life.augmented

AN5050
Application note

Octal-SPI interface (OctoSPI) on STM32L4+ Series

Introduction

The growing demand for richer graphics, wider range of multimedia and other data-intensive content, is driving 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 the first MCU products in the market with the new integrated high throughput OctoSPI interface; STM32L4+ Series devices.

The OctoSPI interface enables the connection of the external compact-footprint Octal-SPI and the HyperBus™ high-speed volatile and non-volatile memories available today in the market. Thanks to its low-pin count, the OctoSPI interface allows easier PCB designs and lower costs. Its high throughput allows code execution and data storage.

Thanks to the OctoSPI's memory-mapped mode, the external memory can be accessed as if it was an internal memory allowing to the system masters (GP-DMA, LTDC, DMA2D, GFXMMU…) 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 interface on the STM32 MCUs and explains how to configure it in order to write and read external Octal-SPI and HyperBus™ memories. It describes some typical use cases to use OctoSPI interface and provides some practical examples on how to configure the OctoSPI depending on the type of the targeted memory.

Related documents

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

- STM32LRxxx and STM32LSxxx advanced Arm®-based 32-bit MCUs (RM0432)
- STM32L4Rxxx and STM32L4Sxxx datasheets
- AN4760 Quad-SPI (QSPI) interface on STM32 microcontrollers

Contents

1 Overview of STM32L4+ Series OctoSPI interface 6
1.1 OctoSPI availability and features across STM32L4+ Series 6
1.2 OctoSPI in a smart architecture 6
1.2.1 STM32L4Rxxx and STM32L4Sxxx system architecture 7
2 OctoSPI interface description 9
2.1 OctoSPI hardware interface 9
2.1.1 OctoSPI pins and signal interface 9
2.1.2 OctoSPI IO manager 9
2.2 Two low-level protocols 10
2.2.1 Regular-command mode 10
2.2.2 HyperBus™ mode 11
2.3 Three operating modes 12
2.3.1 Indirect mode 12
2.3.2 Status-flag polling mode 12
2.3.3 Memory-mapped mode 13
3 OctoSPI configuration 15
3.1 OctoSPI common configuration 15
3.1.1 GPIOs and OctoSPI IO manager configuration 15
3.1.2 Interrupts and clocks configuration 16
3.2 OctoSPI configuration for regular-command mode 17
3.3 OctoSPI configuration for HyperBus™ mode 18
3.4 Memory configuration 18
3.4.1 Octal-SPI memory device configuration 18
3.4.2 HyperBus™ memory device configuration 19
4 OctoSPI application examples 20
4.1 Implementation examples 20
4.1.1 Using OctoSPI in a graphical application 20
4.1.2 Executing from external memory: extend internal memory size 21
4.2 OctoSPI STM32CubeMX examples 22
4.2.1 Hardware description 22

4.2.2 Use case description 24
4.2.3 OctoSPI GPIOs and clocks configuration 25
4.2.4 Regular-command mode 33
4.2.5 HyperBus™ mode 45

5 Performance and power 49

5.1 How to get the best read performance 49
5.1.1 Read performance 49
5.2 Decreasing power consumption 50
5.2.1 STM32 low-power modes 50
5.2.2 Decreasing Octal-SPI memory's power consumption 51

6 Supported devices 52

7 Conclusion 53

8 Revision history 54

List of tables

Table 1. OctoSPI availability and features across STM32 families. 6
Table 2. OctoSPI memory-mapped mode Reading performance. 50
Table 3. OctoSPI peripheral state in different power modes on STM32L4Rxxx and STM32L4Sxxx 50
Table 4. Document revision history 54

List of figures

Figure 1. STM32L4Rxxx and STM32L4Sxxx system architecture 8
Figure 2. Example of connecting an Octal-SPI Flash memory and a HyperRAM™ memory to an STM32 device 10
Figure 3. Regular-command mode: octal DTR read operation example in Macronix mode 11
Figure 4. HyperBus™ mode: example of reading operation from HyperRAM™ 12
Figure 5. OCTOSPI1 and OCTOSPI2 clock scheme 17
Figure 6. OctoSPI graphic application use case 21
Figure 7. Executing code from memory connected to OCTOSPI2 22
Figure 8. Octal-SPI Flash and HyperRAM™ memories connection to the STM32L4R9AI device in the STM32L4R9I-EVAL board 23
Figure 9. Examples configuration: OCTOSPI1 set to regular-command mode and OCTOSPI2 set to HyperBus™ 24
Figure 10. STM32CubeMX: setting Octal I/O mode for OCTOSPI1 25
Figure 11. STM32CubeMX: mapping OCTOSPI1 signals to Port1 26
Figure 12. STM32CubeMX: setting PB0 pin to OCTOSPIM_P1_IO1 alternate function 27
Figure 13. STM32CubeMX - OCTOSPI1 GPIOs correct configuration 28
Figure 14. STM32CubeMX - configuration tab 28
Figure 15. OCTOSPI1 button in the configuration tab 29
Figure 16. STM32CubeMX - setting GPIOs to very high speed 29
Figure 17. STM32CubeMX: mapping OCTOSPI2 signals to port 2 30
Figure 18. STM32CubeMX: OCTOSPI2 GPIOs correct configuration 31
Figure 19. STM32CubeMX: enabling OCTOSPI1 global interrupt 31
Figure 20. STM32CubeMX: enabling OCTOSPI2 global interrupt 32
Figure 21. STM32CubeMX: system clock configuration 32
Figure 22. STM32CubeMX: OCTOSPI1 and OCTOSPI2 clock source configuration 33
Figure 23. OCTOSPI1 peripheral configuration in regular-command mode 34
Figure 24. OCTOSPI2 peripheral configuration in HyperBus™ mode 46

