves, interconnecting 19 masters and 20 slaves.

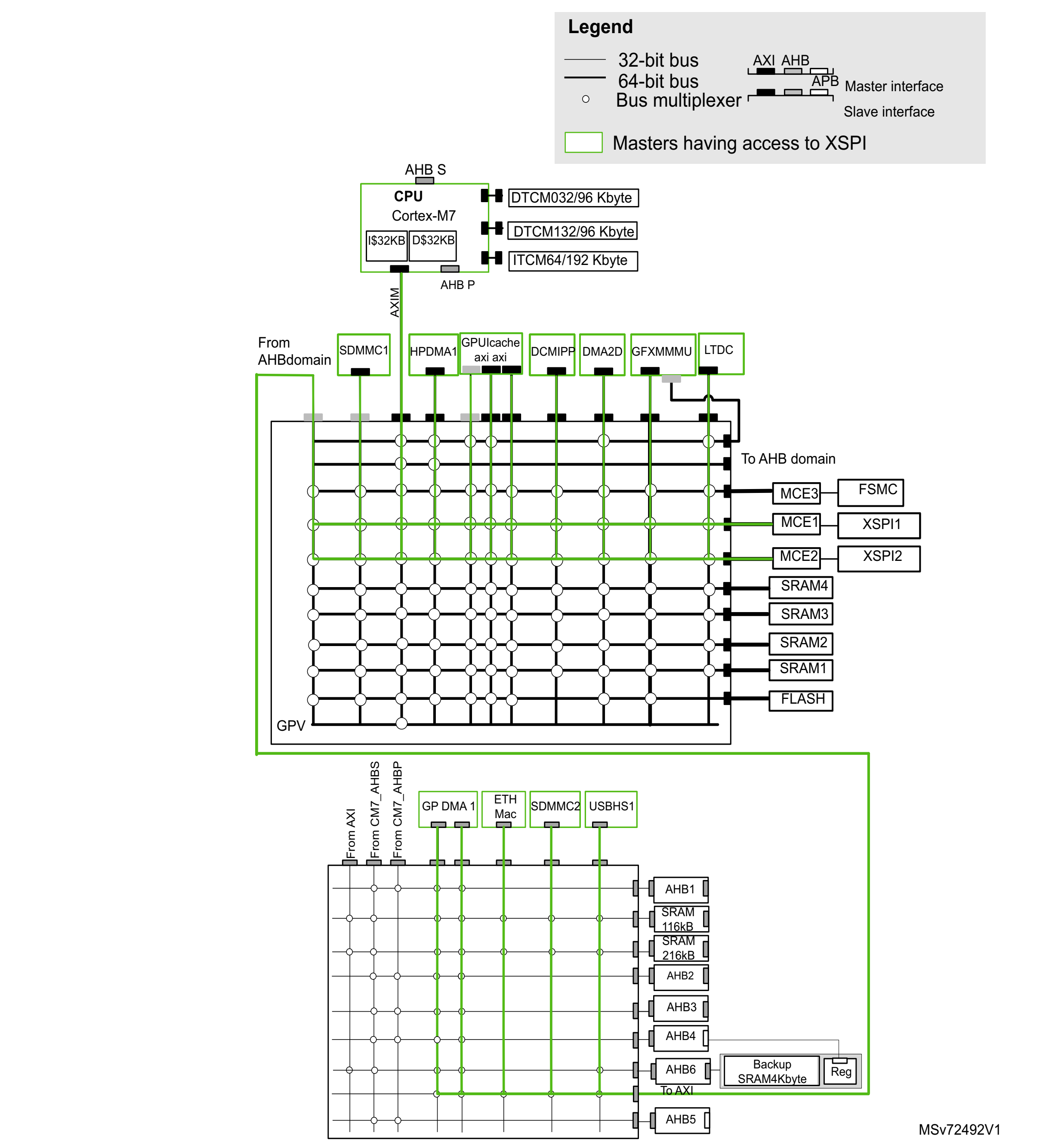
These devices integrate two slaves (XSPI1 and XSPI2), with the following characteristics.

The system of these devices integrates the XSPI peripheral as described below:

- XSPI slaves are completely independent from each other. Each XSPI slave can be configured independently.
- Each XSPI slave is independently accessible by all the masters on the AXI bus matrix or AHB bus matrix and AXI bus matrix.
- In memory mapped mode:
  - XSPI1 addressable space is from 0x9000 0000 to 0x9FFF FFFF
  - XSPI2 addressable space is from 0x7000 0000 to 0x7FFF FFFF
- In a graphical application, the LTDC can autonomously fetch pixels data from the connected memory.

The figure below shows the XSPI slaves interconnection in the STM32H7Rx/Sx series system architecture.

Figure 3. STM32H7Rx/Sx system architecture



3 Octo/Hexadeca/XSPI interface description

The Octo-SPI is a serial interface that allows communication on eight data lines between a host (STM32) and an external slave device (memory). The Hexadeca-SPI is a serial interface that allows communication on 16 data lines between a host (STM32) and an external slave device (memory). The XSPI is a serial interface that allows communication 16 data lines between a host (STM32) and an external slave device (memory).

This interface is integrated on the STM32 MCU to fit memory-hungry applications without compromising performances, to simplify PCB (printed circuit board) designs and to reduce costs.

3.1 OCTOSPI, HSPI, and XSPI hardware interfaces

The OCTOSPI provides a flexible hardware interface, that enables the support of multiple hardware configurations: Single-SPI (legacy SPI), Dual-SPI, Quad-SPI, Dual-quad-SPI and Octo-SPI. The HSPI and XSPI integrate all protocols supported by the OCTOSPI and it provides new ones: the Dual-octal and the 16-bit modes. The XSPI integrates all protocols supported by:

- the OCTOSPI when the XSPI has only 8 data lines
- the HSPI when the XSPI has 16 data lines

They also support the HyperBus protocol with single-ended clock (3.3 V signals) or differential clock (1.8 V signals). The flexibility of the Octo/Hexadeca/XSPI hardware interface permits the connection of most serial memories available in the market.

3.1.1 OCTOSPI pins and signal interface

The Octo-SPI interface uses the following lines:

- OCTOSPI_NCS line for chip select
- OCTOSPI_CLK line for clock
- OCTOSPI_NCLK to support 1.8 V HyperBus protocol
- OCTOSPI_DQS line for data strobe/write mask signals to/from the memory
- OCTOSPI_IO[0...7] lines for data

Note: The HyperBus differential clock (1.8 V) is not supported with the STM32L4Rxxx and STM32L4Sxxx products.

Figure 7 shows Octo-SPI interface signals.

3.1.2 HSPI pins and signal interface

The HSPI interface uses the following lines:

- HSPI_NCS line for chip select
- HSPI_CLK line for clock
- HSPI_NCLK to support 1.8 V HyperBus protocol
- HSPI_DQS0/1 line for data strobe/write mask signals to/from the memory
- HSPI_IO[0...15] lines for data

3.1.3 XSPI manager pins and signal interface

The XSPI interface uses the following lines:

- XSPIM_Px_NCS1/2 line for chip select
- XSPIM_Px_CLK line for clock
- XSPIM_Px_NCLK to support 1.8 V HyperBus protocol
- XSPIM_Px_DQS0/1 line for data strobe/write mask signals to/from the memory
- XSPIM_Px_IO[0...15] lines for data

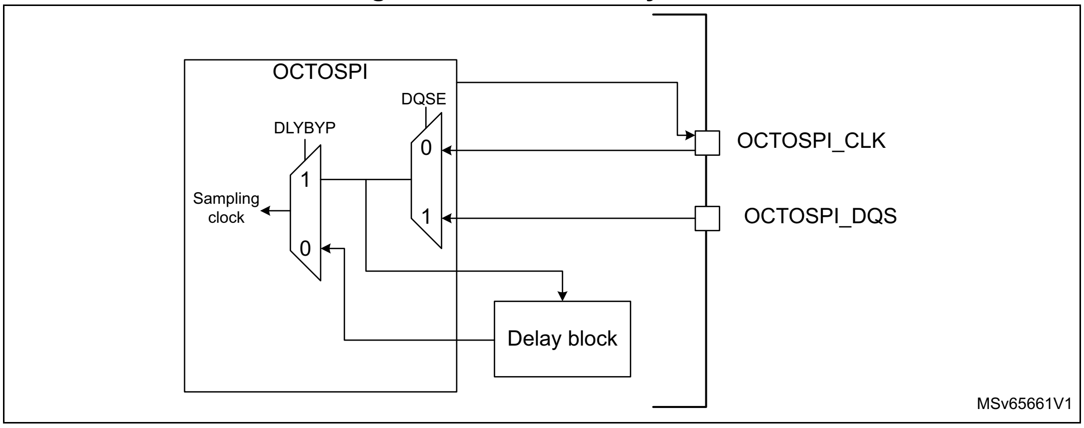
  x = 1 to 2

3.1.4 OCTOSPI delay block

The delay block (DLYB) can be used to insert delays between data and DQS or CLK, during data read operations to compensate for data propagation delays.

The following figure shows the DLYB and OCTOSPI interconnection.

Figure 4. OCTOSPI delay block



From the OCTOSPI registers, the user can chose:

- to select the delay block output clock as sampling clock or not, by enabling/disabling the DLYBYP bit in the OCTOSPI_DCR1 register.
- to select the delay block delayed signal (clock or DQS) by enabling/disabling the DQSE bit.

To operate properly and deliver a precise delay, the delay block must be calibrated before use:

- In the STM32L4+ and STM32L5 series, the delay block is a feature provided to the OCTOSPI interface, for which unitary delays can be configured from OCTOSPIx_DLY in the RCC_DLYCFGR register.
- In the STM32H7A3/7B3/7B0, STM32H72x/73x, and STM32U5 devices, the delay block is an independent peripheral that can be configured for the Octo-SPI interface.

For more informations about delay block configuration, refer to the delay block section in the product reference manual. For more information about delay block and unitary delays characteristics, refer to the specific product datasheet.

3.2 HSPI and XSPI high-speed interfaces and calibration

3.2.1 HSPI and XSPI high-speed interfaces

To reach higher frequencies, a dedicated high-speed interface is inserted between the HSPI/XSPI and the I/O pads. The aim of the calibration is to have correct a delay to sample the data. It might also be inserted between the I/O manager in case the considered device embeds an I/O manager and I/O pads. Registers found in the HSPI/XSPI are used to control the high-speed interface functionality.

Re-synchronization registers are embedded in the block and are clocked by a delayed clock generated by a DLL (see Note below) which is also present in the high-speed interface. The main purpose of the re-synchronization phase is primary to shift data or data strobe by one quarter of octal bus clock period, with a correct timing accuracy. DLL must be calibrated versus this clock period. When using the high-speed-interface, the system clock (AHB clock or AXI clock) must be at least as fast as the SPI clock.

Note: DLL: the delay locked loop

3.2.2 HSPI and XSPI calibration process

The HSPI and XSPI interfaces have a specialized and dedicated mechanism for the calibration procedure.

The calibration process is automatically enabled within the interface when one of the conditions below is met:

- The XSPI/HSPI exits reset state.
- A value is written in PRESCALER field of the HSPI_DCR2/ XSPI_DCR2 register for HSPI/XSPI interface.
- A value is written in HSPI_CCR/ XSPI_CCR register for HSPI/XSPI interface.

Other than having one of the three aforementioned circumstances which make the calibration possible, when having the bit BUSY = 1, a calibration process is set on the interface and begins.

Note: Once the calibration is completed, BUSY returns to 0.

Some applications need periodic recalibration. The HSPI/XSPI high-speed interface performs this kind of recalibration. It must be triggered only by writing periodically in PRESCALER[7:0] of HSPI_DCR2/ XSPI_DCR2 for HSPI/XSPI interface, while BUSY = 0.

Setting the PRESCALER field automatically starts a new calibration of high-speed interface DLL at the start of next transfer, except in case HSPI_CALOSR/ XSPI_CALOSR or HSPI_CALISR/ XSPI_CALISR have been written in the meantime for HSPI/XSPI. BUSY stays high during the whole calibration execution.

The user can access the delay values set by auto-calibration process, for the feedback clock delay (in HSPI_CALMR/XSPI_CALMR for HSPI/XSPI), the data output delay (in HSPI_CALSOR/ XSPI_CALSOR for HSPI/XSPI), and the data strobe input delay (in HSPI_CALSIR/XSPI_CALSIR for HSPI/XSPI). After the auto-calibration of the interface, HSPI_CALSOR/ XSPI_CALSOR and HSPI_CALSIR/ XSPI_CALSIR for HSPI/XSPI have the same value. This value corresponds to the delay for a quarter cycle.

Note: When the controller is configured with DTR mode and prescaler is bypassed (PRESCALER[7:0] =0), the kernel clock provided to interface must have a 50% duty-cycle.

3.3 Two low-level protocols

The Octo/Hexadeca/XSPI interface can operate in two different low-level protocols: Regular-command and HyperBus. Each protocol supports three operating modes:

- Indirect mode
- Memory-mapped mode

The Regular-command supports the Automatic status-polling operating mode.

3.3.1 Regular-command protocol

The Regular-command protocol is the classical frame format where the OCTOSPI, HSPI, and XSPI communicate with the external memory device by using commands where each command can include up to five phases. The external memory device can be a Single-SPI, Dual-SPI, Quad-SPI, Dual-quad-SPI, Octo-SPI, dual-octal or 16-bit memory.

Flexible-frame format and hardware interface

The Octo/Hexadeca/XSPI interface provides a fully programmable frame composed of five phases. Each phase is fully configurable, allowing the phase to be configured separately in terms of length and number of lines.

The five phases are the following:

- Instruction phase: can be set to send a 1-, 2-, 3- or 4-byte instruction (SDR or DTR). This phase can send instructions using the Single-SPI, Dual-SPI, Quad-SPI, Octo-SPI, or 16-bit SPI mode.
- Address phase: can be set to send a 1-, 2-, 3- or 4-byte address. This phase can send addresses using the Single-SPI, Dual-SPI, Quad-SPI, Octo-SPI, or 16-bit SPI mode mode.
- Alternate-bytes phase: can be set to send a 1-, 2-, 3- or 4-alternate bytes. This phase can send alternate bytes using the Single-SPI, Dual-SPI, Quad-SPI, Octo-SPI, or 16-bit SPI mode.
- Dummy-cycles phase: can be set to 0 to up to 31 cycles.
- Data phase: for Indirect or Automatic status-polling mode, the number of bytes to be sent/received is specified in the OCTOSPI_DLR / HSPI_DLR / XSPI_DLR register. For Memory-mapped mode the bytes are sent/received following any AHB/AXI data interface. This phase can send/receive data using the Single-SPI, Dual-SPI, Quad-SPI, Octo-SPI or 16-bit mode[a].

Any of these phases can be configured to be skipped.

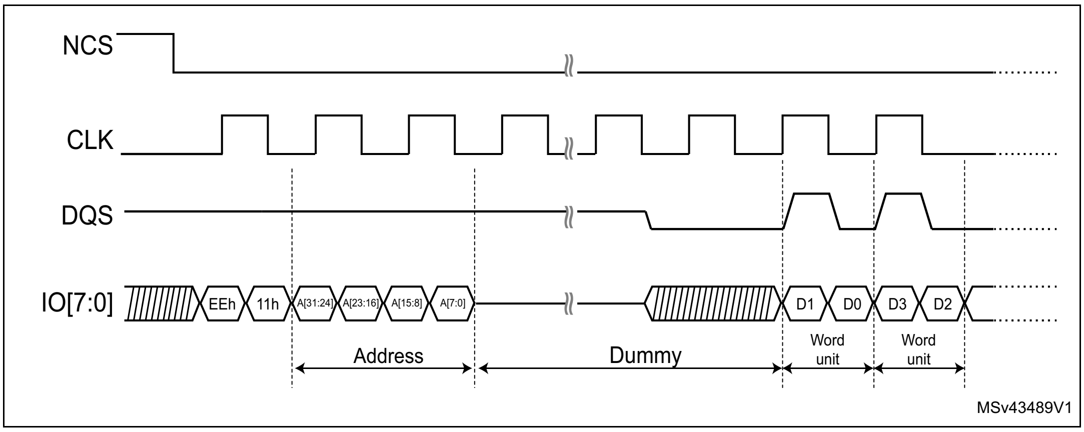
The figure below illustrates an example of an octal DTR read operation, showing instruction, address, dummy and data phases.

Data strobe (DQS) usage

The DQS signal can be used for data strobing during the read transactions when the device is toggling the DQS aligned with the data.

a. Only the HSPI/XSPI can be supported by the 16-bit mode.

Figure 5. Regular-command protocol: octal DTR read operation example in Macronix mode



3.3.2 HyperBus protocol

The OCTOSPI/HSPI/XSPI support the HyperBus protocol that enables the communication with HyperRAM and HyperFlash memories.

The HyperBus has a double-data rate (DTR) interface where two data-bytes per clock cycle are transferred over the DQ input/output (I/O) signals for OCTOSPI/HSPI and over the IO input/output (I/O) signals for XSPI interface, leading to high read and write throughputs.

Note: For additional information on HyperBus interface operation, refer to the HyperBus specification protocol.

The HyperBus frame is composed of two phases:

- Command/address phase: the OCTOSPI/HSPI/XSPI sends 48 bits (CA[47:0]) over IO[7:0] to specify the operations to be performed with the external device.
- Data phase: the OCTOSPI performs data transactions from/to the memory in Octal-SPI mode. The HSPI can transfer data in Octal-SPI or 16-bit mode. The XSPI can transfer data in Octal-SPI mode when the XSPI allows communication on only 8 data lines and can transfer data in 16-bit mode when the XSPI allows communication on 16 data lines.

During the command/address (CA) phase, the read-write data strobe (named RWDS or DQS) is used by the HyperRAM memory to indicate if an additional initial access latency has to be inserted or not. If RWDS/DQS was low during the CA period, only one latency count is inserted (t_{ACC} initial access). If RWDS/DQS was high during the CA period, an additional latency count is inserted ($2*t_{ACC}$).

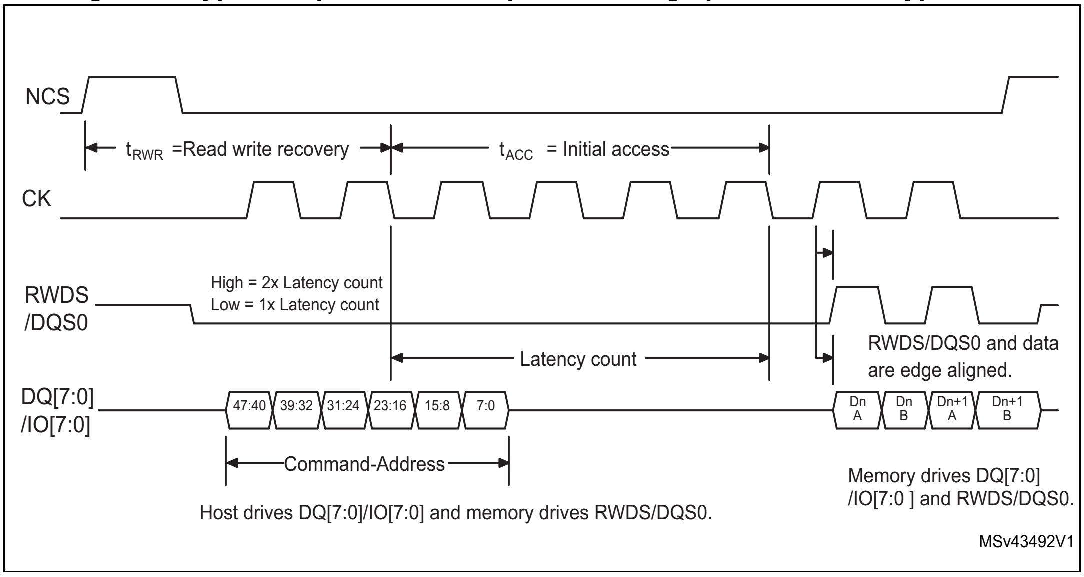
The initial latency count (t_{ACC}) represents the number of clock cycles without data transfer used to satisfy any initial latency requirements before data is transferred. The initial latency count required for a particular clock frequency is device dependent, it is defined in the memory device configuration register.

Note: - For HyperFlash memories, the RWDS/DQS is only used as a read data strobe.

- The RWDS from memory corresponds to DQS for the OCTOSPI/HSPI interface from the STM32 MCU.

The figure below illustrates an example of an HyperBus read operation.

Figure 6. HyperBus protocol: example of reading operation from HyperRAM



Depending on the application needs, the OCTOSPI/HSPI/XSPI peripheral can be configured to operate in the following HyperBus modes:

- HyperBus memory mode: the protocol follows the HyperBus specification, allowing read/write access from/to the HyperBus memory.
- HyperBus register mode: must be used to access to the memory register space, that is useful for memory configuration.

3.4 Three operating modes

Whatever the used low-level protocol, the OCTOSPI/HSPI/XSPI can operate in the indirect mode and in the memory-mapped mode. When using the Regular-command protocol, the OCTOSPI/HSPI/XSPI can operate in the Automatic status-polling mode. The three operating modes detailed below.

3.4.1 Indirect mode

The Indirect mode is used in the following cases (whatever the HyperBus or Regular-command protocol):

- read/write/erase operations
- if there is no need for AHB or AXI masters to access autonomously the OCTOSPI/HSPI/XSPI peripheral (available in Memory-mapped mode)
- for all the operations to be performed through the OCTOSPI/HSPI/XSPI data register, using CPU or DMA
- to configure the external memory device

3.4.2 Automatic status-polling mode

The Automatic status-polling mode allows an automatic polling fully managed by hardware on the memory status register. This feature avoids the software overhead and the need to perform software polling. An interrupt can be generated in case of match.