1 Overview of STM32L4+ Series OctoSPI interface

This section provides an overview of the OctoSPI interface availability across the different STM32 devices. It gives a clear explanation on the OctoSPI integration in the STM32 MCUs system architecture.

The STM32L4+ Series products are Arm®[(a)]-based.

1.1 OctoSPI availability and features across STM32L4+ Series

Table 1 summarizes the STM32 MCUs embedding the OctoSPI interface and details their related features.

Table 1. OctoSPI availability and features across STM32 families

OctoSPI features	Max AHB frequency (MHz)	Maximum OctoSPI speed (MHz)[(1)]			Max addressable space[(2)]	
		Regular-command mode		HyperBus™ mode		
		SDR	DTR	(DTR with DQS)	Memory-mapped	Indirect mode
STM32L4Rxxx/ STM32L4Sxxx	120	86	60	32	256 Mbytes	4 Gbytes

1. Maximum OctoSPI speed reported on product's datasheet. For more details on the OctoSPI maximum speed refer to the relevant device's datasheet.
2. 32-bit address mode should be used to reach the 256 Mbytes in memory-mapped mode and 4 Gbytes in indirect mode.

1.2 OctoSPI in a smart architecture

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

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

In addition, the OCTOSPI interface is integrated in a smart architecture that enables:

- All masters can access autonomously to external memory without any CPU intervention.
- Masters can read/write data from/to memory in SLEEP mode when the CPU is stopped.
- CPU as a master can access the OCTOSPI and then execute code from the memory.
- GP DMA can do transfers from OCTOSPI to other internal or external memories.
- Graphical DMA2D can directly build framebuffer using graphic primitives from the connected Octal-SPI Flash or HyperFlash™ memory.
- Graphical DMA2D can directly build framebuffer in Octal-SPI SRAM or HyperRAM™.
- GFXMMU as a master can autonomously access the OCTOSPI.
- LTDC can fetch framebuffer directly from the memory that is connected to the OCTOSPI.

1.2.1 STM32L4Rxxx and STM32L4Sxxx system architecture

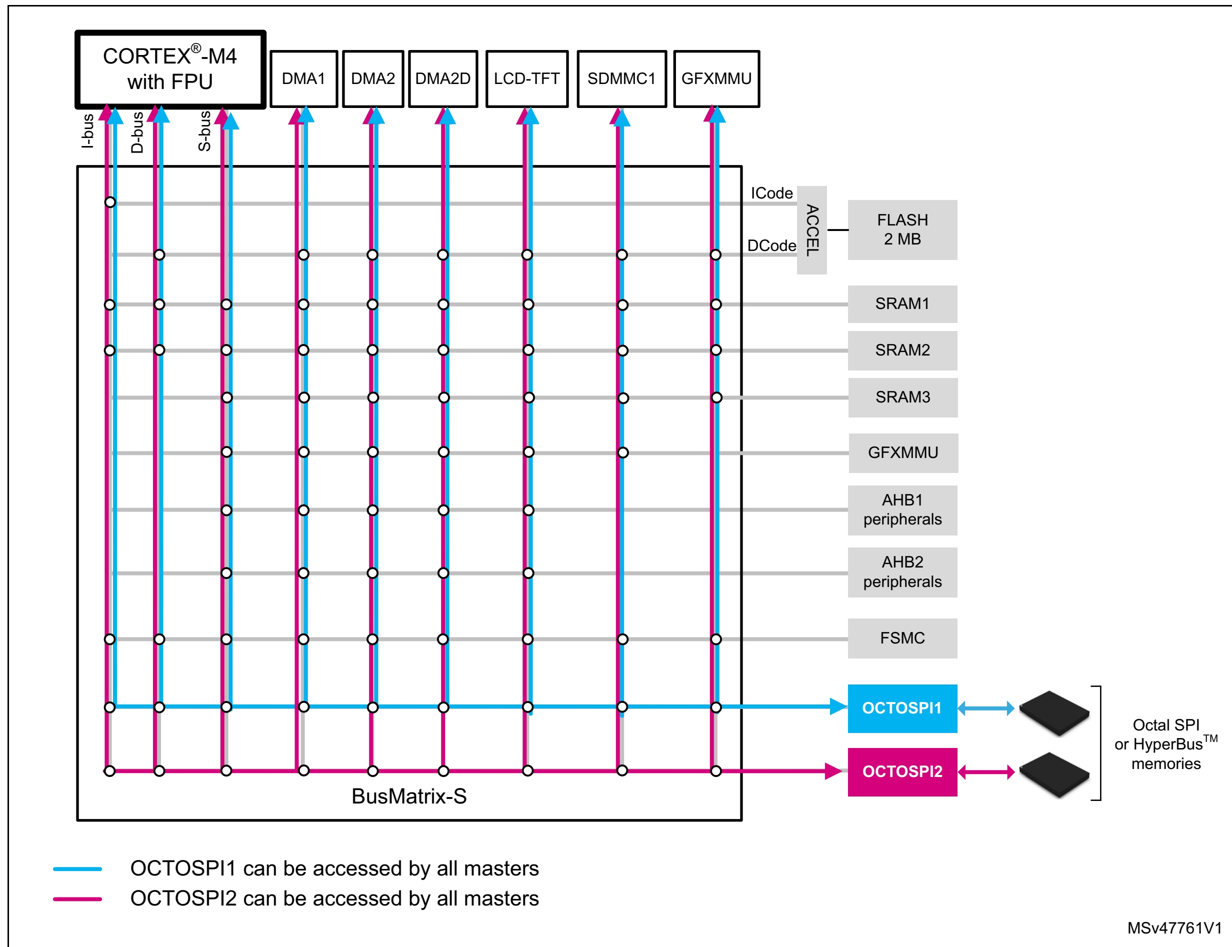
The STM32L4Rxxx and STM32L4Sxxx system architecture consists mainly of a 32-bit multilayer AHB bus matrix that interconnects nine masters and eleven slaves.

The system of the STM32L4Rxxx and STM32L4Sxxx devices integrates the OCTOSPI peripheral as described below:

- Two OCTOSPI slaves: OCTOSPI1 and OCTOSPI2. Each of them is mapped on a dedicated AHB layer.
- Each OCTOSPI slave is 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 0x90000000 to 0x9FFFFFFF
  - OCTOSPI2 addressable space is from 0x70000000 to 0x7FFFFFFF.
- 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.

Figure 1 shows OCTOSPI1 and OCTOSPI2 slaves interconnection in the STM32L4Rxxx and STM32L4Sxxx system architecture.

Figure 1. STM32L4Rxxx and STM32L4Sxxx system architecture



2 OctoSPI interface description

The OctoSPI is a serial interface which allows communication on 8 data lines between a host (STM32) and an external slave device (like a 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.

2.1 OctoSPI hardware interface

The OctoSPI provides a flexible hardware interface, which enables the support of multiple hardware configurations. It supports the Single-SPI (traditional SPI), Dual-SPI, Quad-SPI, Dual Quad-SPI and Octal-SPI. The flexibility of the OctoSPI's hardware interface permits the connection of several serial memories available in the market.

2.1.1 OctoSPI pins and signal interface

The OctoSPI interface uses up to eleven lines:

- OCTOSPI_NCS line for chip select
- OCTOSPI_CLK line for clock
- OCTOSPI_DQS line for data strobe
- OCTOSPI_IO[0...7] eight lines for data

Note: The HyperBus™ protocol supports single-ended clocks with 3 V signals and differential clock with 1 V8 signals, please note that only single ended clock is supported by OctoSPI.

Figure 2 shows OctoSPI interface signals.

2.1.2 OctoSPI IO manager

The OctoSPI IO manager allows the user to set a fully programmable pre-mapping of the OCTOSPI1 and OCTOSPI2 signals. Any OCTOSPIM_Pn_x port signal can be mapped independently to OCTOSPI1 or OCTOSPI2.

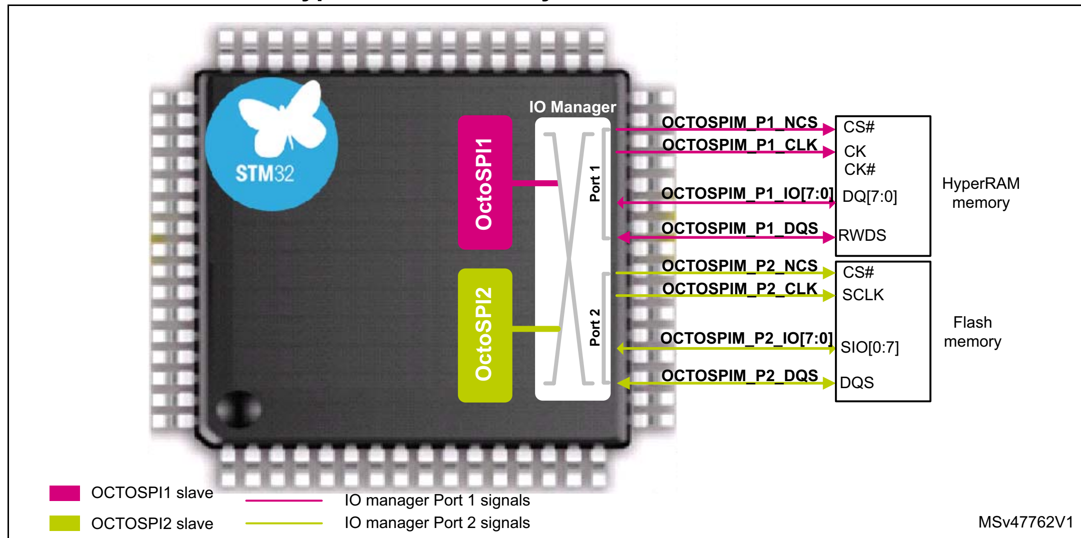
By default, after reset all the signals of the OCTOSPI1 and OCTOSPI2 are mapped respectively on port 1 and on port 2.