The Automatic status-polling mode is mainly used in the below cases:

- to check if the application has successfully configured the memory: after a write register operation, the OCTOSPI/HSPI/XSPI periodically reads the memory register and checks if bits are properly set. An interrupt can be generated when the check is ok.
  - Example: this mode is commonly used to check if the write enable latch bit (WEL) is set. Once the WEL bit is set, the status match flag is set and an interrupt can be generated (if the status-match interrupt-enable bit (SMIE) is set)
- to autonomously poll for the end of an ongoing memory operation: the OCTOSPI/HSPI/XSPI polls the status register inside the memory while the CPU continues the execution. An interrupt can be generated when the memory operation is finished.
  - Example: this mode is commonly used to wait for an ongoing memory operation (programming/erasing). The OCTOSPI/HSPI/XSPI in Automatic status-polling mode reads continuously the memory status register and checks the write-in progress bit (WIP). As soon as the operation ends, the status-match flag is set and an interrupt can be generated (if SMIE is set).

3.4.3 Memory-mapped mode

The Memory-mapped mode is used in the cases below:

- read and write operations
- to use the external memory device exactly like an internal memory (so that any AHB/AXI master can access it autonomously)
- for code execution from an external memory device

In Memory-mapped mode, the external memory is seen by the system as if it was an internal memory. This mode allows all AHB/AXI masters to access an external memory device as if it was an internal memory. The CPU can execute code from the external memory as well.

When the Memory-mapped mode is used for reading, a prefetching mechanism, fully managed by the hardware, enables the optimization of the read and the execution performances from the external memory.

Each OCTOSPI/ HSPI/XSPI peripheral is able to manage up to 256 Mbytes of memory space:

- OCTOSPI1 addressable space: from 0x9000 0000 to 0x9FFF FFFF
- OCTOSPI2 addressable space: from 0x7000 0000 to 0x7FFF FFFF
- HSPI addressable space: from 0xA000 0000 to 0xAFFF FFFF
- XSPI1 addressable space: from 0x9000 0000 to 0x9FFF FFFF
- XSPI2 addressable space: from 0x7000 0000 to 0x7FFF FFFF

Note: For the HSPI memory-mapped region, it is required to configure the MPU in order to support the XIP.

Starting memory-mapped read or write operation

A memory-mapped operation is started as soon as there is an AHB/AXI master read or write request to an address in the range defined by DEVSIZE.

If there is an on-going memory-mapped read (respectively write) operation, the application can start a write operation as soon as the on-going read (respectively write) operation is terminated.

Note: Reading the OCTOSPI_DR/HSPI_DR/XSPI_DR data register in Memory-mapped mode has no meaning and returns 0. The data length register OCTOSPI_DLR/HSPI_DR/XSPI_DR has no meaning in Memory-mapped mode.

Execute in place (XIP)

The OCTOSPI /HSPI/XSPI supports the execution in place (XIP) thanks to its integrated prefetch buffer. The XIP is used to execute the code directly from the external memory device. The OCTOSPI /HSPI/XSPI loads data from the next address in advance. If the subsequent access is indeed made at a next address, the access is completed faster since the value is already prefetched.

Send instruction only once (SIOO)

The SIOO feature is used to reduce the command overhead and boost non-sequential reading performances (like execution). When SIOO is enabled, the command is sent only once, when starting the reading operation. For the next accesses, only the address is sent.

Note: Refer to the reference manual to confirm the availability of this feature for the selected product.

4 OCTOSPI and XSPI I/O managers

The OCTOSPI I/O manager allows the user to set a fully programmable pre-mapping of the OCTOSPI1 and OCTOSPI2 signals. The XSPI I/O manager allows the user to set a fully programmable pre-mapping of the XSPI1 and XSPI2 signals. Any OCTOSPIM_Pn_x / XSPIM_Pn_x port signal can be mapped independently to the OCTOSPI1 or OCTOSPI2 / XSPI1 or XPI2.

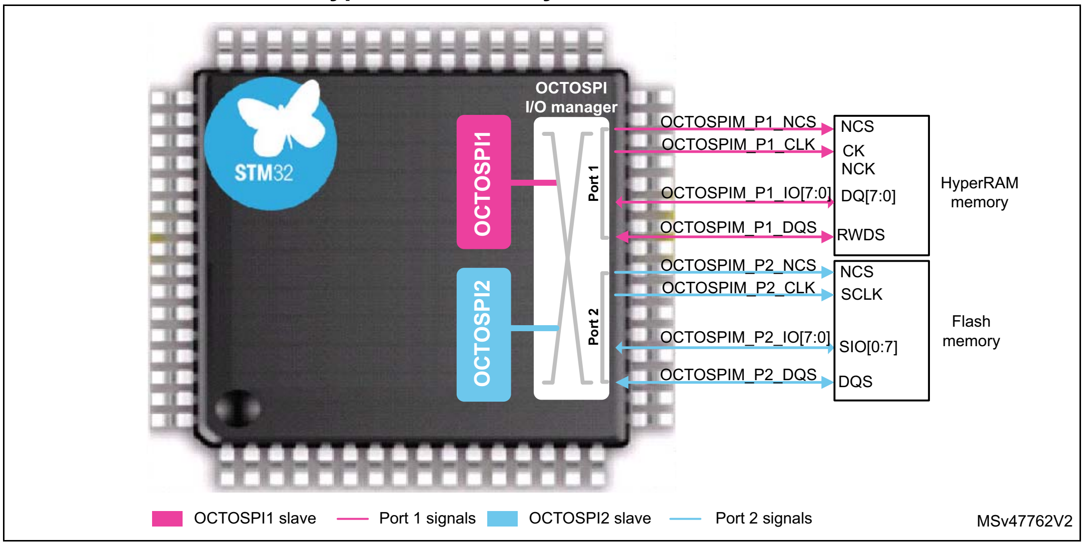
By default, after reset, all the signals of the OCTOSPI1 / XSPI1 and OCTOSPI2 / XSPI2 are mapped respectively on Port1 and Port2.

For instance when two external memories are used, an HyperRAM can be connected to Port1 and flash memory can be connected to Port2. An example of connecting an Octo-SPI flash memory and an HyperRAM memory to an STM32 device using an OCTOSPI I/O manager as shown in the figure below. In that case, the user has two possibilities:

- HyperRAM memory linked to OCTOSPI1 and flash memory linked to OCTOSPI2
- HyperRAM memory linked to OCTOSPI2 and flash memory linked to OCTOSPI1

The figure below shows an example of Octo-SPI flash and an HyperRAM memories connected to the STM32 MCU using the Octo-SPI interface. Thanks to the OCTOSPI I/O manager, the HyperRAM memory can be linked to the OCTOSPI1 and the flash memory can be linked to the OCTOSPI2, and vice versa.

Figure 7. Example of connecting an Octo-SPI flash memory and an HyperRAM memory to an STM32 device



OCTOSPI and XSPI I/O managers Multiplexed mode

The OCTOSPI/XSPI I/O manager implements a Multiplexed mode feature.When enabled, both OCTOSPI1/2 or both XSPI1/2 signals are mixed over one OCTOSPI/XSPI I/O port except the OCTOSPI1/2_NCS / XSPI1/2_NCS1/2 pins. Each XSPI delivers one NCS signal together with a CSSEL control signal. The CSSEL signal selects which of the two outputs (NCS1, NCS2) is active. In multiplexed modes the active CCSEL signal is the one of the XSPI owning the bus. A configurable arbitration system manages the transactions to the two external memories.

This feature allows two external memories to be exploited using few pins on small packages, in order to reduce the number of pins, PCB design cost and time.

The Multiplexed mode is enabled after setting MUXEN bit in OCTOSPIM_CR or in XSPIM_CR for OCTOSPI and XSPI I/O manager.

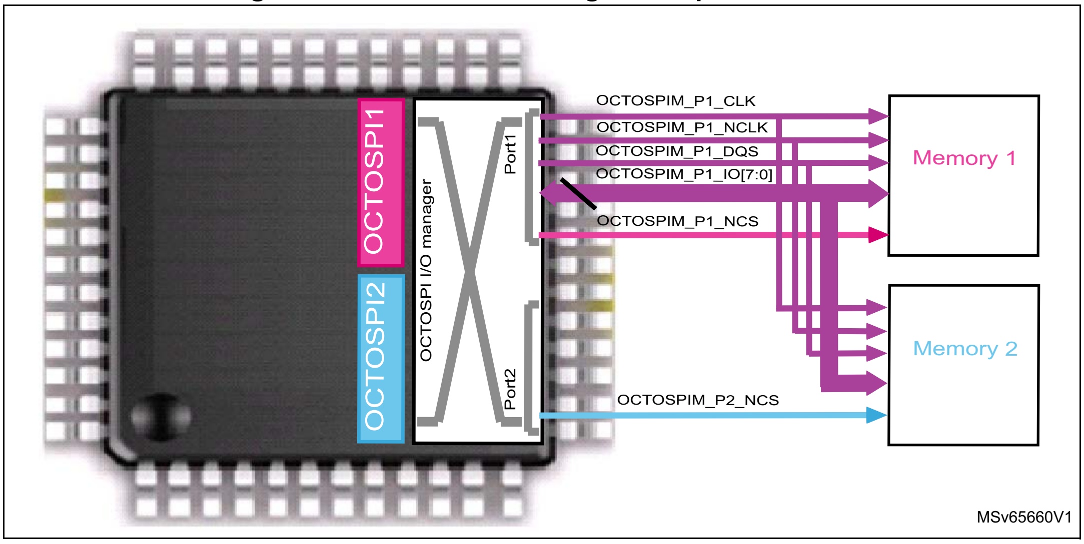
For the OCTOSPI/XSPI interfaces,the arbitration system can be configured with MAXTRAN[7:0] field in OCTOSPI_DCR3 /XSPI_DCR3 register, respectively. This field manages the max duration in which the OCTOSPIx / XSPIx takes control of the bus. If MAXTRAN + 1 OCTOSPI /XSPI bus clock cycles is reached and the second OCTOSPI /XSPI is not requesting an access, the transaction is not stopped and NCS is not released.

The time between transactions in Multiplexed mode can be managed with REQ2ACK_TIME[7:0] field in OCTOSPIM_CR /XSPIM_CR in case of the OCTOSPI and XSPI interface register, respectively.

The following figure shows an example of two connected memories over two OCTOSPI instances using only 13 pins thanks to the Multiplexed mode.

To enable the Multiplexed mode, at least one OCTOSPI I/O port signals and the CS signal from the other port must be accessible.

Figure 8. OCTOSPI I/O manager Multiplexed mode



Note: The Multiplexed mode is only available on STM32L4P5xx/Q5xx, STM32H7A3/B3/B0, STM32H72x/73x, STM32U5F9/U5G7/U5G9, STM32U595/U599/U5A/U5A9, STM32U575/585, and STM32H7Rx/Sx devices.

The XSPI I/O manager matrix allows the user to set multiplexed modes for these cases:

- Multiplexed to Port1: XSPI1 and XSPI2 both mapped to Port 1, with arbitration (multiplexed mode)
- Multiplexed to Port2: XSPI1 and XSPI2 both mapped to Port 2, with arbitration (multiplexed mode)

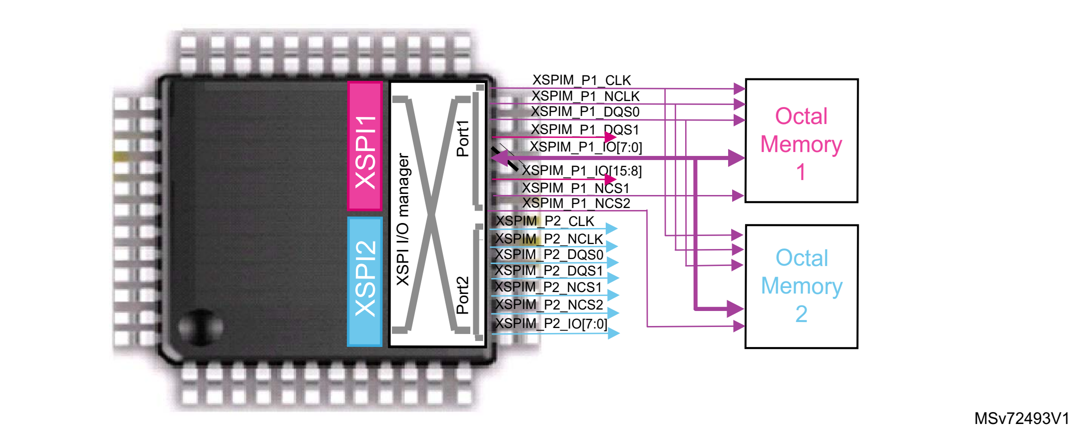
Note: There is no possibility to use mixed combinations of signals (like NCS of XSPI1 with data of XSPI2).

The following figure shows an example of two XSPIs multiplexed mode to Port 1 accessing two external memories.

The arbiter in I/O manager selects XSPI1 or XSPI2 to own the octal-SPI bus according to the existing transfer requests and status of the two MAXTRAN fairness counters.

The external memories can be two separate chips or embedded in a single multi chip package. In this case, each memory requests a chip select, selected according to the CSSEL control of the XSPI currently owning the bus.

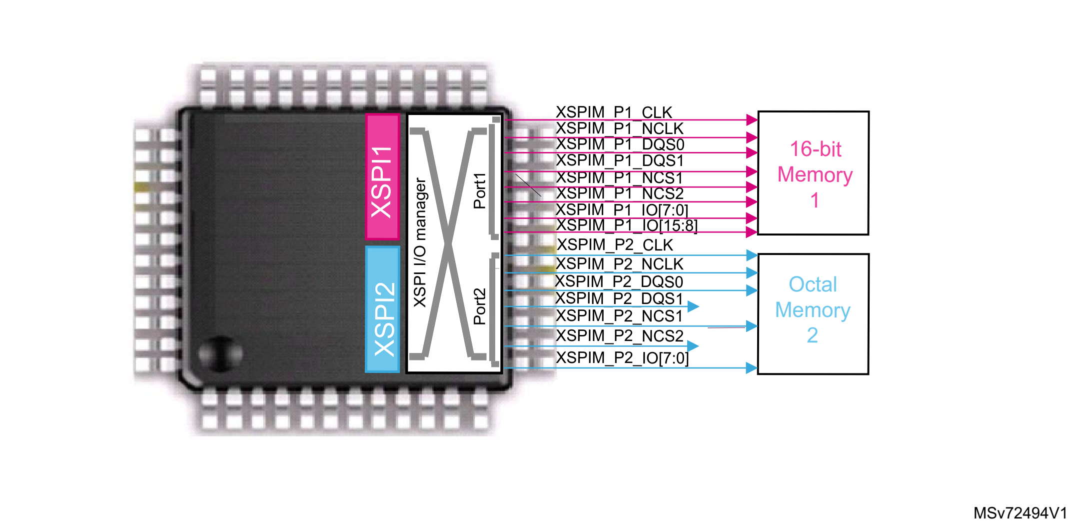
Figure 9. XSPI I/O manger Multiplexed mode



XSPI I/O manager swapped mode

The swapped mode is similar to XSPI direct mode, but the ports are swapped to help I/O mapping. In this mode, the XSPI2 can be configured in 16-bit mode, and the XSPI1 can be configured in octal mode to connect an external 16-bit memory on Port 1, and to connect in a concurrent way, an octal external memory to Port 2. The figure below shows an example of XSPI I/O manager swapped mode.

Figure 10. XSPI I/O manager Swapped mode



5 OCTOSPI, HSPI, and XSPI configuration

In order to enable the read or write from external memory, the application must configure the OCTOSPI/HSPI/XSPI peripheral and the connected memory device.

There are some common and some specific configuration steps regardless of the low-level protocol used (Regular-command or HyperBus protocol).

- OCTOSPI/HSPI/XSPI common configuration steps:
  - GPIOs and OCTOSPI/XSPI I/O manager configuration
  - interrupts and clock configuration
- OCTOSPI/HSPI/XSPI specific configuration steps:
  - OCTOSPI/HSPI/XSPI low-level protocol specific configurations (Regular-command or HyperBus)
  - memory device configuration

The following subsections describe all needed OCTOSPI/HSPI/XSPI configuration steps to enable the communication with external memories.

5.1 OCTOSPI, HSPI, and XSPI common configuration

This section describes the common steps needed to configure the OCTOSPI/HSPI/XSPI peripheral regardless of the used low-level protocol (Regular-command or HyperBus).

Note: It is recommended to reset the OCTOSPI/HSPI/XSPI *peripheral before starting a configuration. This action also guarantees that the peripheral is in reset state.*

5.1.1 GPIOs, OCTOSPI/HSPI/XSPI I/Os configuration

The user has to configure the GPIOs to be used for interfacing with the external memory. The number of GPIOs to be configured depends on the preferred hardware configuration (Single-SPI, Dual-SPI, Quad-SPI, Dual-quad-SPI, Octo-SPI, dual octal-SPI or 16-bit).

Octo-SPI mode when one memory is connected

Ten GPIOs are needed. An additional GPIO for DQS is optional for the Regular-command protocol and mandatory for the HyperBus protocol. An additional GPIO for differential clock (NCLK) is also needed only in HyperBus protocol 1.8 V.

HSPI and XSPI mode with single 16-bit configuration

21 GPIOs are needed, two additional GPIO for separate DQS data strobe/write mask signals used: DQS0 for the eight IO[7:0]: and DQS1 for the eight IO[15:8].

An additional GPIO for differential clock (NCLK) is also needed only in HyperBus protocol 1.8 V.

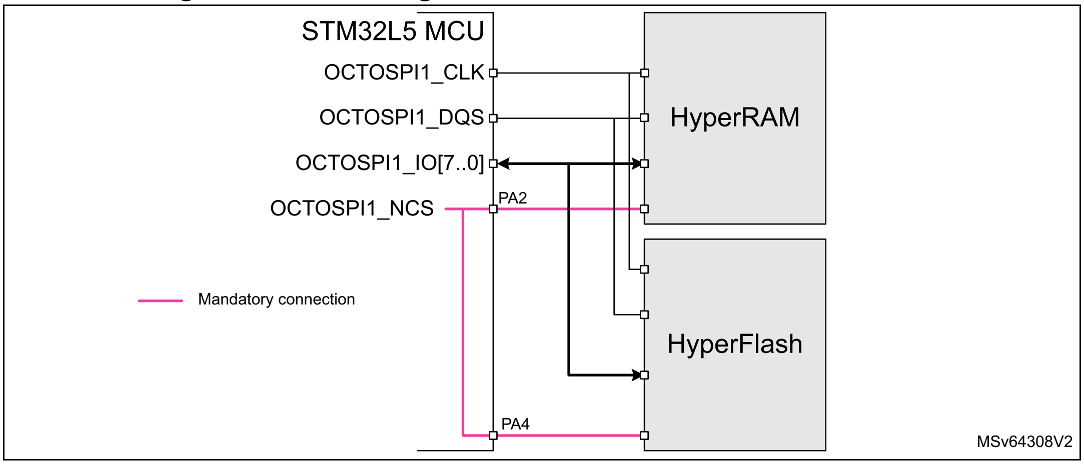
Octo-SPI mode when two external octal memories are connected

- **to one Octo-SPI interface** using pseudo-static communication

  Example: one HyperRAM and one HyperFlash connected to an STM32L5 series MCU in single-ended clock, in order to execute code from the external HyperFlash at the start of the application, then switch to the HyperRAM for data transfer.

  The two memories must be connected to the same instance, then the CS pin of each memory must be connected to an OCTOSPI_NCS GPIO port as demonstrated in the figure below. This connection requires 12 GPIOs. In this case, and when transfering data to HyperFlash memory, it is recommended to set the HyperRAM chip select (PA2) to high voltage by using a pull-up resistor for example.

Figure 11. Connecting two memories to an Octo-SPI interface



- **to two Octo-SPI interfaces**
  - with Multiplexed mode disabled/not supported: Each memory must be connected to an OCTOSPI I/O manager port. It requires up to 24 GPIOs (see example in *Figure 7*).
  - with Multiplexed mode enabled: Both memories are connected to an OCTOSPI I/O manager port. Only the second memory requires an additional GPIO for NCS from the remaining OCTOSPI I/O manager port. It requires up to 13 GPIOs (see example in *Figure 8*).

The user must select the proper package depending on its needs in terms of GPIOs availability.

The OCTOSPI GPIOs must be configured to the correspondent alternate function. For more details on OCTOSPI alternate functions availability versus GPIOs, refer to the alternate function mapping table in the product datasheet.

Note: All GPIOs have to be configured in very high-speed configuration.

GPIOs configuration using STM32CubeMX

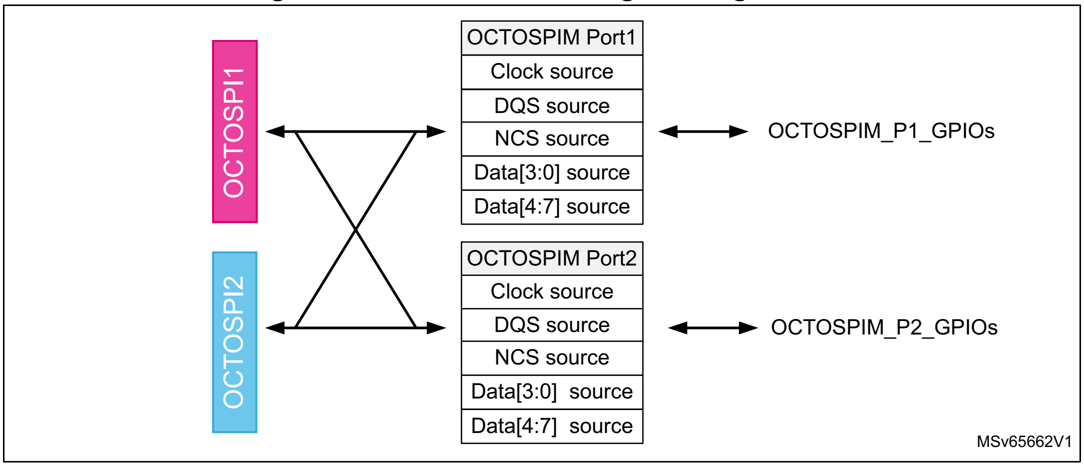
Thanks to the STM32CubeMX tool, the OCTOSPI/XSPI peripheral and its GPIOs can be configured very simply, easily and quickly. The STM32CubeMX is used to generate a project with a preconfigured OCTOSPI/XSPI. *Section 6.2.3* details how to configure the OCTOSPI GPIOs.