For instance when two external memories are used, a HyperRAM™ can be connected to Port1 and an Octal-SPI Flash can be connected to Port2 as shown in *Figure 2*. 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.

Figure 2 shows an Octal-SPI Flash memory and a HyperRAM™ memory connected to the STM32 MCU using the OctoSPI interface. Thanks to the OctoSPI IO manager, the HyperRAM™ memory can be linked to OCTOSPI1 and the Flash memory can be linked to OCTOSPI2 and vice versa.

Figure 2. Example of connecting an Octal-SPI Flash memory and a HyperRAM™ memory to an STM32 device



2.2 Two low-level protocols

The OctoSPI interface can operate in two different low-level protocols, the regular-command mode and the HyperBus™ mode. Each protocol supports three operating modes: the indirect mode, the status-flag polling mode, and the memory-mapped mode.

2.2.1 Regular-command mode

The regular-command mode is the classical frame format where the OctoSPI communicates with the external memory device by using commands where each command can include up to five phases. The external memory device can be an SPI, Dual-SPI, Quad-SPI, Dual Quad-SPI or Octal-SPI memory.

Flexible frame format and hardware interface

The OctoSPI 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:

- Instruction phase: this phase can be set to send a one, two, three or four bytes instruction (SDR or DTR). This phase can send instructions using the Single-SPI (2 lines), Dual-SPI, Quad-SPI or Octal-SPI mode.
- Address phase: this phase can be set to send one, two, three or four bytes address. This phase can send addresses using the Single-SPI (2 lines), Dual-SPI, Quad-SPI or Octal-SPI mode.
- Alternate-bytes phase: this phase can be set to send one, two, three or four alternate-bytes. This phase can send alternate-bytes using the Single-SPI (2 lines), Dual-SPI, Quad-SPI or Octal-SPI mode.
- Dummy-cycles phase: this phase can be set to 0 to up to 31 cycles.
- Data phase: for indirect or automatic-polling mode, the number of bytes to be sent/received is specified in the OCTOSPI_DLR register. For memory-mapped mode the bytes are sent/received following any AHB access request. This phase can

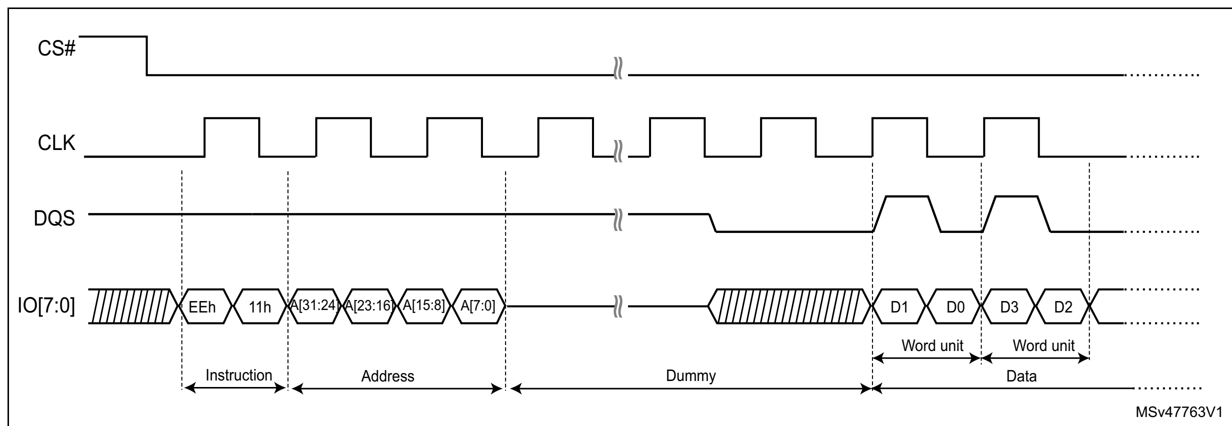
send/receive data using the Single-SPI (2 lines), Dual-SPI, Quad-SPI, Dual Quad-SPI or Octal-SPI mode.

Any of these phases can be configured to be skipped. *Figure 3* 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.

Figure 3. Regular-command mode: octal DTR read operation example in Macronix mode



2.2.2 HyperBus™ mode

The OctoSPI supports the HyperBus™ protocol which enables the communication with HyperRAM™ and HyperFlash™ memories.

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

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

The HyperBus™ frame is composed of two phases:

- Command/address phase: during this phase, the OctoSPI sends 48 bits (CA[47:0]) over IO[7:0] to specify the operations to be performed with the external device.
- Data phase: during this phase, the OctoSPI performs data transactions from/to the memory.

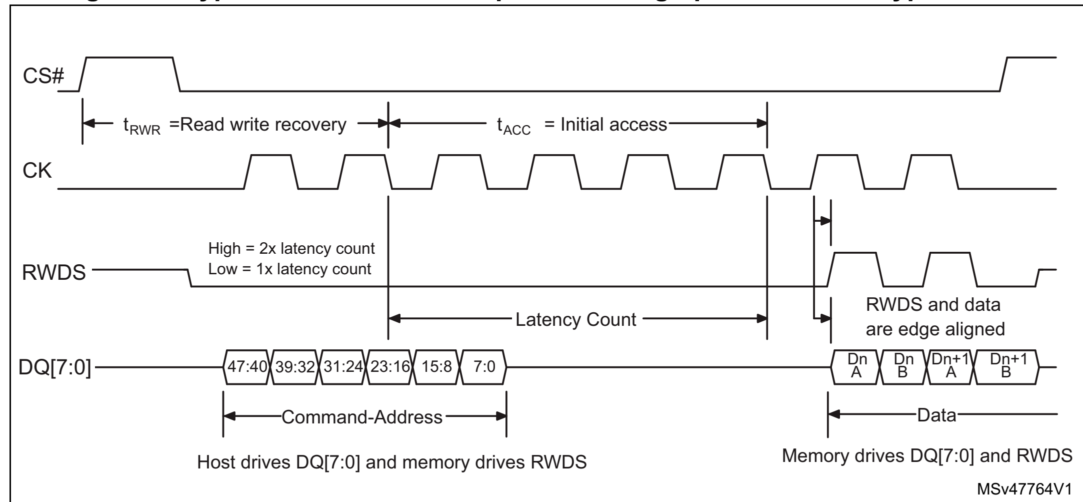
During the command/address (CA) phase, the read-recovery time (RWDS) is used by the HyperRAM™ memory to indicate if an additional initial access latency has to be inserted or not. If RWDS was *low* during the CA period, only one latency count is inserted (t_{ACC} initial access). If RWDS was *high* during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

Figure 4 illustrates an example of a HyperBus™ read operation.

Figure 4. HyperBus™ mode: example of reading operation from HyperRAM™



Depending on the application needs, the OctoSPI 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: should be used to access to the memory's register space, which is useful for memory configuration.

2.3 Three operating modes

Whatever the used low-level protocol, the OctoSPI can operate in the three operating modes: the indirect mode, the status-flag polling mode and the memory-mapped mode.

2.3.1 Indirect mode

The indirect mode is used either for HyperBus™ or for regular-command low-level protocols. The indirect mode is used in below cases:

- For reading, writing or erasing operations.
- If there is no need for AHB masters to access autonomously the OctoPI peripheral (available in memory-mapped mode).
- For all the operations to be performed through the OctoSPI data register using CPU or using DMA.
- To configure the external memory device.

2.3.2 Status-flag polling mode

The status-flag 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 status-flag 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 periodically reads the memory's register and checks if a bit or 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 polls the status register inside the memory while the CPU continues the execution, an interrupt can be generated when the memory operation has finished.
  – Example: this mode is commonly used to wait for an ongoing memory operation (programming/erasing). The OctoSPI in status-flag 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).

2.3.3 Memory-mapped mode

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

- For reading and writing operations.
- To use the external memory device exactly like an internal memory, so any AHB master can access it autonomously.
- For code execution from 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 masters to access to 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 peripheral is able to manage up to 256 Mbytes of memory space:

- OCTOSPI1 addressable space: from 0x90000000 to 0x9FFFFFFF (256 Mbytes)
- OCTOSPI2 addressable space: from 0x70000000 to 0x7FFFFFFF (256 Mbytes)

Starting memory-mapped read or write operation

A memory-mapped operation is started:

- As soon as there is an AHB master read request to an address in the range defined by DEVSIZE
- As soon as there is an AHB master write request to an address in the range defined by DEVSIZE

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

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

Note: When programming a Flash memory in memory-mapped mode the application must wait until the programming operation finishes. Since it is not possible to poll on the write in

progress memory flag in memory-mapped mode, the application must insert a delay corresponding to the operation duration.

Note: Reading the OCTOSPI_DR data register in memory-mapped mode has no meaning and returns 0.

The data length register OCTOSPI_DLR has no meaning in memory-mapped mode.

Execute in place (XIP)

The OCTOSPI supports execution in place (XIP) thanks to its integrated prefetch buffer. The XIP permits to execute the code directly from the external memory device. The OctoSPI anticipates the next CPU access and loads the byte in advance at the following address. If the subsequent access is indeed made at a continuous address, the access is completed faster since the value is already prefetched.

Send instruction only once (SIOO)

The SIOO feature permits the reduction of 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, then for the next accesses only the address is sent.

3 OctoSPI configuration

In order to enable the read or write form/to external memory, the application must configure the OctoSPI peripheral and the connected memory device.

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

- OctoSPI common configuration steps:
  - GPIOs and OctoSPI IO manager configuration
  - Interrupts and clock configuration
- OctoSPI specific configuration steps:
  - OctoSPI low-level protocol specific configurations (regular-command or HyperBus™)
  - Memory device configuration

The following subsections describe all needed OctoSPI configuration steps to enable the communication with external memories.