OCTOSPI and XSPI I/O manager configuration

By default, after reset, all OCTOSPI1/XSPI1 and OCTOSPI2/XSPI2 signals are mapped respectively to Port 1 and to Port 2.

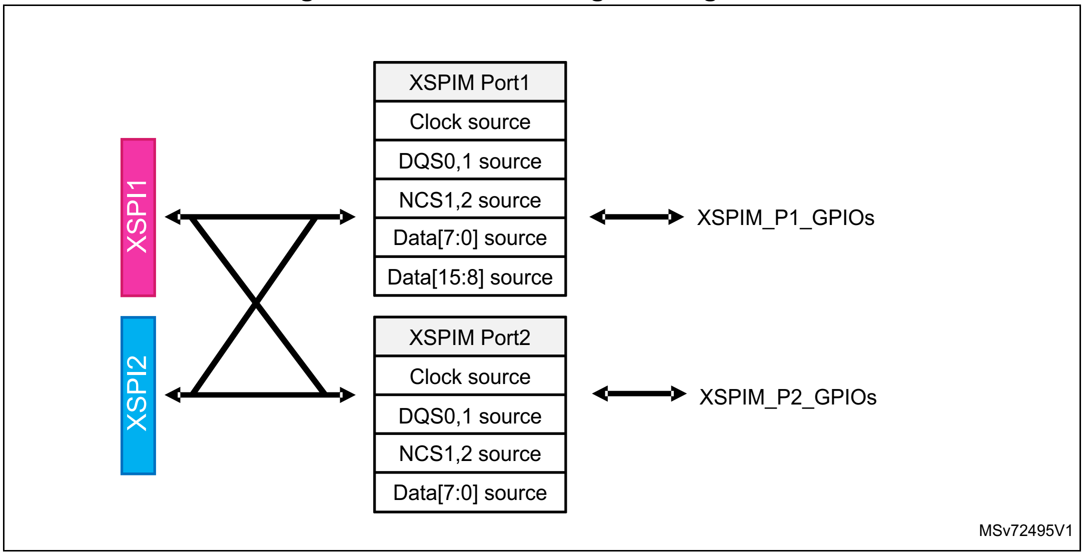
For OCTOSPI, the user can configure the OCTOSPIM_PnCR (n = 1 to 2) registers in order to select the desired source signals for the configured port as shown in the figure below:

Figure 12. OCTOSPI I/O manager configuration



For the XSPI, the user can configure the XSPIM_CR registers in order to select the desired source signals for the configured port as shown in the figure below:

Figure 13. XSPI I/O manager configuration



To enable the Multiplexed mode for an OCTOSPI/XSPI interface, the user must configure the OCTOSPIM_PnCR (n = 1 to 2) register and XSPIM_CR register in order to:

- select the desired port to be muxed in and enable it.
  The remaining signals of not selected ports must be configured to unused in Multiplexed mode and disabled.
- configure and enable NCS for both Port 1 and Port 2.

After configuring both OCTOSPIM_PnCR/XSPIM_CR (n = 1 to 2), enable Multiplexed mode by setting MUXEN bit in OCTOSPIM_CR/XSPIM_CR, the user can also configure the REQ2ACK_TIME[7:0] to define the time between two transactions.

During Multiplexed mode, configuring each OCTOSPI/XSPI MAXTRAN feature in OCTOSPI_DCR3/XSPI_DCR3 allows the user:

- to limit an OCTOSPI/XSPI to allocate the bus during all the data transaction precisely during long burst (example: DMA2D bursts)
- to privilege or not an OCTOSPI/XSPI through put from another

Note: The OCTOSPI I/O manager is not supported in STM32L5 series products. For multiplexed mode, it is recommended to enable at least MAXTRAN or timeout feature for each OCTOSPI/XSPI instance.

5.1.2 Interrupts and clocks configuration

This section describes the steps required to configure interrupts and clocks.

Enabling interrupts

Each OCTOSPI, HSPI, or XSPI peripheral has its dedicated global interrupt connected to the NVIC.

To be able to use OCTOSPI1 and/or OCTOSPI2 and/or HSPI and/or XSPI1 and/or XSPI2 interrupts, the user must enable the OCTOSPI1 and/or OCTOSPI2 and/or HSPI and/or XSPI1 and/or XSPI2 global interrupts on the NVIC side.

Once the global interrupts are enabled on the NVIC, each interrupt can be enabled separately via its corresponding enable bit.

Clock configuration

Both OCTOSPI1 and OCTOSPI2 peripherals have the same clock source. For XSPI1 and XSPI2 peripherals can have the same or different clock sources. Each peripheral has its dedicated prescaler allowing the application to connect two different memories running at different speeds. The following formula shows the relationship between OCTOSPI /HSPI/XSPI clock and the prescaler.

$$OCTOSPIx/HSPI/XSPIx\_CLK = F_{Clock\_source} / (PRESCALER + 1)$$

For instance, when the PRESCALER[7:0] is set to 2, OCTOSPIx/HSPI/XSPIx_CLK = F_{Clock_source} / 3.

In STM32L4+ and STM32L5 series devices, any of the three different clock sources, (SYSCLK, MSI or PLLQ) can be used for OCTOSPI clock source.

In STM32U5 devices, any of the four clock sources, (SYSCLK, pll1_q_ck, pll2_q_ck, pll3_r_ck) can be used for HSPI clock source and any of the four different clock sources (SYSCLK, MSIK, pll1_q_ck, pll2_q_ck) can be used for OCTOSPI clock source.

In STM32H7A3/7B3/7B0 and STM32H72x/73x devices, any of the three different clock sources, (rcc_hclk3, pll1_q_ck, pll2_r_ck, per_ck) can be used for OCTOSPI clock source.

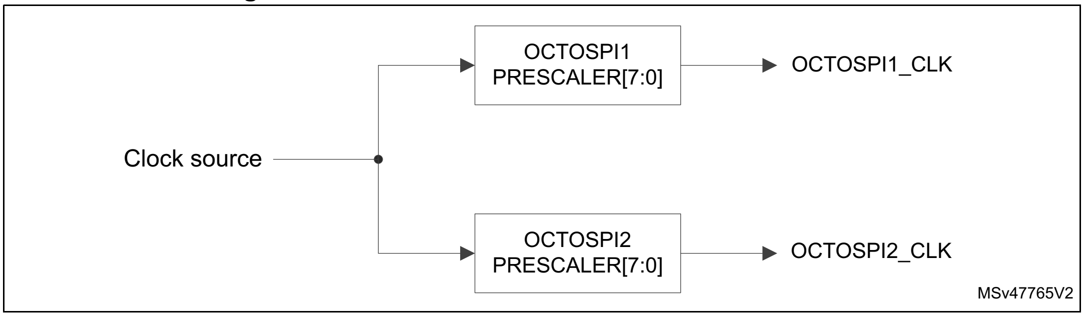
In STM32H7Rx/Sx, any of the three different clock sources, (hclk5, pll2_s_ck, pll2_t_ck) can be used for each XSPI clock source.

The OCTOSPI/HSPI/XSPI kernel clock and system clock can be completely asynchronous: as example, when selecting the HSI source clock for system clock and the MSI source clock for OCTOSPI kernel clock.

Note: The user must consider the frequency drift when using the MSI or HSI oscillator. Refer to relevant datasheet for more details on MSI and HSI oscillator frequency drift.

The figure below illustrates the OCTOSPI1 and OCTOSPI2 clock scheme.

Figure 14. OCTOSPI1 and OCTOSPI2 clock scheme



Note: In STM32L5 and STM32U535/545 series, only OCTOSPI1 is supported.

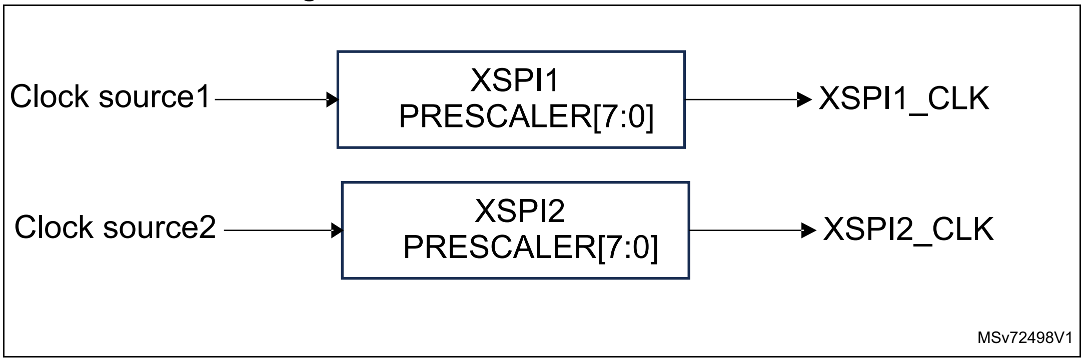
The figure below illustrates the HSPI1 clock scheme.

Figure 15. HSPI clock scheme

Note: The HSPI interface is available only in STM32U59x/5Ax and STM32U5Fx/5Gx series.

The figure below illustrates the XSPI1 and XSPI2 clock scheme.

Figure 16. XSPI1 and XSPI2 clock scheme



Note: The XSPI interface is available only in STM32H7Rx/Sx series.

5.2 OCTOSPI/HSPI/XSPI configuration for Regular-command protocol

The Regular-command protocol must be used when an external Single-SPI, Dual-SPI, Quad-SPI, Dual-quad-SPI, Octo-SPI, Dual-octal, or 16-bit memory is connected to the STM32.

The user must configure the following parameters:

- memory type: Micron, AP Memory, Macronix or Macronix RAM, or APmemory 16-bit mode.
- device size: number of bytes in the device = $2^{[DEVSIZE+1]}$
- chip-select high time (CSHT): must be configured according to the memory datasheet. CSHT is commonly named CS# Deselect Time and represents the period between two successive operations in which the memory is deselected.
- clock mode: low (Mode 0) or high (Mode 3)
- clock prescaler: must be set to get the targeted operating clock
- DHQC: recommended when writing to the memory. It shifts the outputs by a 1/4 OCTOSPI clock cycle and avoids hold issues on the memory side. For the HSPI and XSPI interfaces, it is managed by the PHY.
- SSHIFT: can be enabled when reading from the memory in SDR mode but must not be used in DTR mode. When enabled, the sampling is delayed by one more 1/2 OCTOSPI/HSPI/XSPI clock cycle enabling more relaxed input timings.
- CSBOUND: can be used to limit a transaction of aligned addresses in order to respect some memory page boundary crossing.
- REFRESH: used with PSRAM memories products to enable the refresh mechanism.

5.3 OCTOSPI, HSPI, and XSPI configuration for HyperBus protocol

The HyperBus protocol must be used when an external HyperRAM or HyperFlash memory is connected to the STM32.

The user must configure the following parameters:

- memory type: HyperBus
- device size: number of bytes in the device = $2^{[DEVSIZE+1]}$
- chip-select high time (CSHT): must be configured according to the memory datasheet. CSHT is commonly named CS# Deselect Time and represents the period between two successive operations in which the memory is deselected.
- clock mode low (Mode 0) or high (Mode 3)
- clock prescaler: must be set to get the targeted operating clock
- DTR mode: must be enabled for HyperBus
- DHQC: recommended when writing to the memory. It shifts the outputs by a 1/4 OCTOSPI/HSPI/XSPI clock cycle and avoids hold issues on the memory side.
- SSHIFT: must be disabled since HyperBus operates in DTR mode
- read-write recovery time (t_{RWR}): used only for HyperRAM and must be configured according to the memory device

- initial latency (t_{ACC}): must be configured according to the memory device and the operating frequency
- latency mode: fixed or variable latency
- latency on write access: enabled or disabled
- for HyperBus 16-bit mode, it is required to configure the DMODE[2:0] field
- CSBOUND: can be used to limit a transaction of aligned addresses in order to respect some memory page boundary crossing
- REFRESH: used with HyperRAM memories to enable the refresh mechanism

5.4 Memory configuration

The external memory device must be configured depending on the targeted operating mode. This section describes some commonly needed configurations for HyperBus/Regular mode, Octo-SPI/16-bit memories.

5.4.1 Memory device configuration

It is common that the application needs to configure the memory device. An example of commonly needed configurations is presented below:

1. Set the dummy cycles according to the operating speed (see relevant memory device datasheet).

Note: The dummy cycles of the STM32 MCUs is equivalent to latency term used by external memory side.

2. Enable the Octo-SPI mode or 16-bit mode.
3. Enable DTR mode.

Note: It is recommended to reset the memory device before the configuration. In order to reset the memory, a reset enable command then a reset command must be issued.

For Octo-SPI AP Memory device configuration, the delay block must be enabled to compensate the DQS skew. For detailed examples, refer to Section 6.

5.4.2 HyperBus memory device configuration

The HyperBus memory device contains the following addressable spaces:

- a register space
- a memory space

Before accessing the memory space for data transfers, the HyperBus memory device must be configured by accessing its register space when setting MTYP[2:0] = 0b101 in the OCTOSPI_DCR1 or HSPI_DCR1 or XSPI_DCR1 register.

When memory voltage range is 1.8 V, HyperBus requires differential clock and the NCLK pin must be configured.

Here below an example of HyperBus device parameters in the configuration register fields of the memory:

- Deep power-down (DPD) operation mode
- Initial latency count (must be configured depending on the memory clock speed)
- Fixed or variable latency
- Hybrid wrap option
- Wrapped burst length and alignment

6 OCTOSPI application examples

This section provides some typical OCTOSPI implementation examples with STM32L4P5xx/Q5xx products, and STM32CubeMX examples using the STM32L4P5G-DK Discovery kit for the STM32L4P5AGI6PU microcontroller.

6.1 Implementation examples

This section describes the following typical OCTOSPI use case examples:

- using OCTOSPI in a graphical application with Multiplexed mode
- code execution from Octo-SPI memory

6.1.1 Using OCTOSPI in a graphical application

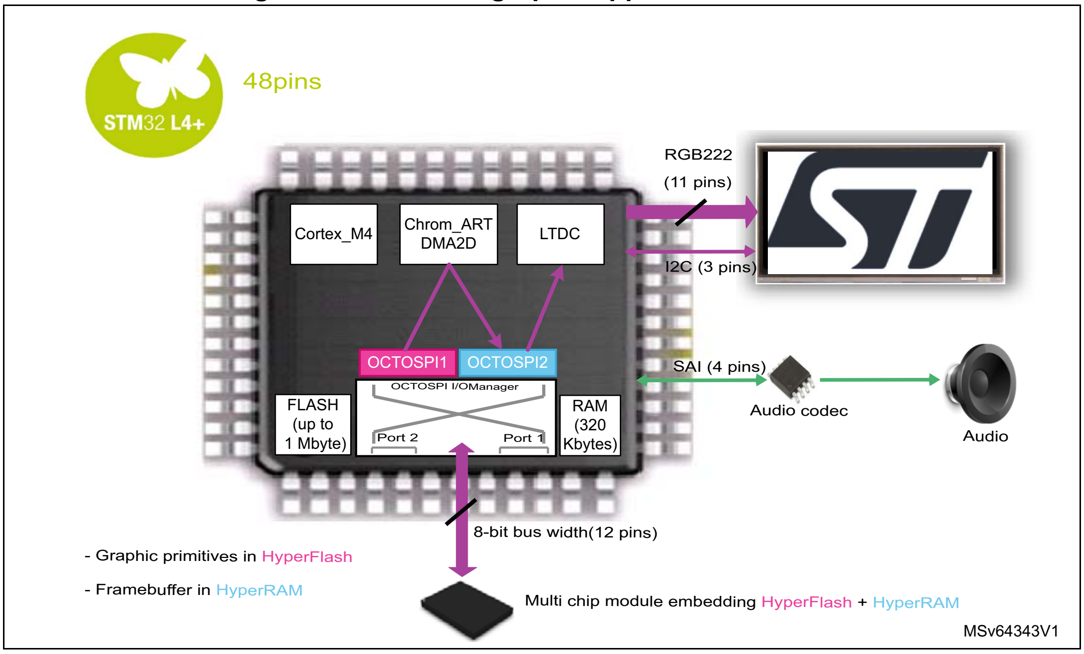
The STM32L4P5xx/Q5xx products embed two independent OCTOSPI peripherals that enable the connection of two external memories.For this example, the two external memories are Hyperbus memories.

This configuration is ideal for graphical applications on small packages, where:

- An HyperFlash memory is connected to OCTOSPI2 that is used to store graphical primitives.
- An HyperRAM memory is connected to OCTOSPI1 that is used to build frame buffer.
- Both OCTOSPI1 and OCTOSPI2 must be configured in HyperBus Memory-mapped mode, with Multiplexed mode enabled.
- Any AHB master (such as CPU, LTDC, DMA2D or SDMMC1/2) can autonomously access to both memories, exactly like an internal memory.

The figure below gives a use-case example of a multi-chip module connecting two HyperBus memories (HyperRAM and HyperFlash) over 12 pins (HyperBus single-ended clock) to a STM32L4Pxxx/Qxxx in LQFP48 package, for a graphical application with the OCTOSPI I/O manager Multiplexed mode enabled.

Figure 17. OCTOSPI graphic application use case



6.1.2 Executing from external memory: extend internal memory size

Using the external Octo-SPI memory permits to extend the available memory space for the total application.

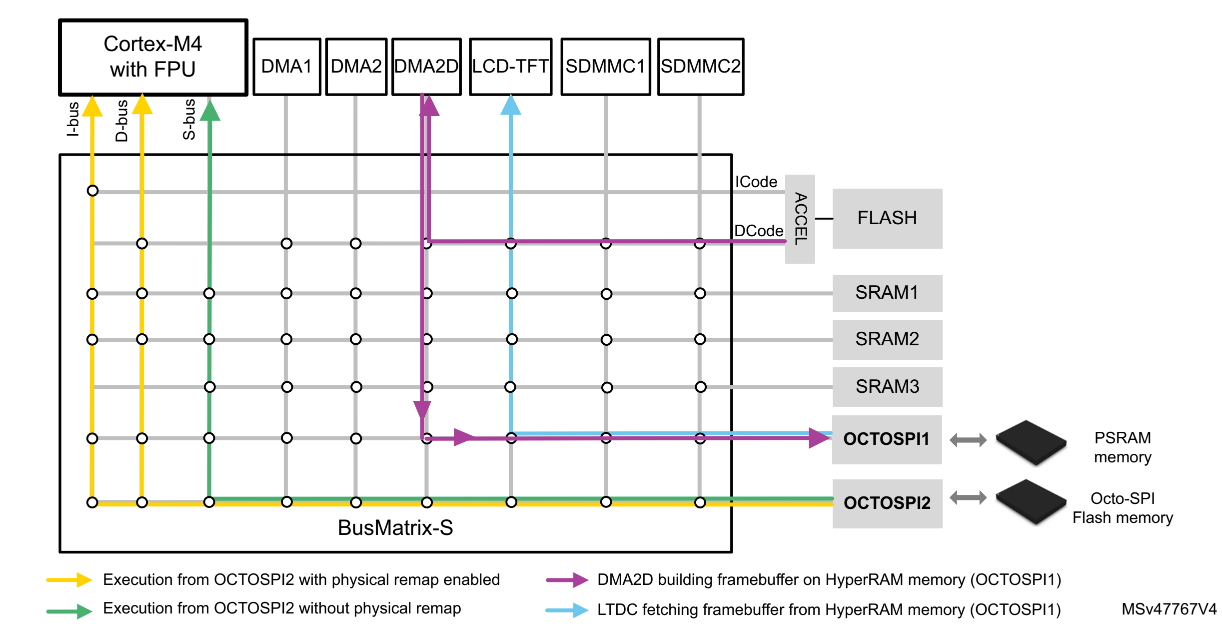
To execute code from an external memory, the following is needed:

- The application code must be placed in the external memory.
- The OCTOSPI must be configured in Memory-mapped mode during the system initialization before jumping to the Octo-SPI memory code.

As illustrated in the figure below, the CPU can execute code from the external memory connected to OCTOSPI2, while in parallel DMA2D and LTDC access to the memory connected to OCTOSPI1 for graphics.

By default OCTOSPI1 and OCTOSPI2 are accessed by the Cortex-M4 through S-bus. In order to boost execution performances, physical remap to 0x0000 0000 can be enabled for OCTOSPI2, allowing execution through I-bus and D-bus.

Figure 18. Executing code from memory connected to OCTOSPI2



6.2 OCTOSPI configuration with STM32CubeMX

This section shows examples of basic OCTOSPI configuration based on the STM32L4P5G-DK Discovery kit:

- Regular-command protocol in Indirect mode for programming and in Memory-mapped mode for reading from Octo-SPI flash memory
- Regular-command protocol in Memory-mapped mode for writing and reading from the Octo-SPI PSRAM
- Regular-command protocol in Memory-mapped mode for writing and reading from the Quad-SPI PSRAM
- Two HyperBus protocols in Memory-mapped mode multiplexed over the same bus for reading from HyperFlash and HyperRAM memories

Note: In order to reproduce the two HyperBus in Multiplexed mode example and the Quad-SPI PSRAM example, some modifications are required on the board, and related memories need to be soldered. For more details on STM32L4P5G-DK Discovery kit, refer to the user manual Discovery kit with STM32L4P5AG MCU (UM2651).

6.2.1 Hardware description

The STM32L4P5G-DK Discovery kit embeds the Octal-SPI Macronix Flash and the Octo-SPI AP Memory PSRAM. Thanks to the STM32L4P5G-DK PCB flexibility, it also allows the user to solder and test other memories:

- any Octo-SPI memory with the same footprint (BGA24)
- differential and single-ended clock memories (V_{DD} memory adjustable 1.8V or 3.3V)
- dual-die MCP memories (such Infineon HyperRAM + HyperFlash MCP)
- Quad-SPI memory on U14 footprint (SOP8)

For more details, refer to the user manual *Discovery kit with STM32L4P5AG MCU* (UM2651).

The next examples show how to configure the following memories using the STM32CubeMX:

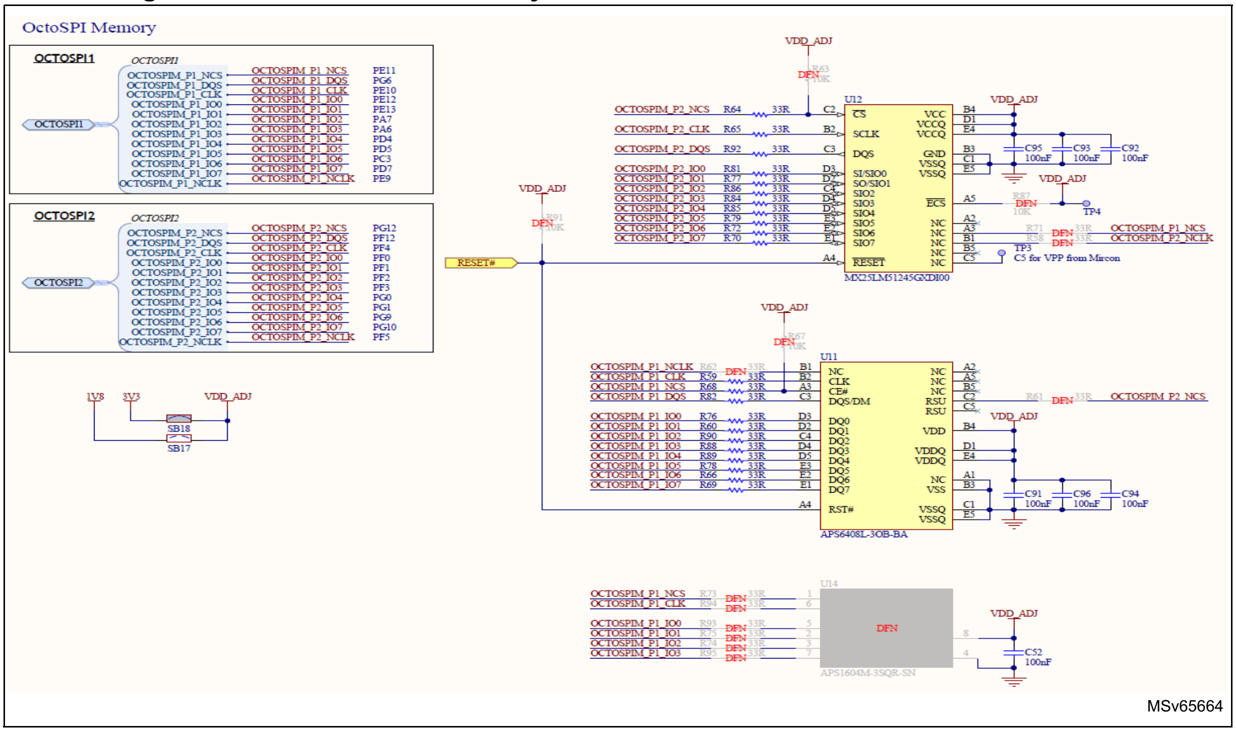
- Macronix MX25LM51245GXDI0A Octo-SPI flash memory connected to OCTOSPIM Port 2.
- AP Memory APS6408L-30B-BA Octo-SPI PSRAM memory connected to OCTOSPIM Port 1
- Infineon HyperBus MCP (S71KL256SC0) embedding HyperRAM and HyperFlash memories connected to OCTOSPIM Port 1 (in Multiplexed mode)
- AP Memory APS1604M-3SQR Quad-SPI PSRAM memory connected to OCTOSPIM Port 1

As shown in the figure below, the Octo-SPI Macronix Flash memory and the Octo-SPI AP Memory PSRAM are connected to the MCU, using each one of them eleven pins:

- OCTOSPI_NCS
- OCTOSPI_CLK
- OCTOSPI_DQS
- OCTOSPI_IO[0..7]

The OCTOSPI_RESET pin, connected to the global MCU reset pin (NRST), can be used to reset the memory.

Figure 19. Octo-SPI flash memory and PSRAM connection on STM32L4P5G-DK



Note: To test the HyperBus MCP Infineon memory S71KL256SC0, it must replace one of the existing Octo-SPI Macronix or Octo-SPI AP Memory.
To test the AP Memory APS1604M-3SQR Quad-SPI PSRAM memory, it must be soldered in the U14 footprint position. For more details, refer to the user manual *Discovery kit with STM32L4P5AG MCU* (UM2651).

6.2.2 Use case description

The adopted configuration for each example is the following:

- Octo-SPI AP Memory PSRAM:
  - OCTOSPI1 signals mapped to Port 1 (AP Memory PSRAM), so OCTOSPI1 must be set to Regular-command protocol
  - DTR Octo-SPI mode (with DQS) with OCTOSPI1 running at 60 MHz
  - Write/read in Memory-mapped mode
- Octo-SPI Macronix Flash:
  - OCTOSPI2 signals mapped to Port 2 (nor Macronix Flash), so OCTOSPI2 must be set to Regular-command protocol
  - DTR Octo-SPI mode (with DQS) with OCTOSPI2 running at 60 MHz
  - Programming the memory in Indirect mode and reading in Memory-mapped mode
- Quad-SPI AP Memory PSRAM:
  - OCTOSPI1 signals mapped to Port 1 (AP Memory PSRAM), so OCTOSPI1 must be set to Regular-command protocol
  - STR Quad-SPI mode with OCTOSPI1 running at 60 MHz
  - Write/read in Memory-mapped mode
- Infineon MCP HyperFlash and HyperRAM:
  - OCTOSPI1 and OCTOSPI2 signals muxed over Port 1, OCTOSPIM_P1_NCS used to access HyperRAM and OCTOSPIM_P2_NCS used to access HyperFlash. OCTOSPI1 and OCTOSPI2 must configured in HyperBus command protocol Multiplexed mode.
  - OCTOSPI1 and OCTOSPI2 with HyperBus protocol running at 60 MHz
  - CPU and DMA reading in Memory-mapped mode in concurrence with Multiplexed mode from the two external memories

The examples described later on the Regular-command and HyperBus protocols for OCTOSPI1 and OCTOSPI2, are based on STM32CubeMX:

- GPIO and OCTOSPI I/O manager configuration
- Interrupts and clock configuration

Each example has the following specific configurations:

- OCTOSPI peripheral configuration
- Memory device configuration

6.2.3 OCTOSPI GPIOs and clocks configuration

This section describes the needed steps to configure the OCTOSPI1 and OCTOSPI2 GPIOs and clocks.

In this section, figures describe the steps to follow and the tables contain the exact configuration to be used in order to run the example

I. STM32CubeMX: GPIOs configuration

Referring to *Figure 19*, the STM32CubeMX configuration examples are based on the connection detailed in the table below.

Table 6. STM32CubeMX - Memory connection port

Memory	OCTOSPI I/O manager port
Octo-SPI PSRAM AP Memory(1) APS6408L-30B-BA	Port 1
Octo-SPI Macronix Flash(1) MX25LM51245GXDI00	Port 2
HyperBus MCP Infineon memory(2) S71KL256SC0 Including both HyperRAM and HyperFlash	Multiplexed over Port 1: – OCTOSPIM_P1_NCS connected to HyperRAM – OCTOSPIM_P2_NCS connected to HyperFlash
Quad-SPI PSRAM AP Memory(3) APS1604M-3SQR	Port 1

1. Already Available on STM32L4P5G-DK Discovery kit.
2. Soldered in U11 (see *Figure 19*).
3. Soldered in U14 (see *Figure 19*).

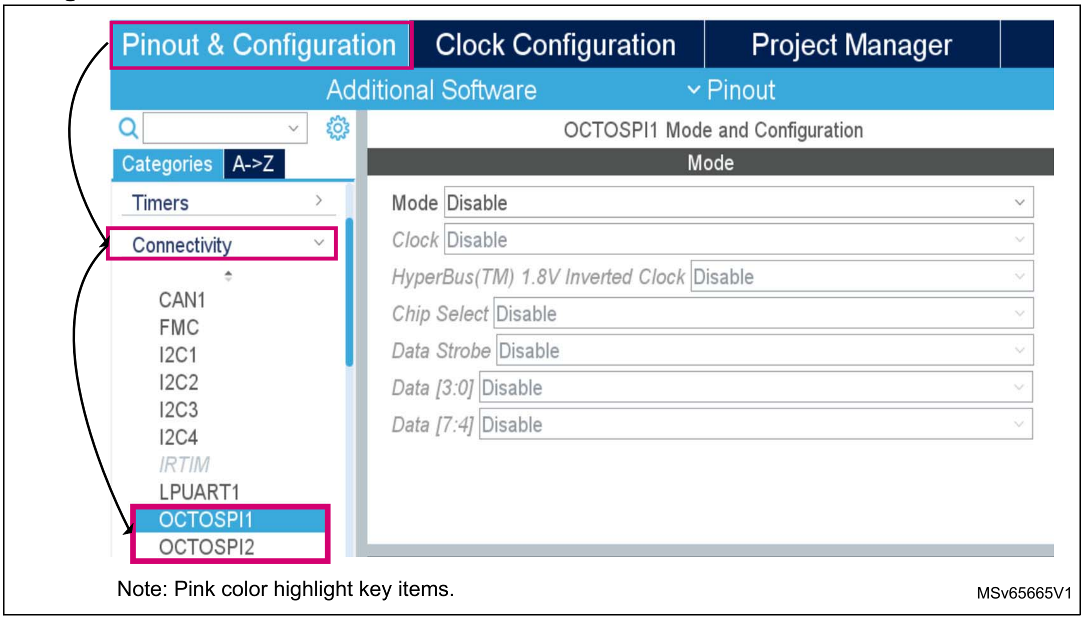
Based on this hardware implementation, the user must configure all the GPIOs shown in *Figure 19*.

STM32CubeMX: OCTOSPI GPIOs configuration

Once the STM32CubeMX project is created for the STM32L4P5AG product, the user must follow the steps below:

1. Select the *Pinout and Configuration* tab and, under *Connectivity,* uncollapse the OCTOSPI1 or OCTOSPI2 as shown in the figure below, then configure it by referencing to *Table 7*.

Figure 20. STM32CubeMX - Octo-SPI mode window for OCTOSPI1 or OCTOSPI2



2. Depending on the memory used, configure the OCTOSPI signals and mode as detailed in the following table.

Table 7. STM32CubeMX - Configuration of OCTOSPI signals and mode

Parameter	Memory				
	HyperBus MCP Infineon memory S71KL256SC0(1)		Octo-SPI PSRAM AP Memory APS6408L-30B -BA	Octo-SPI Flash Macronix MX25LM51245 GXDI00	Quad-SPI PSRAM AP Memory APS1604M -3SQR
Instance	OCTOSPI1	OCTOSPI2	OCTOSPI1	OCTOSPI2	OCTOSPI1
Mode	HyperBus --Multiplexed-		Octo-SPI		Quad-SPI
Clock	Port 1 CLK --Multiplexed-		Port 1 CLK	Port 2 CLK	Port 1 CLK
HyperBus 1.8 inverted clock	Disable				
Chip select	Port 1 NCS	Port 2 NCS	Port 1 NCS	Port 2 NCS	Port 1 NCS

Table 7. STM32CubeMX - Configuration of OCTOSPI signals and mode (continued)

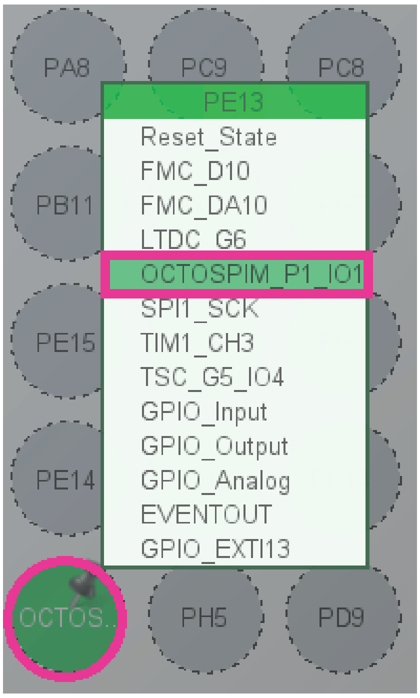
Parameter	Memory			
	HyperBus MCP Infineon memory S71KL256SC0[(1)]	Octo-SPI PSRAM AP Memory APS6408L-30B -BA	Octo-SPI Flash Macronix MX25LM51245 GXDI00	Quad-SPI PSRAM AP Memory APS1604M -3SQR
Data strobe	Port 1 DQS (RWDS) --MULTIPLEXED-	Port 1 DQS (RWDS)	Port 2 DQS (RWDS)	Disable
Data[3:0]	Port 1 IO[3:0] --MULTIPLEXED-	Port 1 IO[3:0]	Port 2 IO[3:0]	Port 1 IO[3:0]
Data[7:4]	Port 1 IO[7:4] --MULTIPLEXED-	Port 1 IO[7:4]	Port 2 IO[7:4]	Disable

1. This configuration provides access to both of HyperFlash and HyperRAM memories in Multiplexed mode.

The configured GPIOs must match the memory connection as shown in *Figure 19*. If the configuration is not correct, the user must manually configure all the GPIOs, one by one, by clicking directly on each pin.

The figure below shows how to configure manually the PE13 pin to OCTOSPIM_P1_IO1 alternate function (AF).

Figure 21. STM32CubeMX - Setting PE13 pin to OCTOSPIM_P1_IO1 AF

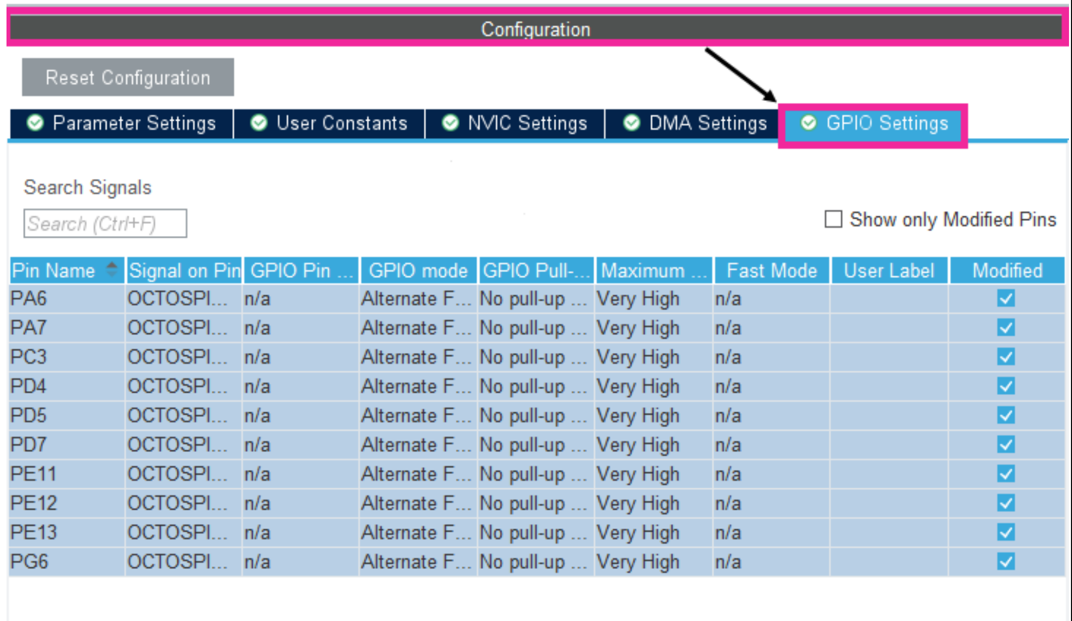


Note: Pink color highlights key items.

MSv65666V2

3. Configuring OCTOSPI GPIOs to very-high speed:
   a) Depending on the selected instance OCTOSPI1 or OCTOSPI2, in the *Configuration* window, select the GPIO settings tab as shown in the figure below:

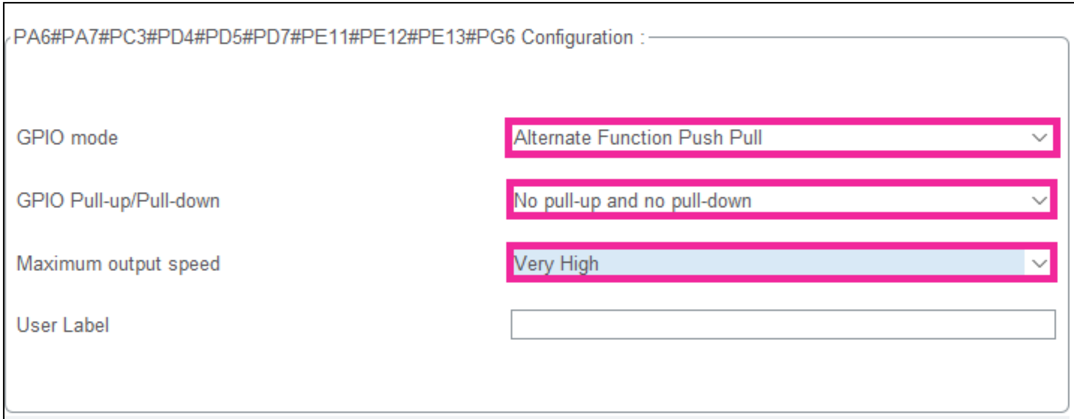
Figure 22. STM32CubeMX - GPIOs setting window



Note: Pink color highlights key items. The same steps are needed if OCTOSPI2 is used. MSv65667V2

   b) Scroll down the window and make sure that the output speed is set to "very high" for all the GPIOs.

Figure 23. STM32CubeMX - Setting GPIOs to very-high speed



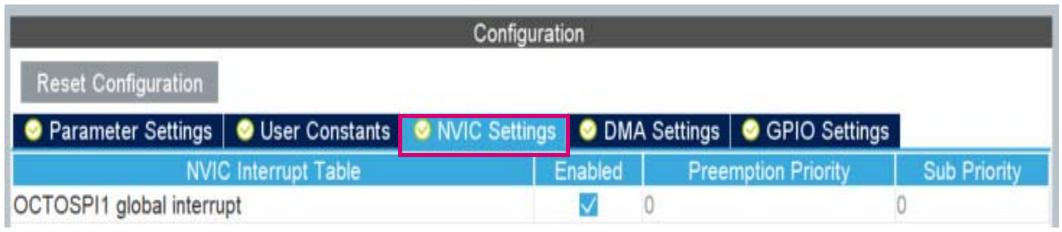
Note: Pink color highlights key items. The same steps are needed if OCTOSPI2 is used. MSv65668V2

II. STM32CubeMX: Enabling interrupts

As previously described in *Section 5.1.2: Interrupts and clocks configuration*, each OCTOSPI peripheral has its dedicated global interrupt connected to the NVIC, so each peripheral interrupt must be enabled separately.

Depending on the selected instance OCTOSPI1 or OCTOSPI2, In the OCTOSPI *Configuration* window (see the figure below), select the NVIC settings tab then check the OCTOSPI global interrupts.

Figure 24. STM32CubeMX - Enabling OCTOSPI global interrupt



Note: Pink color highlight key items.
Same steps needed if OCTOSPI2 is used.

MSv65669V1

III. STM32CubeMX: clocks configuration

In this example, the system clock is configured as shown below:

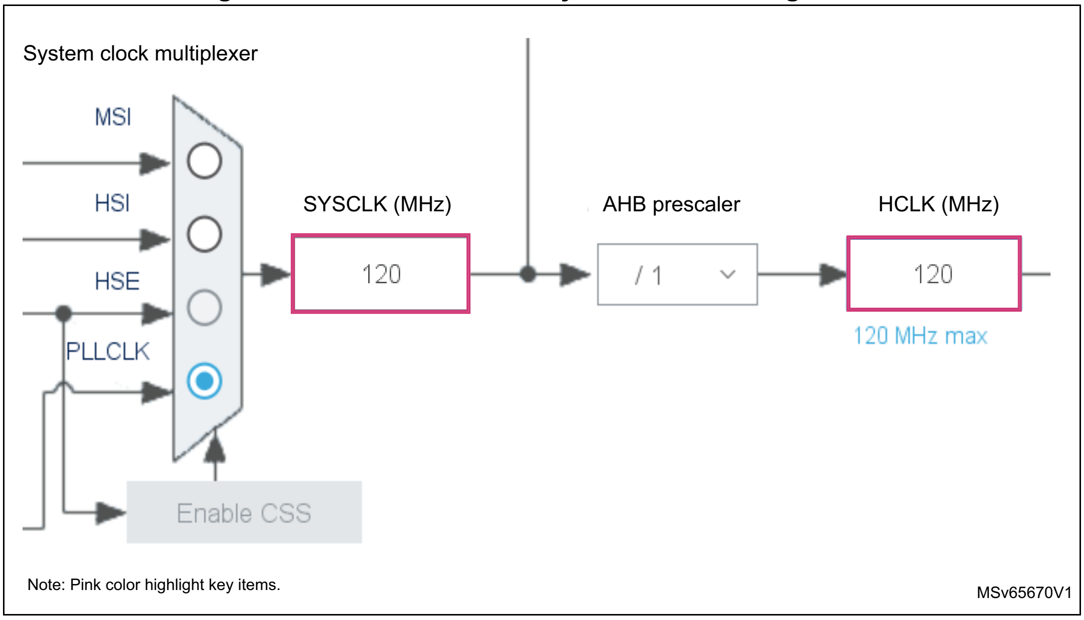
- Main PLL is used as system source clock.
- SYSCLK and HCLK set to 120 MHz, so Cortex-M4 and AHB operate at 120 MHz.