3.1 OctoSPI common configuration

This section describes the common steps needed to configure the OctoSPI peripheral regardless of the used low-level protocol (regular-command mode or HyperBus™ mode).

Note: It is recommended to reset the OctoSPI peripheral before starting a configuration. This action also guarantees that the peripheral is in reset state.

3.1.1 GPIOs and OctoSPI IO manager 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 or Octal-SPI).

In Octal-SPI mode, when only one external memory is connected, ten GPIOs are needed. An additional GPIO for DQS is optional for regular-command mode and mandatory for HyperBus™ mode.

When two external octal memories are connected, each memory should be connected to an IO manager port, which requires up to 22 GPIOs.

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

The OctoSPI GPIOs should 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 relevant datasheet.

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

GPIOs configuration using STM32CubeMX tool

Using the STM32CubeMX tool is a very simple, easy and rapid way to configure the OctoSPI peripheral and its GPIOs. STM32CubeMx permits to generate a project with a preconfigured OctoSPI. *Section 4.2.3* provides a guide on how to configure the OctoSPI GPIOs.

OctoSPI IO manager configuration

By default, after reset all the signals of the OCTOSPI1 and OCTOSPI2 are mapped respectively to port 1 and to port 2.

3.1.2 Interrupts and clocks configuration

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

Enabling interrupts

In the STM32L4Rxxx and STM32L4Sxxx devices, each OctoSPI peripheral has its dedicated global interrupt connected to the NVIC.

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

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

Clock configuration

In the STM32L4Rxxx and STM32L4Sxxx devices both OCTOSPI1 and OCTOSPI2 peripherals have the same clock source. 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 clock and the prescaler.

$$OCTOSPIx_CLK = F_{Clock_source} / (PRESCALER + 1)$$

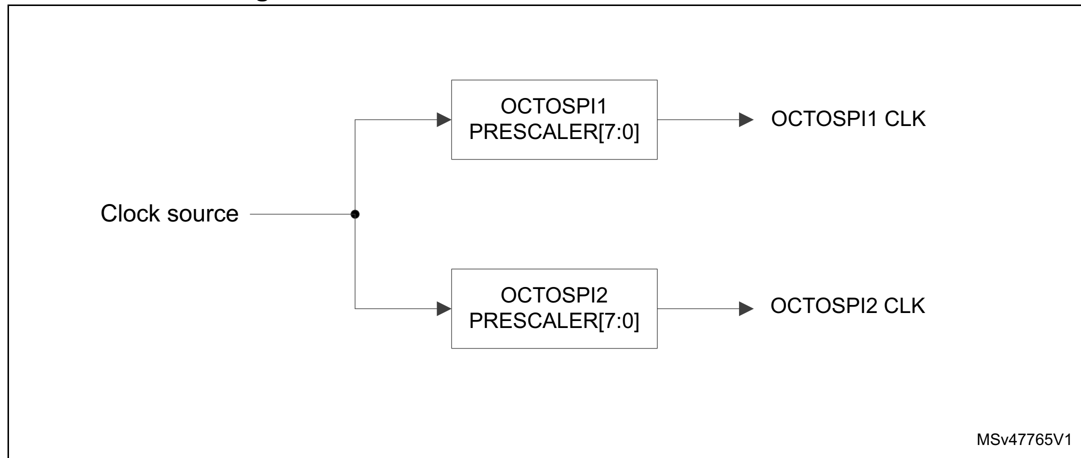
For instance when the PRESCALER[7:0] is set to 2 then
$OCTOSPIx_CLK = F_{Clock_source} / 3$

On STM32L4Rxxx and STM32L4Sxxx devices any of the three different clock sources, SYSCLK, MSI or PLLQ, can be used for OctoSPI clock source.

Note: User should 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.

Figure 5 illustrates the OCTOSPI1 and OCTOSPI2 clock scheme.

Figure 5. OCTOSPI1 and OCTOSPI2 clock scheme



3.2 OctoSPI configuration for regular-command mode

The regular-command mode should be used when an external-SPI, Quad SPI, Dual Quad-SPI or Octal-SPI memory is connected to the STM32.

The user should configure the following OctoSPI parameters:

- Memory type: Micron mode, Macronix mode or Macronix RAM mode.
- Device size: number of bytes in device = $2^{[DEVSIZE+1]}$.
- Chip-select high time: the CSHT should be configured according to the memory datasheet. It is commonly named CS# Deselect Time. It represents the period between two successive operations where the memory is deselected.
- Clock mode: low (Mode 0) or high (Mode 3).
- Clock prescaler should be set to get the targeted operating clock.
- DHQC is 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
- 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 ½ OCTOSPI clock cycle enabling more relaxed input timings.

3.3 OctoSPI configuration for HyperBus™ mode

The HyperBus™ mode should be used when an external HyperRAM™ or HyperFlash™ memory is connected to the STM32.