  As previously described in *Section 5.1.2: Interrupts and clocks configuration*, both OCTOSPI peripherals have the same clock source, but each one has its dedicated prescaler allowing the connection of two memories running at different speeds.

  In this example, the SYSCLK is used as clock source for OCTOSPI1 and OCTOSPI2.

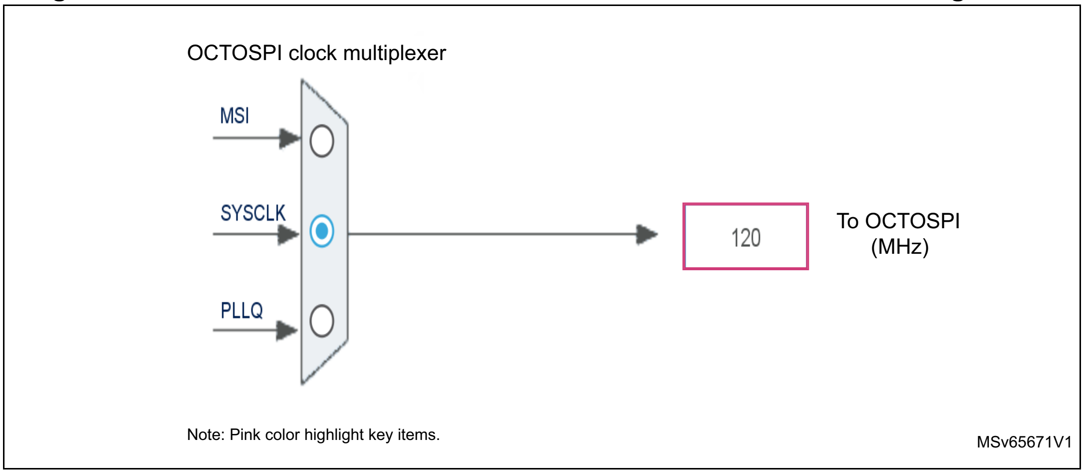
- System clock configuration:
  a) Select the clock configuration tab.
  b) In the *Clock configuration* tab, set the PLLs and the prescalers to get the system clock at 120 MHz as shown in the figure below.

Figure 25. STM32CubeMX - System clock configuration



- OCTOSPI clock source configuration: In the *Clock configuration* tab, select the SYSCLK clock source (see the figure below).

Figure 26. STM32CubeMX - OCTOSPI1 and OCTOSPI2 clock source configuration



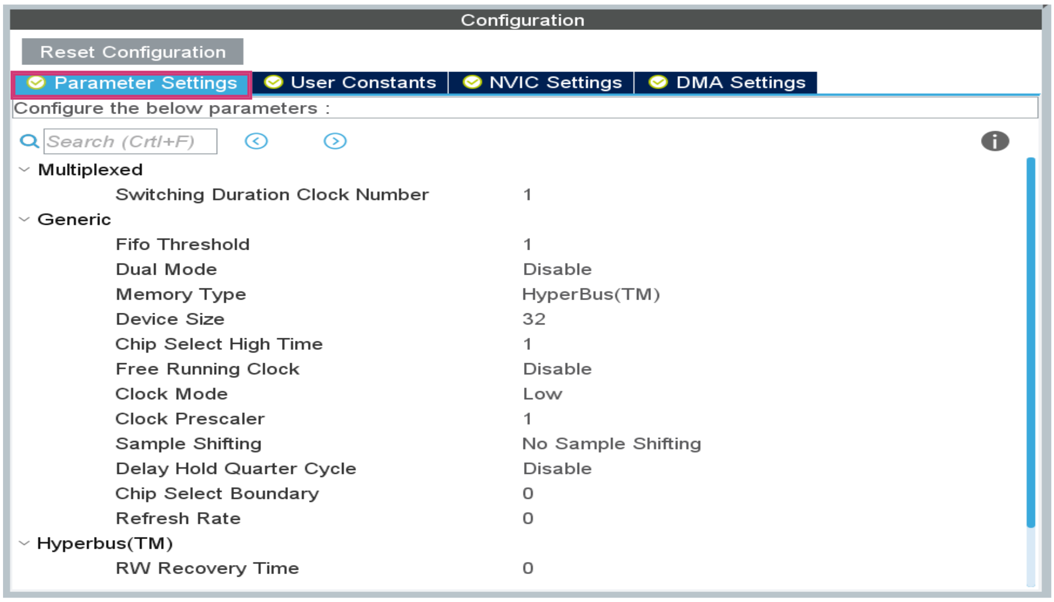
With this configuration, OCTOSPI1 and OCTOSPI2 are clocked by SYSCLK@120 MHz. Then, for each peripheral, the prescaler is configured to get the 60 MHz targeted speed (see *Section 6.2.4: OCTOSPI configuration and parameter settings*).

6.2.4 OCTOSPI configuration and parameter settings

Once all of the OCTOSPI GPIOs and the clock configuration have been done, the user must configure the OCTOSPI depending on the used external memory and its communication protocol.

1. In the OCTOSPI *Configuration* window, select the *Parameter Settings* tab as shown in the figure below and configure it by referencing to *Table 8*.

Figure 27. STM32CubeMX - OCTOSPI configuration window



Note: Pink color highlight key items.

MSv65672V1

2. Configure the OCTOSPI parameters depending on the memory used.

Table 8. STM32CubeMX - Configuration of OCTOSPI parameters

OCTOSPI parameter	Memory				
	HyperBus MCP Infineon memory S71KL256SC0		Octal-SPI PSRAM AP Memory APS6408L-30B -BA	Octal-SPI Flash Macronix MX25LM51245 GXDI00	Quad-SPI PSRAM AP Memory APS1604M -3SQR
Instance	OCTOSPI1	OCTOSPI2	OCTOSPI1	OCTOSPI2	OCTOSPI1
Multiplexed					
Switching duration clock number(1)	1		N/A		
Generic					
FIFO threshold	1				
Dual mode	Disable				
Memory type	HyperBus		AP Memory	Macronix	Micron(2)

Table 8. STM32CubeMX - Configuration of OCTOSPI parameters (continued)

<table>
<tr><th rowspan="2">OCTOSPI parameter</th><th colspan="5">Memory</th></tr>
<tr><th colspan="2">HyperBus MCP Infineon memory S71KL256SC0</th><th>Octal-SPI PSRAM AP Memory APS6408L-30B -BA</th><th>Octal-SPI Flash Macronix MX25LM51245 GXDI00</th><th>Quad-SPI PSRAM AP Memory APS1604M -3SQR</th></tr>
<tr><td>Device Size[3]</td><td>23 (8 Mbytes)</td><td>25 (32 Mbytes)</td><td>23 (8 Mbytes)</td><td>26 (64 Mbytes)</td><td>21 (2 Mbytes)</td></tr>
<tr><td>Chip select high time[4]</td><td colspan="3">1</td><td>3</td><td>2</td></tr>
<tr><td>Free running clock</td><td colspan="5">Disable</td></tr>
<tr><td>Clock mode</td><td colspan="5">Low</td></tr>
<tr><td>Clock prescaler[5]</td><td colspan="5">2</td></tr>
<tr><td>Sample shifting[6]</td><td colspan="4">No sample shifting</td><td>Sample shifting half-cycle</td></tr>
<tr><td>Delay hold quarter cycle[7]</td><td colspan="4">Enable</td><td>Disable</td></tr>
<tr><td>Chip select boundary[8]</td><td colspan="2">0</td><td>10 (1 Kbyte)</td><td colspan="2">0</td></tr>
<tr><td>Delay block</td><td colspan="5">Enable</td></tr>
<tr><td>Maximum transfer</td><td colspan="2">0</td><td colspan="3">N/A</td></tr>
<tr><td>Refresh rate[9]</td><td>241 (4 µs)</td><td>0</td><td>241 (4 µs)</td><td>0</td><td>482 (8 µs)</td></tr>
<tr><th colspan="6">HyperBus</th></tr>
<tr><td>RW recovery time[10]</td><td colspan="2">3</td><td colspan="3" rowspan="4">N/A</td></tr>
<tr><td>Access time[11]</td><td colspan="2">6</td></tr>
<tr><td>Write zero latency[12]</td><td colspan="2">Enable</td></tr>
<tr><td>Latency mode[13]</td><td colspan="2">Fixed</td></tr>
</table>

1. Switching duration clock number (REQ2ACK_TIME) defining, in Multiplexed mode, the time between two transactions. The value is the number of the OCTOSPI clock cycles.
2. The memory type has no impact in Quad-SPI mode.
3. Device Size defines the memory size in number of bytes = $2^{(DEVSIZE+1)}$.
4. Chip select high time (CSHT) defines the chip-select minimum high time in number of clock cycles, configured depending on the memory datasheet.
5. The system clock prescaler (120MHz) / clock prescaler (2 MHz) = OCTOSPI clock frequency (60MHz).
6. Sample shifting (SSHT) recommended to be enabled in STR mode and disabled in DTR mode.
7. Delay hold quarter cycle (DHQC) enabled in DTR mode and disabled in STR mode.
8. Chip select boundary (CSBOUND) configured depending on the memory datasheet. The chip select must go high when crossing the page boundary ($2^{CSBOUND}$ bytes defines the page size).
9. Refresh rate (REFRESH) required for PSRAMs memories. The chip select must go high each (REFRECH x OCTOSPI clock cycles), configured depending on the memory datasheet.
10. Read/write recovery time (TRWR) define the device read/write recovery time, expressed in number of OCTOSPI clock cycle, configured depending on the memory datasheet.
11. Access time (TACC) is expressed in number of OCTOSPI clock cycles, configured depending on the memory datasheet.
12. Write zero latency enabled (WZL) defines the latency on write accesses.
13. The latency mode (LM) is configured to fixed latency, depending on the memory datasheet.

3. Build and run the project: At this stage, the user can build, debug and run the project.

6.2.5 STM32CubeMX: Project generation

Once all of the GPIOs, the clock and the OCTOSPI configurations have been done, the user must generate the project with the desired toolchain (such as STM32CubeIDE, EWARM or MDK-ARM).

Indirect and Memory-mapped mode configuration

At this stage, the project must be already generated with GPIOs and OCTOSPI properly configured following the steps detailed in *Section 6.2.3* and*Section 6.2.4*.

I. Octo-SPI PSRAM in Regular-command protocol example

In order to configure the OCTOSPI1 in Memory-mapped mode and to configure the external Octo-SPI PSRAM AP Memory allowing communication in DTR Octo-SPI mode (with DQS), some functions must be added to the project. Code can be added to the *main.c* file (see code below) or defines can be added to the *main.h* file (see *Adding defines to the main.h file*).

- Adding code to the *main.c* file

Open the already generated project and follow the steps described below:

Note: Update the main.c file by inserting the lines of code to include the needed functions in the adequate space indicated in green bold below. This task avoids loosing the user code in case of project regeneration.

a) Insert variables declarations in the adequate space (in green bold below).

```
/* USER CODE BEGIN PV */
/*buffer that we will write n times to the external memory, user can modify
the content to write his desired data*/
/* Private variables ------------------------------------------------*/
uint8_t aTxBuffer[] = " **OCTOSPI/Octal-spi PSRAM Memory-mapped
communication example** **OCTOSPI/Octal-spi PSRAM Memory-mapped
communication example** **OCTOSPI/Octal-spi PSRAM Memory-mapped
communication example** **OCTOSPI/Octal-spi PSRAM Memory-mapped
communication example**";
/* USER CODE END PV */
```

b) Insert the functions prototypes in the adequate space (in green bold below).

```
/* USER CODE BEGIN PFP */
/* Private function prototypes ----------------------------------------*/
void EnableMemMapped(void);
void DelayBlock_Calibration(void);
/* USER CODE END PFP */
```

c) Insert the functions to be called in the `main()` function, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 1 */
__IO uint8_t *mem_addr;
uint32_t address = 0;
uint16_t index1;/*index1 counter of bytes used when reading/
                  writing 256 bytes buffer */
uint16_t index2;/*index2 counter of 256 bytes buffer used when reading/
                  writing the 1Mbytes extended buffer */
/* USER CODE END 1 */

/* USER CODE BEGIN 2 */
/*----------------------------------------------------------------------*/
/*Enable Memory Mapped Mode*/
EnableMemMapped();
/*----------------------------------------------------------------------*/
/*Enable the Delay Block Calibration*/
DelayBlock_Calibration();
/*----------------------------------------------------------------------*/
/* Writing Sequence of 1Mbyte */
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE + address);
/*Writing 1Mbyte (256Byte BUFFERSIZE x 4096 times) */
for (index2 = 0; index2 < EXTENDEDBUFFERSIZE/BUFFERSIZE; index2++)
{
for (index1 = 0; index1 < BUFFERSIZE; index1++)
{
*mem_addr = aTxBuffer[index1];
mem_addr++;
}
}
/*----------------------------------------------------------------------*/
/* Reading Sequence of 1Mbyte */
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE + address);
/*Reading 1Mbyte (256Byte BUFFERSIZE x 4096 times)*/
for (index2 = 0; index2 < EXTENDEDBUFFERSIZE/BUFFERSIZE; index2++)     {
for (index1 = 0; index1 < BUFFERSIZE; index1++)
{
if (*mem_addr != aTxBuffer[index1])
{
/*if data read is corrupted we can toggle a led here: example blue led*/
}
mem_addr++;
```

```
}
}
/*if data read is correct we can toggle a led here: example green led*/

/* USER CODE END 2 */
```

d) Insert the function definitions, called in the `main()`, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 4 */
/*----------------------------------------------------------------------*/
/* This function enables memory-mapped mode for Read and Write operations */
void EnableMemMapped(void)
{
OSPI_RegularCmdTypeDef sCommand;
OSPI_MemoryMappedTypeDef sMemMappedCfg;
sCommand.FlashId              = HAL_OSPI_FLASH_ID_1;
sCommand.InstructionMode      = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize      = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.AddressMode          = HAL_OSPI_ADDRESS_8_LINES;
sCommand.AddressSize          = HAL_OSPI_ADDRESS_32_BITS;
sCommand.AddressDtrMode       = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode             = HAL_OSPI_DATA_8_LINES;
sCommand.DataDtrMode          = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.DQSMode              = HAL_OSPI_DQS_ENABLE;
sCommand.SIOOMode             = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.Address              = 0;
sCommand.NbData               = 1;
/* Memory-mapped mode configuration for Linear burst write operations */
sCommand.OperationType = HAL_OSPI_OPTYPE_WRITE_CFG;
sCommand.Instruction    = LINEAR_BURST_WRITE;
sCommand.DummyCycles    = DUMMY_CLOCK_CYCLES_SRAM_WRITE;
if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Memory-mapped mode configuration for Linear burst read operations */
sCommand.OperationType = HAL_OSPI_OPTYPE_READ_CFG;
sCommand.Instruction    = LINEAR_BURST_READ;
sCommand.DummyCycles    = DUMMY_CLOCK_CYCLES_SRAM_READ;
```

```
if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/*Disable timeout counter for memory mapped mode*/
sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_DISABLE;
/*Enable memory mapped mode*/
if (HAL_OSPI_MemoryMapped(&hospi1, &sMemMappedCfg) != HAL_OK)
{
Error_Handler();
}
}
/*----------------------------------------------------------------------*/
/*This function is used to calibrate the Delayblock before initiating
USER's application read/write transactions*/
void DelayBlock_Calibration(void)
{
/*buffer used for calibration*/
uint8_t Cal_buffer[] = " ****Delay Block Calibration Buffer****  ****Delay
Block Calibration Buffer****  ****Delay Block Calibration Buffer****
****Delay Block Calibration Buffer****  ****Delay Block Calibration
Buffer****  ****Delay Block Calibration Buffer**** ";
uint16_t index;
__IO uint8_t *mem_addr;
uint8_t test_failed;
uint8_t delay = 0x0;
uint8_t Min_found = 0;
uint8_t Max_found = 0;
uint8_t Min_Window = 0x0;
uint8_t Max_Window = 0xF;
uint8_t Mid_window = 0;
uint8_t calibration_ongoing = 1;
/* Write the Cal_buffer to the memory*/
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE);
for (index = 0; index < DLYB_BUFFERSIZE; index++)
{
*mem_addr = Cal_buffer[index];
mem_addr++;
}
while (calibration_ongoing)
{
/* update the Delayblock calibration */
HAL_RCCEx_OCTOSPIDelayConfig(delay, 0);
test_failed = 0;
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE);
```

```
for (index = 0; index < DLYB_BUFFERSIZE; index++)
{
/* Read the Cal_buffer from the memory*/
if (*mem_addr != Cal_buffer[index])
{
/*incorrect data read*/
test_failed = 1;
}
mem_addr++;
}
/*          search for the Min window       */
if (Min_found!=1)
{
if (test_failed == 1)
{
if (delay < 15)
{
delay++;
}
else
{
/* If delay set to maximum and error still detected: can't use external
PSRAM */
Error_Handler();
}
}
else
{
Min_Window = delay;
Min_found=1;
delay = 0xF;
}
}
/*             search for the Max window         */
else if (Max_found!=1)
{
if (test_failed == 1)
{
if (delay > 0)
{
delay--;
}
else
{
```

```
/* If delay set to minimum and error still detected: can't use external
PSRAM */
Error_Handler();
}
}
else
{
Max_Window = delay;
Max_found=1;
}
}
/* min and max delay window found, configure the delay block with the middle
window value and exit calibration */
else
{
Mid_window = (Max_Window+Min_Window)/2;
HAL_RCCEx_OCTOSPIDelayConfig(Mid_window, 0);
/* exit calibration */
calibration_ongoing = 0;
}
}
}
/* USER CODE END 4 */
```

- Adding defines to the *main.h* file

Update the *main.h* file by inserting the defines in the adequate space (in green bold below).

```
/* USER CODE BEGIN Private defines */
/*APS6408L-3OB PSRAM APmemory*/
#define LINEAR_BURST_READ                          0x20
#define LINEAR_BURST_WRITE                         0xA0
#define DUMMY_CLOCK_CYCLES_SRAM_READ              5
#define DUMMY_CLOCK_CYCLES_SRAM_WRITE             4
/* Exported macro -----------------------------------------------------*/
#define BUFFERSIZE (COUNTOF(aTxBuffer) - 1)
#define COUNTOF(__BUFFER__) (sizeof(__BUFFER__) / sizeof(*(__BUFFER__)))
#define DLYB_BUFFERSIZE (COUNTOF(Cal_buffer) - 1)
#define EXTENDEDBUFFERSIZE                 (1048576)
/* USER CODE END Private defines */
```

- Build and run the code.

II. Octo-SPI FLASH in Regular-command protocol example

In order to configure the OCTOSPI2 in Indirect/Memory-mapped mode and to configure the external Octo-SPI Macronix Flash memory allowing communication in DTR Octo-SPI mode (with DQS), some functions must be added to the project. Code can be added to the *main.c* file (see code below) or defines can be added to the *main.h* file (see *Adding defines to the main.h file*).

- Adding code to the main.c file

Open the already generated project and follow the steps described below:

Note: Update the main.c file by inserting the lines of code to include the needed functions in the adequate space indicated in green bold below. This task avoids loosing the user code in case of project regeneration.

a) Insert variables declarations in the adequate space (in green bold below).

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------*/
uint8_t aTxBuffer[]=" Programming in indirect mode - Reading in memory-mapped mode ";
__IO uint8_t *nor_memaddr = (__IO uint8_t *)(OCTOSPI2_BASE);
__IO uint8_t aRxBuffer[BUFFERSIZE] ="";
/* USER CODE END PV */
```

b) Insert the functions prototypes in the adequate space (in green bold below).

```
/* USER CODE BEGIN PFP */
/* Private function prototypes ---------------------------------------*/
void WriteEnable(void);
void OctalWriteEnable(void);
void OctalDTR_MemoryCfg(void);
void OctalSectorErase(void);
void OctalDTR_MemoryWrite(void);
void AutoPollingWIP(void);
void OctalPollingWEL(void);
void OctalPollingWIP(void);
void EnableMemMapped(void);
/* USER CODE END PFP */
```

c) Insert the functions to be called in the `main()` function, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 1 */
uint16_t index1;
/* USER CODE END 1 */
```

```
/* USER CODE BEGIN 2 */
/*----------------------------------------------------------------------*/
/*-------------- MX25LM51245G memory configuration --------------*/
/* Configure MX25LM51245G memory to DTR Octal I/O mode */
OctalDTR_MemoryCfg();
/*----------------------------------------------------------------------*/
/*----------------------- Erasing the first sector ----------------------*/
/* Enable writing to memory using Octal Write Enable cmd */
OctalWriteEnable();
/* Enable Octal Software Polling to wait until WEL=1 */
OctalPollingWEL ();
/* Erasing first sector using Octal erase cmd */
OctalSectorErase();
/* Enable Octal Software Polling to wait until memory is ready WIP=0*/
OctalPollingWIP();
/*----------------------------------------------------------------------*/
/*--------------------- Programming operation ---------------------*/
/* Enable writing to memory using Octal Write Enable cmd */
OctalWriteEnable();
/* Enable Octal Software Polling to wait until WEL=1 */
OctalPollingWEL();
/* Writing (using CPU) the aTxBuffer to the memory */
OctalDTR_MemoryWrite();
/* Enable Octal Software Polling to wait until memory is ready WIP=0*/
OctalPollingWIP();
/*---------------------------------------------------------------------*/
/*------ Configure memory-mapped Octal SDR Read/write ------*/
EnableMemMapped();
/*----------------------------------------------------------------------*/
/*---------------- Reading from the NOR memory ------------------*/
for(index = 0; index < BUFFERSIZE; index++)
{
/* Reading back the written aTxBuffer in memory-mapped mode */
aRxBuffer[index] = *nor_memaddr;
if(aRxBuffer[index] != aTxBuffer[index])
{
/* Can add code to toggle a LED when data doesn't match */
}
nor_memaddr++;
}
/*----------------------------------------------------------------------*/
/* USER CODE END 2 */
```

d) Insert the function definitions, called in the `main()`, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 4 */
/* This function Enables writing to the memory: write enable cmd is sent in
single SPI mode */
void WriteEnable(void)
{
OSPI_RegularCmdTypeDef sCommand;
OSPI_AutoPollingTypeDef sConfig;
/* Initialize the Write Enable cmd in single SPI mode */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = WRITE_ENABLE_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_1_LINE;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.AddressMode = HAL_OSPI_ADDRESS_NONE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_NONE;
sCommand.DummyCycles = 0;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
/* Send Write Enable command in single SPI mode */
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Initialize Automatic-Polling mode to wait until WEL=1 */
sCommand.Instruction = READ_STATUS_REG_CMD;
sCommand.DataMode = HAL_OSPI_DATA_1_LINE;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_DISABLE;
sCommand.NbData = 1;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Set the mask to 0x02 to mask all Status REG bits except WEL */
/* Set the match to 0x02 to check if the WEL bit is set */
sConfig.Match = WRITE_ENABLE_MATCH_VALUE;
sConfig.Mask = WRITE_ENABLE_MASK_VALUE;
sConfig.MatchMode = HAL_OSPI_MATCH_MODE_AND;
sConfig.Interval = AUTO_POLLING_INTERVAL;
sConfig.AutomaticStop = HAL_OSPI_AUTOMATIC_STOP_ENABLE;
```

```
/* Start Automatic-Polling mode to wait until WEL=1 */
if (HAL_OSPI_AutoPolling(&hospi2, &sConfig, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
{
Error_Handler();
}
}
/* This functions Enables writing to the memory: write enable cmd is sent in
Octal SPI mode */
void OctalWriteEnable(void)
{
OSPI_RegularCmdTypeDef sCommand;
/* Initialize the Write Enable cmd */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = OCTAL_WRITE_ENABLE_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.AddressMode = HAL_OSPI_ADDRESS_NONE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_NONE;
sCommand.DummyCycles = 0;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
/* Send Write Enable command in Octal mode */
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
}
/* This function Configures Software polling to wait until WEL=1 */
void OctalPollingWEL(void)
{
OSPI_AutoPollingTypeDef sConfig;
OSPI_RegularCmdTypeDef sCommand;
/* Initialize Indirect read mode for Software Polling to wait until WEL=1 */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = OCTAL_READ_STATUS_REG_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.Address = 0x0;
sCommand.AddressMode = HAL_OSPI_ADDRESS_8_LINES;
```

```
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.AddressDtrMode = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_8_LINES;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.NbData = 2;
sCommand.DummyCycles = DUMMY_CLOCK_CYCLES_READ_REG;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
/* Set the mask to 0x02 to mask all Status REG bits except WEL */
/* Set the match to 0x02 to check if the WEL bit is Set */
sConfig.Match = WRITE_ENABLE_MATCH_VALUE;
sConfig.Mask = WRITE_ENABLE_MASK_VALUE;
sConfig.MatchMode = HAL_OSPI_MATCH_MODE_AND;
sConfig.Interval = 0x10;
sConfig.AutomaticStop = HAL_OSPI_AUTOMATIC_STOP_ENABLE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Start Automatic-Polling mode to wait until the memory is ready WEL=1 */
if (HAL_OSPI_AutoPolling(&hospi2, &sConfig, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
{
Error_Handler();
}
}
/* This function Configures Automatic-polling mode to wait until WIP=0 */
void AutoPollingWIP(void)
{

OSPI_RegularCmdTypeDef sCommand;
OSPI_AutoPollingTypeDef sConfig;
/* Initialize Automatic-Polling mode to wait until WIP=0 */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = READ_STATUS_REG_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_1_LINE;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.AddressMode = HAL_OSPI_ADDRESS_NONE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DummyCycles = 0;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
```

```
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.DataMode = HAL_OSPI_DATA_1_LINE;
sCommand.NbData = 1;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_DISABLE;
/* Set the mask to 0x01 to mask all Status REG bits except WIP */
/* Set the match to 0x00 to check if the WIP bit is Reset */
sConfig.Match = MEMORY_READY_MATCH_VALUE;
sConfig.Mask = MEMORY_READY_MASK_VALUE;
sConfig.MatchMode = HAL_OSPI_MATCH_MODE_AND;
sConfig.Interval = 0x10;
sConfig.AutomaticStop = HAL_OSPI_AUTOMATIC_STOP_ENABLE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Start Automatic-Polling mode to wait until the memory is ready WIP=0 */
if (HAL_OSPI_AutoPolling(&hospi2, &sConfig, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
{
Error_Handler();
}
}
/* This function Configures Software polling mode to wait the memory is
ready WIP=0 */
void OctalPollingWIP(void)
{

OSPI_RegularCmdTypeDef sCommand;
OSPI_AutoPollingTypeDef sConfig;
/* Initialize Automatic-Polling mode to wait until WIP=0 */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = OCTAL_READ_STATUS_REG_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.Address = 0x0;
sCommand.AddressMode = HAL_OSPI_ADDRESS_8_LINES;
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.AddressDtrMode = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_8_LINES;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.NbData = 2;
sCommand.DummyCycles = DUMMY_CLOCK_CYCLES_READ_REG;
```

```
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
/* Set the mask to 0x01 to mask all Status REG bits except WIP */
/* Set the match to 0x00 to check if the WIP bit is Reset */
sConfig.Match = MEMORY_READY_MATCH_VALUE;
sConfig.Mask = MEMORY_READY_MASK_VALUE;
sConfig.MatchMode = HAL_OSPI_MATCH_MODE_AND;
sConfig.Interval = 0x10;
sConfig.AutomaticStop = HAL_OSPI_AUTOMATIC_STOP_ENABLE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Start Automatic-Polling mode to wait until the memory is ready WIP=0 */
if (HAL_OSPI_AutoPolling(&hospi2, &sConfig, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
{
Error_Handler();
}
}
/*** This function configures the MX25LM51245G memory ***/
void OctalDTR_MemoryCfg(void)
{
 OSPI_RegularCmdTypeDef sCommand;
uint8_t tmp;
/* Enable writing to memory in order to set Dummy */
WriteEnable();
/* Initialize Indirect write mode to configure Dummy */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_1_LINE;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.Instruction = WRITE_CFG_REG_2_CMD;
sCommand.Address = CONFIG_REG2_ADDR3;
sCommand.AddressMode = HAL_OSPI_ADDRESS_1_LINE;
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.AddressDtrMode = HAL_OSPI_ADDRESS_DTR_DISABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_1_LINE;
sCommand.DataDtrMode= HAL_OSPI_DATA_DTR_DISABLE;
sCommand.NbData = 1;
sCommand.DummyCycles = 0;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
```

```
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Write Configuration register 2 with new dummy cycles */
tmp = CR2_DUMMY_CYCLES_66MHZ;
if (HAL_OSPI_Transmit(&hospi2, &tmp, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
AutoPollingWIP();
/* Enable writing to memory in order to set Octal DTR mode */
WriteEnable();
/* Initialize OCTOSPI1 to Indirect write mode to configure Octal mode */
sCommand.Instruction = WRITE_CFG_REG_2_CMD;
sCommand.Address = CONFIG_REG2_ADDR1;
sCommand.AddressMode = HAL_OSPI_ADDRESS_1_LINE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Write Configuration register 2 with with Octal mode */
tmp = CR2_DTR_OPI_ENABLE;
if (HAL_OSPI_Transmit(&hospi2, &tmp, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
}
/* This function erases the first memory sector */
void OctalSectorErase(void)
{
OSPI_RegularCmdTypeDef sCommand;
/* Initialize Indirect write mode to erase the first sector */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = OCTAL_SECTOR_ERASE_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_NONE;
```

```
sCommand.DataDtrMode        = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.DummyCycles = 0;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.AddressDtrMode     = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.AddressMode = HAL_OSPI_ADDRESS_8_LINES;
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.Address = 0;
/* Send Octal Sector erase cmd */
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
}
/* This function writes the memory */
void OctalDTR_MemoryWrite(void)
{
OSPI_RegularCmdTypeDef sCommand;
/* Initialize Indirect write mode for memory programming */
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = OCTAL_PAGE_PROG_CMD;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.AddressMode = HAL_OSPI_ADDRESS_8_LINES;
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.Address = 0x00000000;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_8_LINES;
sCommand.NbData = BUFFERSIZE;
sCommand.DummyCycles = 0;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.AddressDtrMode = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.DQSMode = HAL_OSPI_DQS_ENABLE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Memory Page programming */
if (HAL_OSPI_Transmit(&hospi2, aTxBuffer, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)!=
HAL_OK)
{
```

```
Error_Handler();
}
}
/* This function enables memory-mapped mode for Read and Write */
void EnableMemMapped(void)
{ OSPI_RegularCmdTypeDef sCommand;
OSPI_MemoryMappedTypeDef sMemMappedCfg;
/* Initialize memory-mapped mode for read operations */
sCommand.OperationType = HAL_OSPI_OPTYPE_READ_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_8_LINES;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_16_BITS;
sCommand.AddressMode = HAL_OSPI_ADDRESS_8_LINES;
sCommand.AddressSize = HAL_OSPI_ADDRESS_32_BITS;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_8_LINES;
sCommand.DummyCycles = DUMMY_CLOCK_CYCLES_READ;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.Instruction = OCTAL_IO_DTR_READ_CMD;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_ENABLE;
sCommand.AddressDtrMode = HAL_OSPI_ADDRESS_DTR_ENABLE;
sCommand.DataDtrMode = HAL_OSPI_DATA_DTR_ENABLE;
sCommand.DQSMode = HAL_OSPI_DQS_ENABLE;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Initialize memory-mapped mode for write operations */
sCommand.OperationType = HAL_OSPI_OPTYPE_WRITE_CFG;
sCommand.Instruction = OCTAL_PAGE_PROG_CMD;
sCommand.DummyCycles = 0;
if (HAL_OSPI_Command(&hospi2, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Configure the memory mapped mode with TimeoutCounter Disabled*/
sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_DISABLE;
if (HAL_OSPI_MemoryMapped(&hospi2, &sMemMappedCfg) != HAL_OK)
{
Error_Handler();
}
}
/* USER CODE END 4 */
```

- Adding defines to the *main.h* file

Update the *main.h* file by inserting the defines in the adequate space (in green bold below).

```
/* USER CODE BEGIN Private defines */
/* MX25LM512ABA1G12 Macronix memory */
/* Flash commands */
#define OCTAL_IO_DTR_READ_CMD           0xEE11
#define OCTAL_IO_READ_CMD                   0xEC13
#define OCTAL_PAGE_PROG_CMD              0x12ED
#define OCTAL_READ_STATUS_REG_CMD 0x05FA
#define OCTAL_SECTOR_ERASE_CMD         0x21DE
#define OCTAL_WRITE_ENABLE_CMD         0x06F9
#define READ_STATUS_REG_CMD                0x05
#define WRITE_CFG_REG_2_CMD                0x72
#define WRITE_ENABLE_CMD                       0x06
/* Dummy clocks cycles */
#define DUMMY_CLOCK_CYCLES_READ                6
#define DUMMY_CLOCK_CYCLES_READ_REG      4
/* Auto-polling values */
#define WRITE_ENABLE_MATCH_VALUE             0x02
#define WRITE_ENABLE_MASK_VALUE               0x02
#define MEMORY_READY_MATCH_VALUE         0x00
#define MEMORY_READY_MASK_VALUE            0x01
#define AUTO_POLLING_INTERVAL                           0x10
/* Memory registers address */
#define CONFIG_REG2_ADDR1                                   0x0000000
#define CR2_STR_OPI_ENABLE                                   0x01
#define CR2_DTR_OPI_ENABLE                                 0x02
#define CONFIG_REG2_ADDR3                                 0x00000300
#define CR2_DUMMY_CYCLES_66MHZ                 0x07
/* Exported macro -----------------------------------------------------*/
#define COUNTOF(__BUFFER__) (sizeof(__BUFFER__)/sizeof(*(__BUFFER__)))
/* Size of buffers */
#define BUFFERSIZE (COUNTOF(aTxBuffer) - 1)
/* USER CODE END Private defines */
```

- Build and run the code.

III. Quad-SPI PSRAM in Regular-command protocol example

In order to configure the OCTOSPI1 in Indirect/Memory-mapped mode and to configure the external Quad-SPI PSRAM AP Memory allowing communication in STR Quad-SPI mode, some functions must be added to the project. Code can be added to the *main.c* file (see code below) or defines can be added to the *main.h* file (see *Adding defines to the main.h file*).

- Adding code to the main.c file

Open the already generated project and follow the steps described below:

Note: Update the main.c file by inserting the lines of code to include the needed functions in the adequate space indicated in green bold below. This task avoids loosing the user code in case of project regeneration.

a) Insert variables declarations in the adequate space (in green bold below).

```
/* USER CODE BEGIN PV */
/*buffer that we will write n times to the external memory , user can modify
the content to write his desired data  */
uint8_t aTxBuffer[] = " **OCTOSPI/Quad-spi PSRAM Memory-mapped
communication example**  **OCTOSPI/Quad-spi PSRAM Memory-mapped
communication example**  **OCTOSPI/Quad-spi PSRAM Memory-mapped
communication example**  **OCTOSPI/Quad-spi PSRAM Memory-mapped
communication example** ";
/* USER CODE END PV */
```

b) Insert the functions prototypes in the adequate space (in green bold below).

```
/* USER CODE BEGIN PFP */
void EnterQuadMode(void);
void EnableMemMappedQuadMode(void);
/* USER CODE END PFP */
```

c) Insert the functions to be called in the `main()` function, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 1 */
__IO uint8_t *mem_addr;
uint32_t address = 0;
uint16_t index1;   /*index1 counter of bytes used when reading/writing 256
bytes buffer */
uint16_t index2; /*index2 counter of 256 bytes buffer used when
reading/writing the 1Mbytes extended buffer */
/* USER CODE END 1 */
```

```
/* USER CODE BEGIN 2 */
/* Enter Quad Mode 4-4-4  --------------------------------------------- */
EnterQuadMode();
/* Enable Memory mapped in Quad mode  -------------------------------- */
EnableMemMappedQuadMode();
/* Writing Sequence of 1Mbyte  -------------------------------------- */
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE + address);
for (index2 = 0; index2 < EXTENDEDBUFFERSIZE/BUFFERSIZE; index2++)
/*Writing 1Mbyte (256Byte BUFFERSIZE x 4096 times) */
{
for (index1 = 0; index1 < BUFFERSIZE; index1++)
{
*mem_addr = aTxBuffer[index1];
mem_addr++;
}
}
/* Reading Sequence of 1Mbyte --------------------------------------- */
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE + address);
for (index2 = 0; index2 < EXTENDEDBUFFERSIZE/BUFFERSIZE; index2++)
/*Reading 1Mbyte (256Byte BUFFERSIZE x 4096 times)*/
{
for (index1 = 0; index1 < BUFFERSIZE; index1++)
{
if (*mem_addr != aTxBuffer[index1])
{
/*can toggle led here*/
}
mem_addr++;
}
}
/*can toggle led here*/
/* USER CODE END 2 */
```

d) Insert the function definitions, called in the `main()`, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 4 */
/*Function to Enable Memory mapped mode in Quad mode 4-4-4*/
void EnableMemMappedQuadMode(void)
{
OSPI_RegularCmdTypeDef sCommand;
OSPI_MemoryMappedTypeDef sMemMappedCfg;
sCommand.FlashId            = HAL_OSPI_FLASH_ID_1;
sCommand.InstructionMode    = HAL_OSPI_INSTRUCTION_4_LINES;
sCommand.InstructionSize    = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.AddressMode        = HAL_OSPI_ADDRESS_4_LINES;
sCommand.AddressSize        = HAL_OSPI_ADDRESS_24_BITS;
sCommand.AddressDtrMode     = HAL_OSPI_ADDRESS_DTR_DISABLE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode           = HAL_OSPI_DATA_4_LINES;
sCommand.DataDtrMode        = HAL_OSPI_DATA_DTR_DISABLE;
sCommand.SIOOMode           = HAL_OSPI_SIOO_INST_EVERY_CMD;
sCommand.Address            = 0;
sCommand.NbData             = 1;

/* Memory-mapped mode configuration for Quad Read mode 4-4-4*/
sCommand.OperationType = HAL_OSPI_OPTYPE_READ_CFG;
sCommand.Instruction   = FAST_READ_QUAD;
sCommand.DummyCycles   = FAST_READ_QUAD_DUMMY_CYCLES;
if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/* Memory-mapped mode configuration for Quad Write mode 4-4-4*/
sCommand.OperationType = HAL_OSPI_OPTYPE_WRITE_CFG;
sCommand.Instruction   = QUAD_WRITE;
sCommand.DummyCycles   = WRITE_QUAD_DUMMY_CYCLES;
sCommand.DQSMode       = HAL_OSPI_DQS_ENABLE;
if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
/*Disable timeout counter for memory mapped mode*/
sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_DISABLE;
/*Enable memory mapped mode*/
```

```
if (HAL_OSPI_MemoryMapped(&hospi1, &sMemMappedCfg) != HAL_OK)
{
Error_Handler();
}
}

/*Function to configure the external memory in Quad mode 4-4-4*/
void EnterQuadMode(void)
{
OSPI_RegularCmdTypeDef sCommand;
sCommand.OperationType = HAL_OSPI_OPTYPE_COMMON_CFG;
sCommand.FlashId = HAL_OSPI_FLASH_ID_1;
sCommand.Instruction = ENTER_QUAD_MODE;
sCommand.InstructionMode = HAL_OSPI_INSTRUCTION_1_LINE;
sCommand.InstructionSize = HAL_OSPI_INSTRUCTION_8_BITS;
sCommand.InstructionDtrMode = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
sCommand.AddressMode = HAL_OSPI_ADDRESS_NONE;
sCommand.AlternateBytesMode = HAL_OSPI_ALTERNATE_BYTES_NONE;
sCommand.DataMode = HAL_OSPI_DATA_NONE;
sCommand.DummyCycles = ENTER_QUAD_DUMMY_CYCLES;
sCommand.DQSMode = HAL_OSPI_DQS_DISABLE;
sCommand.SIOOMode = HAL_OSPI_SIOO_INST_EVERY_CMD;
/*Enter QUAD mode*/
if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
{
Error_Handler();
}
}

/* USER CODE END 4 */
```

- Adding defines to the *main.h* file

Update the *main.h* file by inserting the defines in the adequate space (in green bold below).

```
/* USER CODE BEGIN Private defines */
/*APS1604M-3SQR PSRAM APmemory*/
#define FAST_READ_QUAD                                    0xEB
#define QUAD_WRITE                                             0x38
#define FAST_READ_QUAD_DUMMY_CYCLES 6
#define WRITE_QUAD_DUMMY_CYCLES          0
#define ENTER_QUAD_DUMMY_CYCLES          0
#define QUAD_WRITE                                             0x38
#define ENTER_QUAD_MODE                              0x35
#define EXIT_QUAD_MODE                                  0xF5
/* Exported macro ------------------------------------------------------*/
#define BUFFERSIZE                    (COUNTOF(aTxBuffer) - 1)
#define COUNTOF(__BUFFER__)           (sizeof(__BUFFER__) /
sizeof(*(__BUFFER__)))
#define EXTENDEDBUFFERSIZE                   (1048576)
/* USER CODE END Private defines */
```

- Build and run the code.

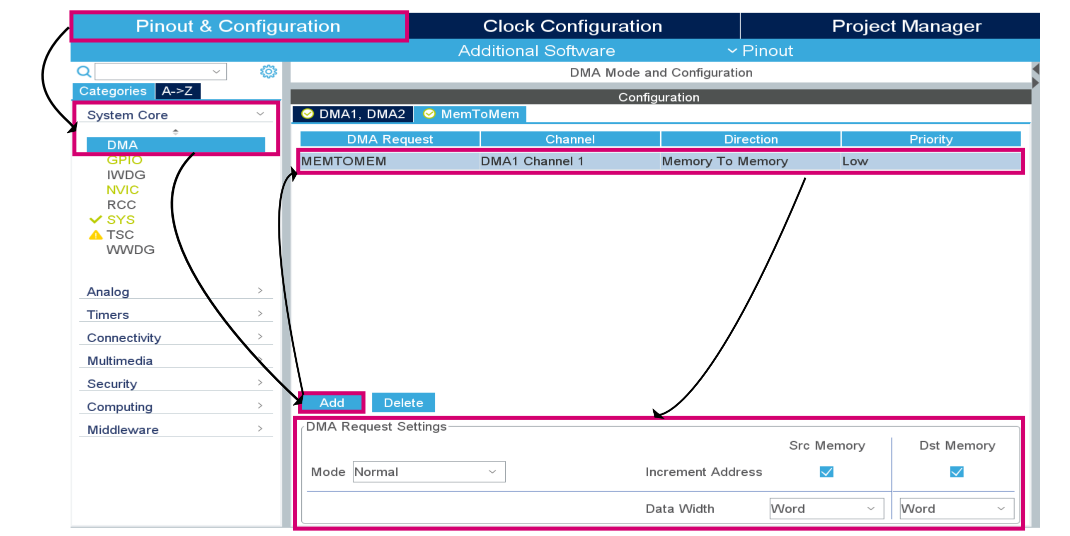
IV. HyperFlash and HyperRAM memories with Multiplexed mode example

The following example shows how to read data from the external HyperFlash using DMA1, while the CPU reads data from the HyperRAM.

The DMA1 must be configured using the STM32CubeMX, with the following steps under system core:

- Select DMA.
- Under MemToMem, select Add.
- Configure the DMA request and the DMA request settings like the figure below.

Figure 28. STM32CubeMX - DMA1 configuration



Note: Pink color highlight key items.

MSv65673V1

In order to configure the OCTOSPI1 and OCTOSPI2 in Memory-mapped mode and to read data from the two external HyperBus memories, some functions must be added to the project. Code can be added to the *main.c* file (see code below) or defines can be added to the *main.h* file (see *Adding defines to the main.h file*).

- Adding code to the main.c file

Open the already generated project and follow the steps described below:

Note: Update the main.c file by inserting the lines of code to include the needed functions in the adequate space indicated in green bold below. This task avoids loosing the user code in case of project regeneration.

a) Insert variables declarations in the adequate space (in green bold below).