The user should configure the following OctoSPI parameters:

- Memory type: HyperBus™ mode.
- Device size: number of bytes in device = $2^{[DEVSIZE+1]}$.
- Chip-select high time: the CSHT should be configured according to the memory datasheet. It is commonly named CS# Deselect Time. It represents the period between two successive operations where the memory is deselected.
- Clock mode low (Mode 0) or high (Mode 3).
- Clock prescaler should be set to get the targeted operating clock.
- DTR (DDR) mode must be enabled for HyperBus™
- DHQC is 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.
- SSHIFT must be disabled since HyperBus™ is operating in DDR mode.
- Read-write recovery time (t_{RWR}): it is used only for HyperRAM™ and it should be configured according to the memory device.
- Initial latency (t_{ACC}): should be configured according to the memory device and the operating frequency.
- Latency mode: fixed or variable latency.
- Latency on write access: enabled or disabled.

3.4 Memory configuration

The external memory device should be configured depending on the targeted operating mode. This section describes some commonly needed configurations for HyperBus™ and Octal-SPI memories.

3.4.1 Octal-SPI memory device configuration

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

- Set the dummy cycles according to the operating speed (see relevant memory device datasheet).
- Enable the Octal mode which enables the communication in Octal I/O mode.
- Enable DTR mode which enables the communication in DTR mode.

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

3.4.2 HyperBus™ memory device configuration

The HyperBus™ memory device should be configured depending on the targeted OctoSPI operating mode. To configure HyperBus™ memory device, the HyperBus™ register mode should be selected in order to address the memory register space. This can be done by setting MTYP[2:0] to 0b101 in the OCTOSPI_DCR1 register.

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

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

4 OctoSPI application examples

This section provides some typical OctoSPI implementation examples and STM32CubeMX examples using the STM32L4R9I-EVAL board.

4.1 Implementation examples

This section provides some typical OctoSPI use case examples. The following examples are described in this section:

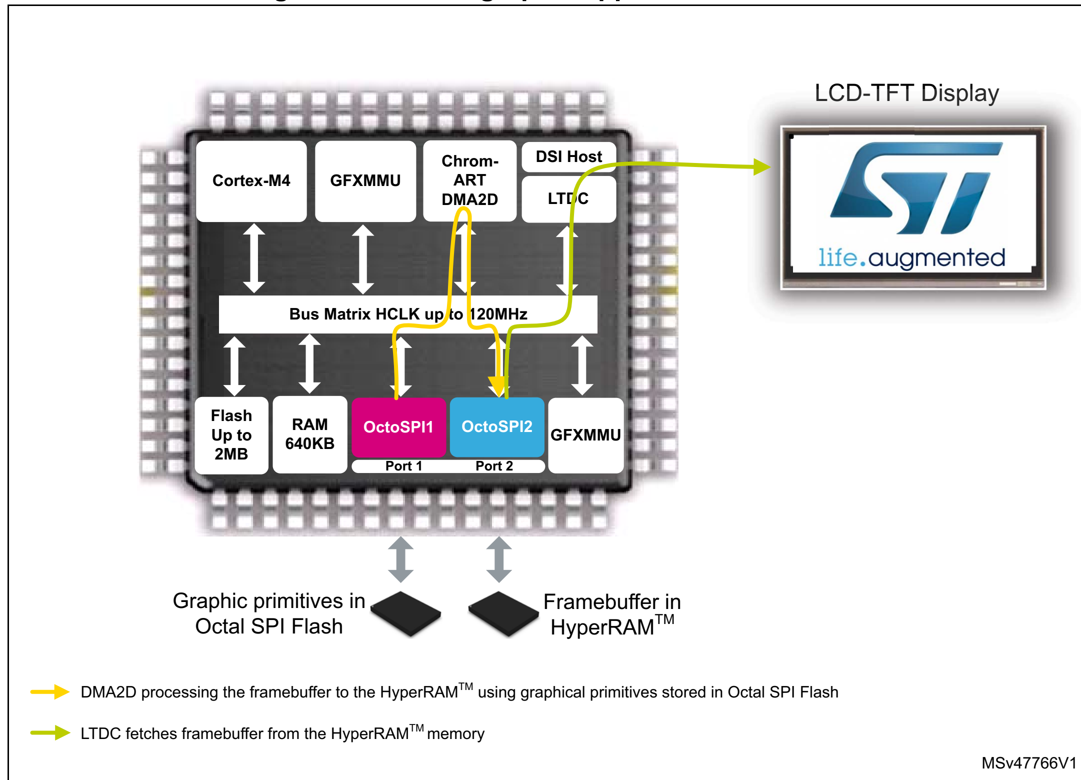
- OctoSPI usage in a graphical application
- Code execution from OctoSPI memory

4.1.1 Using OctoSPI in a graphical application

The STM32L4Rxxx and STM32L4Sxxx devices embed two independent OctoSPI peripherals that enable the connection of two external memories. This configuration is ideal for graphical applications where:

- An Octal-SPI Flash memory is connected to OCTOSPI1 which is used to store graphical primitives.
- A HyperRAM™ memory is connected to OCTOSPI2 which is used to build framebuffer.
- OCTOSPI1 should be set to regular-command mode in order to communicate with the Octal Flash memory.
- OCTOSPI2 should be set to HyperBus™ mode in order to communicate with HyperRAM™ memory.
- Both OCTOSPI1 and OCTOSPI2 should be configured to memory-mapped mode.
- Any AHB master such as CPU, LTDC, DMA2D or GFXMMU can autonomously access to both memories exactly like an internal memory.