```
/* USER CODE BEGIN PV */
/*define a 64Kbyte buffer for HyperRam data read with CPU*/
#pragma location = 0x20020000
uint32_t RxHyperRAM[BUFFERSIZE];
/* USER CODE END PV */
```

b) Insert the functions prototypes in the adequate space (in green bold below).

```
/* USER CODE BEGIN PFP */
void EnableMemMapped(void);
void DelayBlock_Calibration(void);
/* USER CODE END PFP */
```

c) Insert the functions to be called in the `main()` function, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 1 */
/*pointer on OCTOSPI1 memory mapped address region*/
__IO uint32_t *OCTOSPI1_MEMMAPPED_ADD  = (__IO uint32_t *)(OCTOSPI1_BASE);
/* USER CODE END 1 */

/* USER CODE BEGIN 2 */
/*Configure the MAXTRAN feature for 241 clock cycles for OCTOSPI1 and
OCTOSPI2 (4µs of max transaction period)*/
MAXTRAN_Configuration();
/*Configure and Enable the Memory Mapped mode for both OCTOSPI1 and OCTOSPI2
respectively at address 0x90000000 and 0x70000000*/
EnableMemMapped();

/*Delay block Calibration*/
DelayBlock_Calibration();

/*Start Data read (64Kbyte) with DMA1 from the HyperFlash (0x70000000) to
the internal SRAM3 (0x20030000)*/
if(HAL_DMA_Start(&hdma_memtomem_dma1_channel1,OCTOSPI2_BASE, SRAM3_BASE,
BUFFERSIZE) != HAL_OK)
{
Error_Handler();
}
/*Start Data read (64Kbyte) with CPU from the HyperRAM (0x90000000) to
the internal SRAM2 (0x20020000) while the DMA is reading from HyperFLASH*/
for (index = 0; index < BUFFERSIZE; index++)
{
RxHyperRAM[index] = *OCTOSPI1_MEMMAPPED_ADD++;
}
/* USER CODE END 2 */
```

d) Insert the function definitions, called in the `main()`, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 4 */
/* Memory-mapped mode configuration for OCTOSPI1 and OCTOSPI2--------- */
void EnableMemMapped(void)
{
OSPI_HyperbusCmdTypeDef sCommand;
OSPI_MemoryMappedTypeDef sMemMappedCfg;
/* Memory-mapped mode configuration ---------------------------------- */
sCommand.AddressSpace = HAL_OSPI_MEMORY_ADDRESS_SPACE;
sCommand.AddressSize  = HAL_OSPI_ADDRESS_32_BITS;
sCommand.DQSMode      = HAL_OSPI_DQS_ENABLE;
sCommand.Address      = 0;
sCommand.NbData       = 1;
if (HAL_OSPI_HyperbusCmd(&hospi1, &sCommand,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
{
Error_Handler();
}
if (HAL_OSPI_HyperbusCmd(&hospi2, &sCommand,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
{
Error_Handler();
}
sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_ENABLE;
sMemMappedCfg.TimeOutPeriod     = 0x1;
if (HAL_OSPI_MemoryMapped(&hospi1, &sMemMappedCfg) != HAL_OK)
{
Error_Handler();
}
sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_ENABLE;
sMemMappedCfg.TimeOutPeriod     = 0x1;
if (HAL_OSPI_MemoryMapped(&hospi2, &sMemMappedCfg) != HAL_OK)
{
Error_Handler();
}
}
/*This function is used to calibrate the Delayblock before initiating
USER's application read/write transactions*/
void DelayBlock_Calibration(void)
```

```
{
/*buffer used for calibration*/
uint8_t Cal_buffer[] = " ****Delay Block Calibration Buffer****  ****Delay
Block Calibration Buffer****  ****Delay Block Calibration Buffer****
****Delay Block Calibration Buffer****  ****Delay Block Calibration
Buffer****  ****Delay Block Calibration Buffer**** ";
uint16_t index;
__IO uint8_t *mem_addr;
uint8_t test_failed;
uint8_t delay = 0x0;
uint8_t Min_found = 0;
uint8_t Max_found = 0;
uint8_t Min_Window = 0x0;
uint8_t Max_Window = 0xF;
uint8_t Mid_window = 0;
uint8_t calibration_ongoing = 1;
/* Write the Cal_buffer to the memory*/
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE);
for (index = 0; index < DLYB_BUFFERSIZE; index++)
{
*mem_addr = Cal_buffer[index];
mem_addr++;
}
while (calibration_ongoing)
{
/* update the Delayblock calibration */
HAL_RCCEx_OCTOSPIDelayConfig(delay, 0);
test_failed = 0;
mem_addr = (__IO uint8_t *)(OCTOSPI1_BASE);
for (index = 0; index < DLYB_BUFFERSIZE; index++)
{
/* Read the Cal_buffer from the memory*/
if (*mem_addr != Cal_buffer[index])
{
/*incorrect data read*/
test_failed = 1;
}
mem_addr++;
}
/*          search for the Min window       */
```

```
if (Min_found!=1)
{
if (test_failed == 1)
{
if (delay < 15)
{
delay++;
}
else
{
/* If delay set to maximum and error still detected: can't use external
Memory*/
Error_Handler();
}
}
else
{
Min_Window = delay;
Min_found=1;
delay = 0xF;
}
}
/*              search for the Max window          */
else if (Max_found!=1)
{
if (test_failed == 1)
{
if (delay > 0)
{
delay--;
}
else
{
/* If delay set to minimum and error still detected: can't use external
Memory */
Error_Handler();
}
}
else
{
```

```
Max_Window = delay;
Max_found=1;
}
}
/* min and max delay window found , configure the delay block with the
middle window value and exit calibration */
else
{
Mid_window = (Max_Window+Min_Window)/2;
HAL_RCCEx_OCTOSPIDelayConfig(Mid_window, 0);
/* Exit calibration */
calibration_ongoing = 0;
}
}
}
/* MAXTRAN configuration function for OCTOSPI1 and OCTOSPI2 */
void MAXTRAN_Configuration(void)
{
/*Maximum transaction configured for 4us*/
MODIFY_REG(hospi1.Instance->DCR3, OCTOSPI_DCR3_MAXTRAN, 0x000000F1);
MODIFY_REG(hospi2.Instance->DCR3, OCTOSPI_DCR3_MAXTRAN, 0x000000F1);
}

/* USER CODE END 4 */
```

- Adding defines to the *main.h* file

Update the *main.h* file by inserting the defines in the adequate space (in green bold below).

```
/* USER CODE BEGIN Private defines */
#define BUFFERSIZE                  0x4000
#define DLYB_BUFFERSIZE                   (COUNTOF(Cal_buffer) - 1)
#define COUNTOF(__BUFFER__)          (sizeof(__BUFFER__) /
sizeof(*(__BUFFER__)))

/* USER CODE END Private defines */
```

- Build and run the code.

7 XSPI application example configuration with STM32CubeMX

This section provides typical XSPI implementation examples with STM32H7S7xx products, and STM32CubeMX examples using the STM32H7S7L8-DK Discovery kit for the STM32H7S7L8H6H microcontroller.

This section also shows an example of basic XSPI configuration based on the STM32H7S7L8-DK Discovery kit for regular-command protocol in Memory-mapped mode for writing and reading from the XSPI PSRAM.

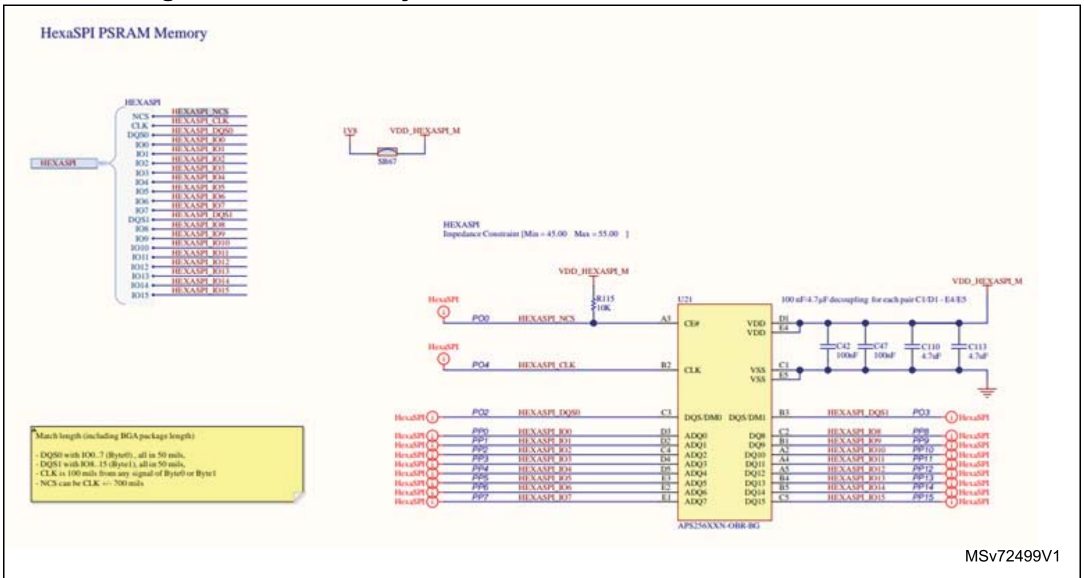
7.1 Hardware description

The STM32H7S7L8-DK Discovery kit embeds the 16-bits AP Memory PSRAM. This AP Memory APS256XXN-OBRx DDR Octal SPI PSRAM memory is connected to XSPIM Port 1.

As shown in *Figure 29*, the 16-bits AP Memory PSRAM memory is connected to the MCU, using each of the eleven pins:

- HEXASPI_NCS
- HEXASPI_CLK
- HEXASPI_DQS0/1
- HEXASPI_IO[0...15]

Figure 29. AP Memory PSRAM connection on STM32H7S7L8-DK



7.2 Use case description

The adopted configuration for 16-bits AP Memory PSRAM is:

- XSPI1 signals mapped to Port 1 (AP Memory PSRAM), so XSPI1 must be set to Regular-command protocol
- DTR 16 mode (with DQS) with XSPI1 running at 200 MHz
- Write/read in Memory-mapped mode

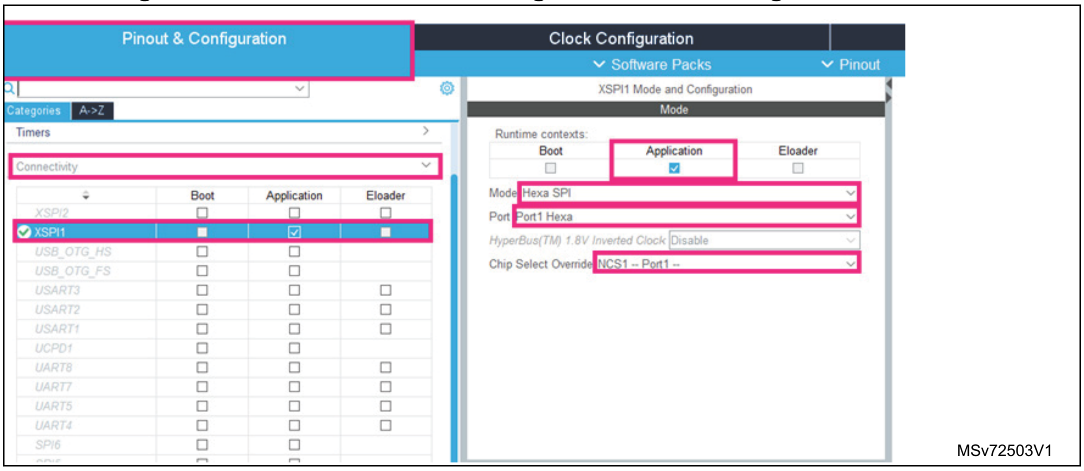
7.3 XSPI GPIOs and clocks configuration

I-STM32CubeMX: GPIOs configuration

Once the STM32CubeMX project is created for the STM32H7S7L8H6H product, follow the steps below:

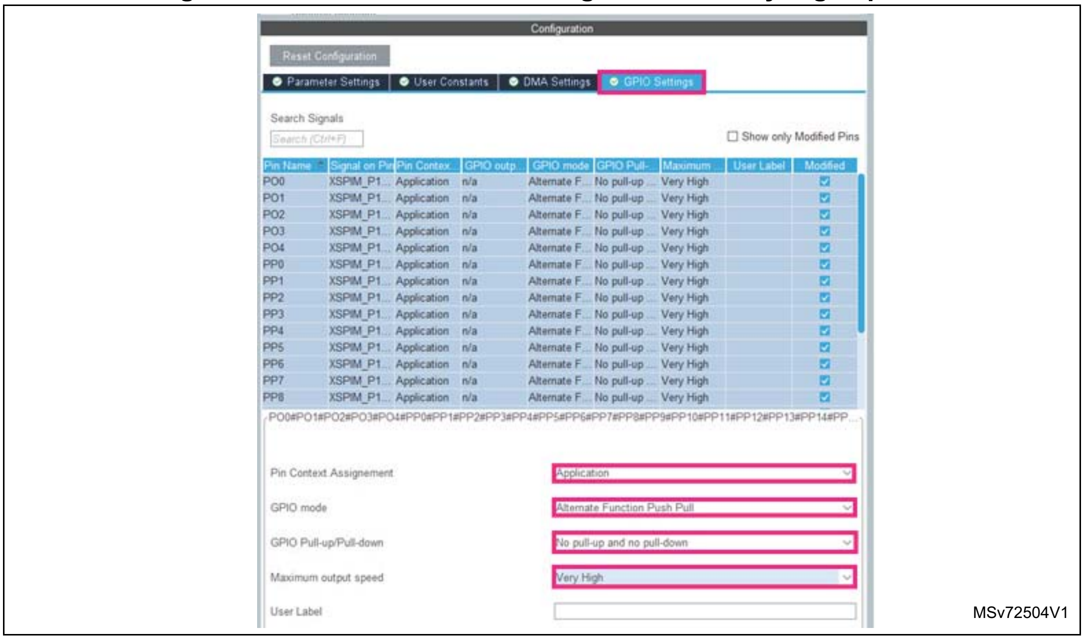
1. Select the *Pinout and Configuration* tab and, under *Connectivity,* expand the XSPI1 and the configuration as shown in *Figure 30*.

Figure 30. STM32CubeMX - Configuration of XSPI signals and mode



2. Configure the XSPI GPIOs to very-high speed.

Figure 31. STM32CubeMX - Setting GPIOs to very-high speed

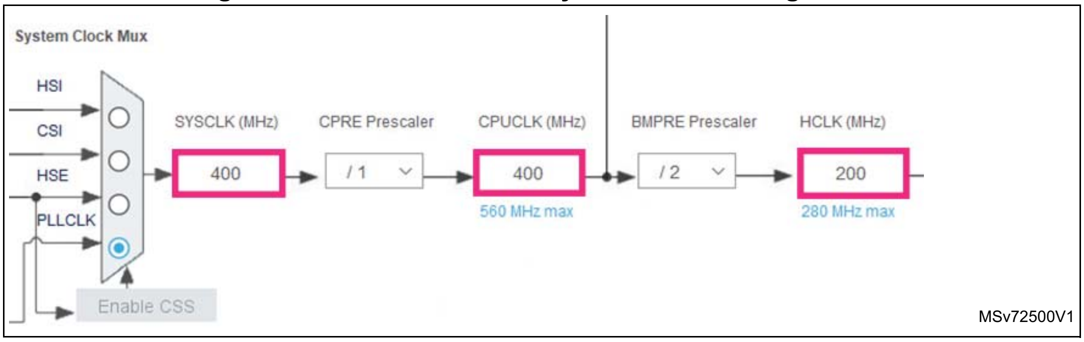


II-STM32CubeMX: clocks configuration

In this example, the system clock is configured with:

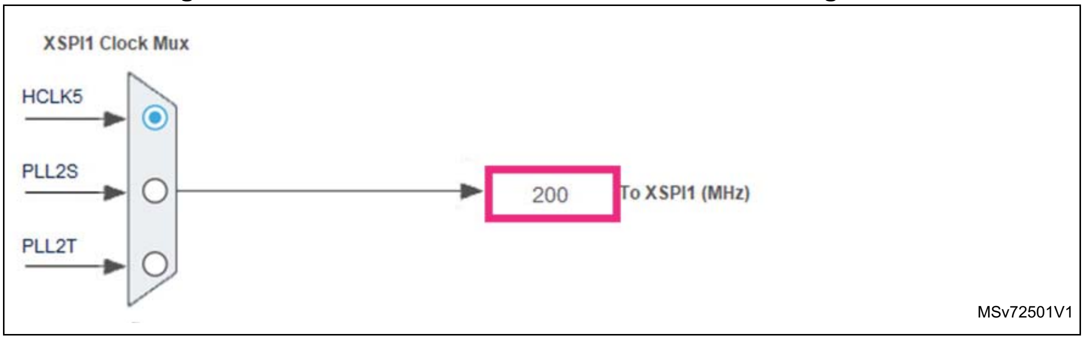
- Main PLL is used as system source clock.
- SYSCLK set to 400MHz and HCLK set to 200 MHz (AP Memory PSRAM max frequency). In this example, the HCLK5 is used as clock source for XSPI1.
- System clock configuration:
  - Select the clock configuration tab.
  - In the Clock configuration tab, set the PLLs and the prescalers so that the system clock at 400MHz is as shown in *Figure 32*.

Figure 32. STM32CubeMX - System clock configuration



- XSPI clock source configuration: In the Clock configuration tab, select the SYSCLK clock source (see *Figure 33*).

Figure 33. STM32CubeMX - XSPI1 clock source configuration



III-XSPI configuration and parameter settings

After setting the XSPI1 GPIOs and the clock configurations, the user must configure the XSPI depending on the AP PSRAM memory and its communication protocol.

In the XSPI1 Configuration window, select the *Parameter Settings* tab and configure it as shown in *Figure 34*.

Figure 34. STM32CubeMX - Configuration of XSPI1 parameters

Build and run the project. At this stage, the user can build, debug, and run the project.

7.4 STM32CubeMX: Project generation

Once all of the GPIOs, the clock, and the XSPI configurations are set, generate the project with the desired toolchain (such as STM32CubeIDE, EWARM or MDK-ARM).

In order to configure the XSPI1 in Memory-mapped mode and to configure the external 16-bit AP Memory PSRAM allowing communication in DTR 16-bit mode (with DQS), some functions must be added to the project. Code can be added to the main.c file (see code below) or defines can be added to the *main.h* file (see Adding defines to the main.h file).

Open the already generated project and follow the steps described below.

Note: Update the main.c file by inserting the lines of code to include the needed functions in the adequate space indicated in green bold below. This task avoids loosing the user code in case of project regeneration.

Adding code to the main.c file

a) Insert variables declarations in the adequate space (in green bold below).

```
/* USER CODE BEGIN PV */
/* Buffer used for transmission */
```

```
uint8_t aTxBuffer[BUFFERSIZE];
__IO uint8_t *mem_addr;
uint8_t CmdCplt, TxCplt , StatusMatch , RxCplt;
XSPI_MemoryMappedTypeDef sMemMappedCfg;
/* USER CODE END PV */
```

b) Insert the functions prototypes in the adequate space (in green bold below).

```
/* USER CODE BEGIN PFP */
uint32_t APS6408_WriteReg(XSPI_HandleTypeDef *Ctx, uint32_t Address,
uint8_t *Value);
uint32_t APS6408_ReadReg(XSPI_HandleTypeDef *Ctx, uint32_t Address, uint8_t
*Value, uint32_t LatencyCode);
static void Configure_APMemory(void);
/* USER CODE END PFP */
```

c) Insert the functions to be called in the main() function, in the adequate space (in green bold below).

```
/* USER CODE BEGIN 1 */
XSPI_RegularCmdTypeDef sCommand = {0};
  uint16_t errorBuffer = 0;
  uint32_t index, index_K;
/* USER CODE END 1 */

/* USER CODE BEGIN 2 */
Configure_APMemory();

/*Configure Memory Mapped mode*/

  sCommand.OperationType       = HAL_XSPI_OPTYPE_WRITE_CFG;
  sCommand.InstructionMode     = HAL_XSPI_INSTRUCTION_8_LINES;
  sCommand.InstructionWidth    = HAL_XSPI_INSTRUCTION_8_BITS;
  sCommand.InstructionDTRMode = HAL_XSPI_INSTRUCTION_DTR_DISABLE;
  sCommand.Instruction         = WRITE_CMD;
  sCommand.AddressMode         = HAL_XSPI_ADDRESS_8_LINES;
  sCommand.AddressWidth        = HAL_XSPI_ADDRESS_32_BITS;
  sCommand.AddressDTRMode      = HAL_XSPI_ADDRESS_DTR_ENABLE;
  sCommand.Address             = 0x0;
  sCommand.AlternateBytesMode = HAL_XSPI_ALT_BYTES_NONE;
  sCommand.DataMode            = HAL_XSPI_DATA_16_LINES;
  sCommand.DataDTRMode         = HAL_XSPI_DATA_DTR_ENABLE;
  sCommand.DataLength          = BUFFERSIZE;
  sCommand.DummyCycles         = DUMMY_CLOCK_CYCLES_WRITE;
  sCommand.DQSMode             = HAL_XSPI_DQS_ENABLE;
if (HAL_XSPI_Command(&hxspi1, &sCommand,
HAL_XSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
  {
    Error_Handler();
```

```
sCommand.OperationType = HAL_XSPI_OPTYPE_READ_CFG;
  sCommand.Instruction = READ_CMD;
  sCommand.DummyCycles = DUMMY_CLOCK_CYCLES_READ;
  sCommand.DQSMode     = HAL_XSPI_DQS_ENABLE;

if (HAL_XSPI_Command(&hxspi1, &sCommand, HAL_XSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
  {
    Error_Handler();
  }

  sMemMappedCfg.TimeOutActivation = HAL_XSPI_TIMEOUT_COUNTER_ENABLE;
  sMemMappedCfg.TimeoutPeriodClock      = 0x34;

if (HAL_XSPI_MemoryMapped(&hxspi1, &sMemMappedCfg) != HAL_OK)
  {
    Error_Handler();
  }

/*fill aTxBuffer */
 for (index_K = 0; index_K < 10; index_K++)
  {
    for (index = (index_K  * KByte); index < ((index_K +1) * KByte);
index++)
    {
      aTxBuffer[index]=index + index_K;
    }
  }
/*Writing Sequence ----------------------------------------------- */
index_K=0;
  for (index_K = 0; index_K < 10; index_K++)
  {
    mem_addr = (uint8_t *)(XSPI1_BASE + (index_K * KByte));
    for (index = (index_K  * KByte); index < ((index_K +1) * KByte);
index++)
    {
      *mem_addr = aTxBuffer[index];
      mem_addr++;
    }

/* In memory-mapped mode, not possible to check if the memory is ready
    after the programming. So a delay corresponding to max page programming
    time is added */
HAL_Delay(1);
  }
```

```
/* Reading Sequence ---------------------------------------------- */
index_K=0;
  for (index_K = 0; index_K < 2; index_K++)
  {
    mem_addr = (uint8_t *)(XSPI1_BASE + (index_K * KByte));
    for (index = (index_K  * KByte); index < ((index_K +1) * KByte);
index++)
    {
      if (*mem_addr != aTxBuffer[index])
      {
        /* can toggle led here*/
        errorBuffer++;
      }
      mem_addr++;
    }

/* In memory-mapped mode, not possible to check if the memory is ready
    after the programming. So a delay corresponding to max page programming
    time is added */
HAL_Delay(1);
  }
  if (errorBuffer == 0)
  {
  /* can toggle led here*/
  }

/* Abort XSPI driver to stop the memory-mapped mode ------------ */ if
(HAL_XSPI_Abort(&hxspi1) != HAL_OK)
  {
    Error_Handler();
  }
/* USER CODE END 2 */

/* USER CODE BEGIN 4 */

/***Write mode register*/
uint32_t APS6408_WriteReg(XSPI_HandleTypeDef *Ctx, uint32_t Address,
uint8_t *Value)
{
  XSPI_RegularCmdTypeDef sCommand1={0};

/*Initialize the write register command */
sCommand1.OperationType       = HAL_XSPI_OPTYPE_COMMON_CFG;
  sCommand1.InstructionMode     = HAL_XSPI_INSTRUCTION_8_LINES;
```

```
  sCommand1.InstructionWidth    = HAL_XSPI_INSTRUCTION_8_BITS;
  sCommand1.InstructionDTRMode = HAL_XSPI_INSTRUCTION_DTR_DISABLE;
  sCommand1.Instruction        = WRITE_REG_CMD;
  sCommand1.AddressMode        = HAL_XSPI_ADDRESS_8_LINES;
  sCommand1.AddressWidth        = HAL_XSPI_ADDRESS_32_BITS;
  sCommand1.AddressDTRMode     = HAL_XSPI_ADDRESS_DTR_ENABLE;
  sCommand1.Address            = Address;
  sCommand1.AlternateBytesMode = HAL_XSPI_ALT_BYTES_NONE;
  sCommand1.DataMode           = HAL_XSPI_DATA_8_LINES;
  sCommand1.DataDTRMode        = HAL_XSPI_DATA_DTR_ENABLE;
  sCommand1.DataLength         = 2;
  sCommand1.DummyCycles        = 0;
  sCommand1.DQSMode            = HAL_XSPI_DQS_DISABLE;

/* Configure the command*/
  if (HAL_XSPI_Command(Ctx, &sCommand1, HAL_XSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
  {
    return HAL_ERROR;
  }

/* Transmission of the data */
  if (HAL_XSPI_Transmit(Ctx, (uint8_t *)(Value),
HAL_XSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
  {
    return HAL_ERROR;
  }

  return HAL_OK;
}

/**Read mode register value*/
uint32_t APS6408_ReadReg(XSPI_HandleTypeDef *Ctx, uint32_t Address, uint8_t
*Value, uint32_t LatencyCode)
{
  XSPI_RegularCmdTypeDef sCommand;

/* Initialize the read register command */
sCommand.OperationType       = HAL_XSPI_OPTYPE_COMMON_CFG;
  sCommand.InstructionMode     = HAL_XSPI_INSTRUCTION_8_LINES;
  sCommand.InstructionWidth     = HAL_XSPI_INSTRUCTION_8_BITS;
  sCommand.InstructionDTRMode = HAL_XSPI_INSTRUCTION_DTR_DISABLE;
  sCommand.Instruction         = READ_REG_CMD;
  sCommand.AddressMode         = HAL_XSPI_ADDRESS_8_LINES;
  sCommand.AddressWidth         = HAL_XSPI_ADDRESS_32_BITS;
  sCommand.AddressDTRMode      = HAL_XSPI_ADDRESS_DTR_ENABLE;
```

```
  sCommand.Address              = Address;
  sCommand.AlternateBytesMode = HAL_XSPI_ALT_BYTES_NONE;
  sCommand.DataMode             = HAL_XSPI_DATA_8_LINES;
  sCommand.DataDTRMode          = HAL_XSPI_DATA_DTR_ENABLE;
  sCommand.DataLength              = 2;
  sCommand.DummyCycles          = (LatencyCode - 1U);
  sCommand.DQSMode              = HAL_XSPI_DQS_ENABLE;

/* Configure the command */
if (HAL_XSPI_Command(Ctx, &sCommand, HAL_XSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
  {
    return HAL_ERROR;
  }

/* Reception of the data */
if (HAL_XSPI_Receive(Ctx, (uint8_t *)Value, HAL_XSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
  {
    return HAL_ERROR;
  }
return HAL_OK;
}

/** Switch from Octal Mode to Hexa Mode on the memory*/
static void Configure_APMemory(void)
{
/* MR0 register for read and write */

  uint8_t regW_MR0[2]={0x24,0x8D}; /* To configure AP memory Latency Type
and drive Strength */
uint8_t regR_MR0[2]={0};

/* MR8 register for read and write */
uint8_t regW_MR8[2]={0x4B,0x08}; /* To configure AP memory Burst Type */
  uint8_t regR_MR8[2]={0};

/*Read Latency */
uint8_t latency=6;

/*Configure Read Latency and drive Strength */
if (APS6408_WriteReg(&hxspi1, MR0, regW_MR0) != HAL_OK)
  {
    Error_Handler();
  }
```

```
/* Check MR0 configuration */
if (APS6408_ReadReg(&hxspi1, MR0, regR_MR0, latency ) != HAL_OK)
  {
    Error_Handler();
  }

/* Check MR0 configuration */
if (regR_MR0 [0] != regW_MR0 [0])
 {
    Error_Handler() ;
  }

/* Configure Burst Length */
if (APS6408_WriteReg(&hxspi1, MR8, regW_MR8) != HAL_OK)
  {
    Error_Handler();
  }

/* Check MR8 configuration */
if (APS6408_ReadReg(&hxspi1, MR8, regR_MR8, 6) != HAL_OK)
  {
    Error_Handler();
  }

  if (regR_MR8[0] != regW_MR8[0])
  {
    Error_Handler() ;
  }
}
/* USER CODE END 4 */
```

- Adding defines to the *main.h* file

Update the *main.h* file by inserting the defines in the adequate space (in green bold below).

```
/* USER CODE BEGIN EC */

/* Aps256xx APMemory memory */

/* Read Operations */
#define READ_CMD 0x00
#define READ_LINEAR_BURST_CMD 0x20

/* Write Operations */
#define WRITE_CMD                                                0x80
#define WRITE_LINEAR_BURST_CMD 0xA0
```

```
/* Registers definition */
#define MR0 0x00000000
#define MR8 0x00000008

/* Register Operations */
#define READ_REG_CMD 0x40
#define WRITE_REG_CMD 0xC0

/* Default dummy clocks cycles */
#define DUMMY_CLOCK_CYCLES_READ 4
#define DUMMY_CLOCK_CYCLES_WRITE 4

/* Size of buffers */
#define BUFFERSIZE 10240
#define KByte 1024
/* USER CODE END EC */
```

- Build and run the code.

8 Performance and power

This section explains how to get the best performances and how to decrease the application power consumption.

8.1 How to get the best read performance

There are three main recommendations to be followed in order to get the optimum reading performances:

- Configure OCTOSPI/HSPI/XSPI at its maximum speed.
- Use Octo-SPI, Hexadeca-SPI, and XSPI DTR mode for Regular command protocol.
- Reduce command overhead:

  Each new read operation needs a command/address to be sent plus a latency period that leads to command overhead. In order to reduce command overhead and boost the read performance, the user must focus on the following points:
  - Use large burst transfers

    Since each access to the external memory issues command/address, it is beneficial to perform large burst transfers rather than small repetitive transfers. This action reduces command overhead.
  - Sequential access

    The best read performance is achieved if the stored data is read out sequentially, which avoids command and address overhead and then leads to reach the maximum performances at the operating OCTOSPI/HSPI/XSPI clock speed.
  - Consider timeout counter

    The user must consider that enabling timeout counter in Memory-mapped mode may increase the command overhead and then decrease the read performance. When timeout occurs, the OCTOSPI/HSPI/XSPI rises chip-select. After that, to read again from the external memory, a new read sequence needs to be initiated. It means that the read command must be issued again, which leads to command overhead.
    Note that timeout counter allows decreasing power consumption, but if the performance is a concern, the user can increase the timeout period in the OCTOSPI_LPTR/HSPI_LPTR/XSPI_LPTR register or even disable it.

8.2 Decreasing power consumption

One of the most important requirements in wearable and mobile applications is the power efficiency. Power consumption can be decreased by following the recommendations presented in this section.

To decrease the total application power-consumption, the STM32 is usually put in low-power mode. To reduce even more the current consumption, the connected memory can also be put in low-power mode.

8.2.1 STM32 low-power modes

The STM32 low-power states are important requirements that must be considered as they have a direct effect on the overall application power consumption and on the Octo-SPI and Hexadeca-SPI interface state.

For more informations about STM32 low-power modes configuration, refer to the product reference manual.

8.2.2 Decreasing Octo-SPI, Hexadeca-SPI, and XSPI memory power consumption

In order to save more energy when the application is in low-power mode, it is recommended to put the memory in low-power mode before entering the STM32 in low-power mode.

Timeout counter usage

The timeout counter feature can be used to avoid any extra power-consumption in the external memory. This feature can be used only in Memory-mapped mode. When the clock is stopped for a long time and after a period of timeout elapsed without any access, the timeout counter releases the NCS pin to put the external memory in a lower-consumption state (so called Standby mode).

Put the memory in deep power-down mode

For most octal memory devices, the default mode after the power-up sequence, is the Standby low-power mode. In Standby mode, there is no ongoing operation. The NCS is high and the current consumption is relatively less than in operating mode.

To save more energy, some memory manufacturers provide another low-power mode commonly known DPD (deep power-down mode). This is different from Standby mode. During the DPD, the device is not active and most commands (such as write, program or read) are ignored.

The application can put the memory device in DPD mode before entering the STM32 in low-power mode, when the memory is not used. This action allows a reduction of the overall application power-consumption and a longer wakeup time.

Entering and exiting DPD mode

To enter DPD mode, a DPD command sequence must be issued to the external memory. Each memory manufacturer has its dedicated DPD command sequence.

To exit DPD mode, some memory devices require an RDP (release from deep power-down) command to be issued. For some other memory devices, a hardware reset leads to exit DPD mode.

Note: Refer to the relevant memory device datasheet for more details.

9 Supported devices

The Octo-SPI, Hexadeca-SPI, and XSPI interface can operate in two different low-level protocols: Regular-command and HyperBus.

Thanks to the Regular-command frame format flexibility, any Single-SPI, Dual-SPI, Quad-SPI or Octo-SPI or 16-bit memory can be connected to an STM32 device. There are several suppliers of Octo-SPI or 16-bit compatible memories (such as Macronix, Adesto, Micron, AP Memory, Infineon, or Winbond).

Thanks to the HyperBus protocol support, several HyperRAM and HyperFlash memories are supported by the STM32 devices. Some memory manufacturers (such as Infineon, Winbond, or ISSI) provide HyperRAM and HyperFlash memories.

As already described in *Section 6.2*, the Macronix MX25LM51245GXDI0A Octo-SPI flash memory is embedded on the STM32L4R9I-EVAL and STM32L552E-EVAL boards, and on the STM32L4R9I-DISCO Discovery kit.

10 Conclusion

Some STM32 MCUs provide a very flexible Octo-SPI, Hexadeca-SPI, and XSPI interface that fits memory hungry applications at a lower cost, and avoids the complexity of designing with external parallel memories by reducing pin count and offering better performances.

This application note demonstrates the excellent Octo-SPI, Hexadeca-SPI, and XSPI interface variety of features and flexibility on the STM32L4+ series, STM32L5 series STM32H7A3/B3/B0, STM32H72x/73x, STM32H5 series, STM32U5 series, and STM32H7Rx/Sx series. The STM32 OCTOSPI, HSPI, and XSPI peripheral allows lower development costs and faster time to market.

11 Revision history

Table 9. Document revision history

Date	Revision	Changes
20-Oct-2017	1	Initial release.
27-Apr-2018	2	Updated – *Section 1: Overview of the OCTOSPI interface in the STM32 MCUs system architecture* – *Section 4.2.2: Use case description* – *Section 4.2.3: OCTOSPI GPIOs and clocks configuration* – *Section 5.2: Decreasing power consumption* and all its subsections – *Section : STM32CubeMX: project generation on page 35* – *Section : STM32CubeMX: OCTOSPI2 peripheral configuration in HyperBus™ mode on page 46* – *Section : STM32CubeMX: project generation on page 47* – *Figure 10: Examples configuration: OCTOSPI1 set to regular-command mode and OCTOSPI2 set to HyperBus™* – *Figure 25: OCTOSPI2 peripheral configuration in HyperBus™ mode* – *Table 2: OCTOSPI availability and features across STM32 families* – Added: – *Section 5: Performance and power* – *Section 5.1: How to get the best read performance* – *Section 5.1.1: Read performance* – *Section 6: Supported devices*
11-Oct-2019	3	Updated: – Doc title and *Introduction* – *Section 1.1: OCTOSPI main features* – *Figure 1: STM32L4+ Series system architecture* – *Section 2.3.3: Memory-mapped mode* Added STM32L5 series: – *Section 1.2.2: STM32L5 Series system architecture* – *Section 3.1.1: Connecting two octal memories to one Octo-SPI interface* – *Section 5.2.1: STM32 low-power modes* – *Conclusion* Removed *Section 5.1.1: Read performance*

Table 9. Document revision history (continued)

Date	Revision	Changes
19-Dec-2019	4	Updated: – *Introduction* and *Table 1: Applicable products* – *Section 1: Overview of the OCTOSPI in STM32 MCUs* – *Table 2: OCTOSPI main features* – *Section 1.2: OCTOSPI in a smart architecture* – *Figure 1: STM32L4+ Series system architecture* – *Figure 2: STM32L5 Series system architecture* – *Section 2.1.2: OCTOSPI I/O manager* – *Section 2.1.3: OCTOSPI delay block* – *Section 2.3.3: Memory-mapped mode* – *Section 3.1.1: GPIOs and OCTOSPI I/Os configuration* – *Section 3.2: OCTOSPI configuration for regular-command protocol* – *Section 3.3: OCTOSPI configuration for HyperBus protocol* – *Section 4: OCTOSPI application examples* – *Section 6: Supported devices* – *Section 7: Conclusion* Added: – *Section 1.2.3: STM32H7A3/B3 system architecture* – *Figure 5: OCTOSPI multiplexed mode use case example* Removed: – Section Connecting two octal memories to one Octo-SPI interface – Section 4.2.5 HyperBus protocol
27-Apr-2020	5	Updated: – *Table 2: OCTOSPI main features* – *Section 1.2.1: STM32L4+ Series system architecture* – *Figure 1*,*Figure 2* and *Figure 3* – Structure of *Section 2: Octo-SPI interface description* – *Section 2.1: OCTOSPI hardware interface* – *Section 2.1.2: OCTOSPI delay block* – *Section 4.1.1: GPIOs and OCTOSPI I/Os configuration* – *Section 4.2: OCTOSPI configuration for regular-command protocol* – *Section 5: OCTOSPI application examples* introduction – *Section 5.1.1: Using OCTOSPI in a graphical application* – *Figure 13: Executing code from memory connected to OCTOSPI2* – *STM32CubeMX: Project generation*
28-Aug-2020	6	Updated: – SMT32H72x/3x in *Table 1: Applicable products* and in the whole document – *Table 2: OCTOSPI main features* – STM32H7B0 in *Section 1.2.3: STM32H7A3/7B3/7B0 system architecture* – new *Section 1.2.4: STM32H72x/73x system architecture* – *Section 6.2.1: STM32 low-power modes*

Table 9. Document revision history (continued)

Date	Revision	Changes
27-Sep-2021	7	Updated: STM32U575/585 line added in – Table 1: Applicable products – Table 2: OCTOSPI and HSPI main features – Section 3.1.2: OCTOSPI delay block – Section 3.3.3: Memory-mapped mode – Section 4: OCTOSPI I/O manager – Section 9: Conclusion Added: – Section 2.2.5: STM32U5 series system architecture
13-Mar-2023	8	Updated: – OCTOSPI updated to OCTOSPI/HSPI – Table 1: Applicable products – Table 2: OCTOSPI and HSPI main features – Section 2.2: OCTOSPI, HSPI, and XSPI in a smart architecture – Section 2.2.2: STM32U5 series system architecture as an HSPI example – Figure 2: STM32U5 system architecture – Section 3: Octo/Hexadeca/XSPI interface description – Section 5.1.1: GPIOs, OCTOSPI/HSPI/XSPI I/Os configuration – Section 8: Supported devices Added: – STM32U5 series, STM32H562/563/573 line – Table 3: HSPI main features – Table 5: Instances on STM32U5 series devices – Section 2.2.3: STM32H7Rx/Sx system architecture as an XSPI example – Figure 3: STM32H7Rx/Sx system architecture – Section 3.1.2: HSPI pins and signal interface

Table 9. Document revision history (continued)

Date	Revision	Changes
11-Mar-2024	9	Updated: – Document title – OCTOSPI/HSPI updated to OCTOSPI/HSPI/XSPI – *Table 1: Applicable products* – *Chapter Table 2.* – *Section 2.2: OCTOSPI, HSPI, and XSPI in a smart architecture* – *Section 3.3: Two low-level protocols* – *Section 3.1.2: HSPI pins and signal interface* – *Section 3.4.3: Memory-mapped mode* – *Section 4: OCTOSPI and XSPI I/O managers* – *Section 5.1.1: GPIOs, OCTOSPI/HSPI/XSPI I/Os configuration* – *Section 5.1.2: Interrupts and clocks configuration* – *Figure 16: XSPI1 and XSPI2 clock scheme* – *Figure 21: STM32CubeMX - Setting PE13 pin to OCTOSPIM_P1_IO1 AF* – *Figure 22: STM32CubeMX - GPIOs setting window* – *Figure 23: STM32CubeMX - Setting GPIOs to very-high speed* – *Table 8: STM32CubeMX - Configuration of OCTOSPI parameters* Added: – STM32N6 series, STM32H7Rx/Sx line – STM32H562, STM32H563/573 lines – STM32H7R7/7S7, STM32H7R3/7S3 lines – *Figure : For the XSPI, the user can configure the XSPIM_CR registers in order to select the desired source signals for the configured port as shown in the figure below:* – *Figure : To enable the Multiplexed mode for an OCTOSPI/XSPI interface, the user must configure the OCTOSPIM_PnCR (n = 1 to 2) register and XSPIM_CR register in order to:* – *Figure 15: HSPI clock scheme* – *Figure 16: XSPI1 and XSPI2 clock scheme* – *Section 2.2.3: STM32H7Rx/Sx system architecture as an XSPI example* – *Section 3.2: HSPI and XSPI high-speed interfaces and calibration* – *Chapter 7: XSPI application example configuration with STM32CubeMX* Removed: – Section 2.2.2: STM32L5 series system architecture – Section 2.2.3: STM32H7A3/7B3/7B0 system architecture – Section 2.2.4: STM32H72x/73x system architecture – Section 2.2.6: STM32H5 system architecture – XSPI1 (replaced by XSPI) throughout the document – XSPIM_PnCR throughout the document – XSPI3 throughout the document – STM32N6 series throughout the document – *Figure 14: XSPI I/O manager configuration for three XSPI interfaces* – Cypress replaced by Infineon throughout the document