Figure 6. OctoSPI graphic application use case



4.1.2 Executing from external memory: extend internal memory size

Using the external Octal-SPI memory permits to extend the total application's available memory space.

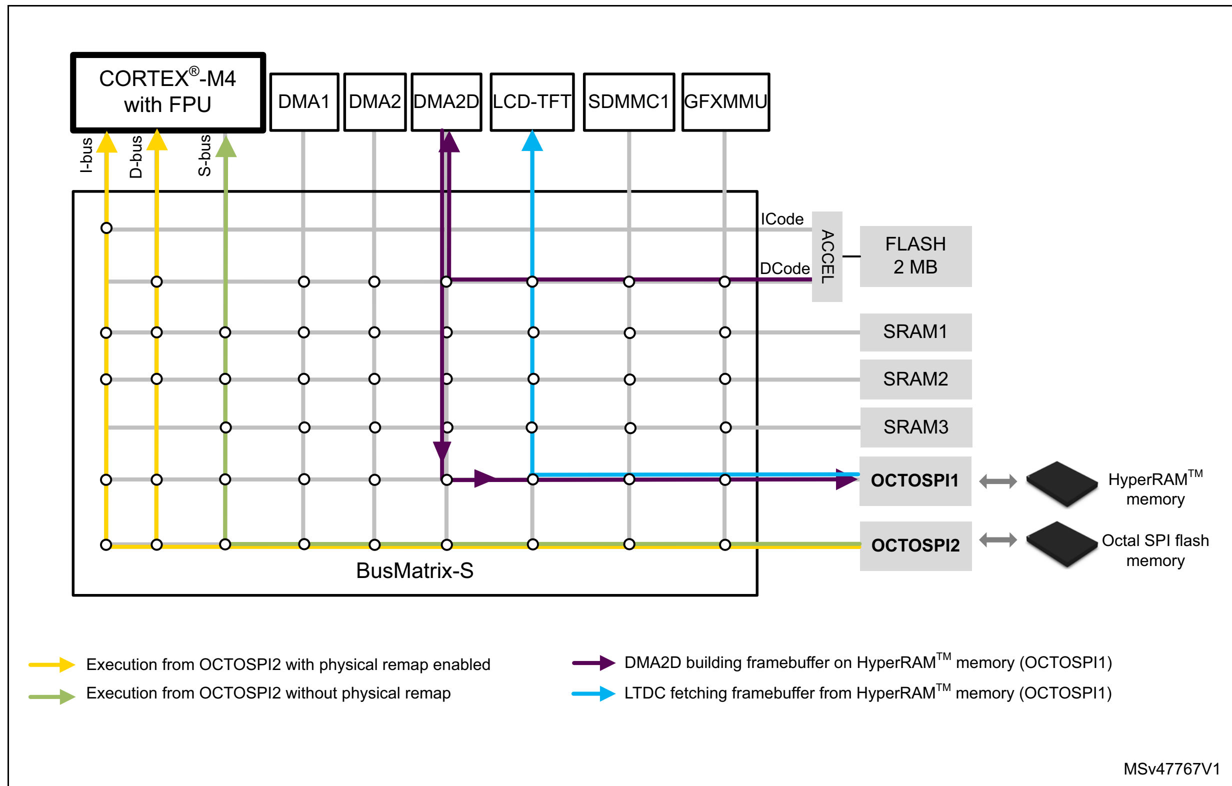
To execute code from an external memory:

- The application's code should be placed in the external memory.
- The OctoSPI should be configured in memory-mapped mode during the system's initialization before jumping to the Octal-SPI memory code.

As illustrated in *Figure 7*, 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 0x00000000 can be enabled for OCTOSPI2 allowing execution through I-bus and D-bus.

Figure 7. Executing code from memory connected to OCTOSPI2



4.2 OctoSPI STM32CubeMX examples

This section provides two examples of basic OctoSPI configuration based on the STM32L4R9I-EVAL board:

- Regular-command low-level protocol in indirect mode for programming and in memory-mapped mode for reading from the Octal-SPI Flash memory.
- HyperBus™ low-level protocol in memory-mapped mode enabling reading and writing from/to the HyperRAM™ memory.

Note: The regular-command mode example can be easily ported to the STM32L4R9I-DISCO board which embeds a MACRONIX MX25LM51245GXDI00 Octal-SPI Flash memory (same memory as STM32L4R9I-EVAL board).

For more details on the STM32L4R9I-EVAL and STM32L4R9I-DISCO boards, refer to UM2248 and UM2271 respectively.

4.2.1 Hardware description

The STM32L4R9I-EVAL board used for these examples embeds two external memories:

- The MACRONIX MX25LM51245GXDI0A Octal-SPI Flash memory connected to Port1.
- The ISSI IS66WVH8M8BLL-100BLI HyperRAM™ memory connected to Port2.

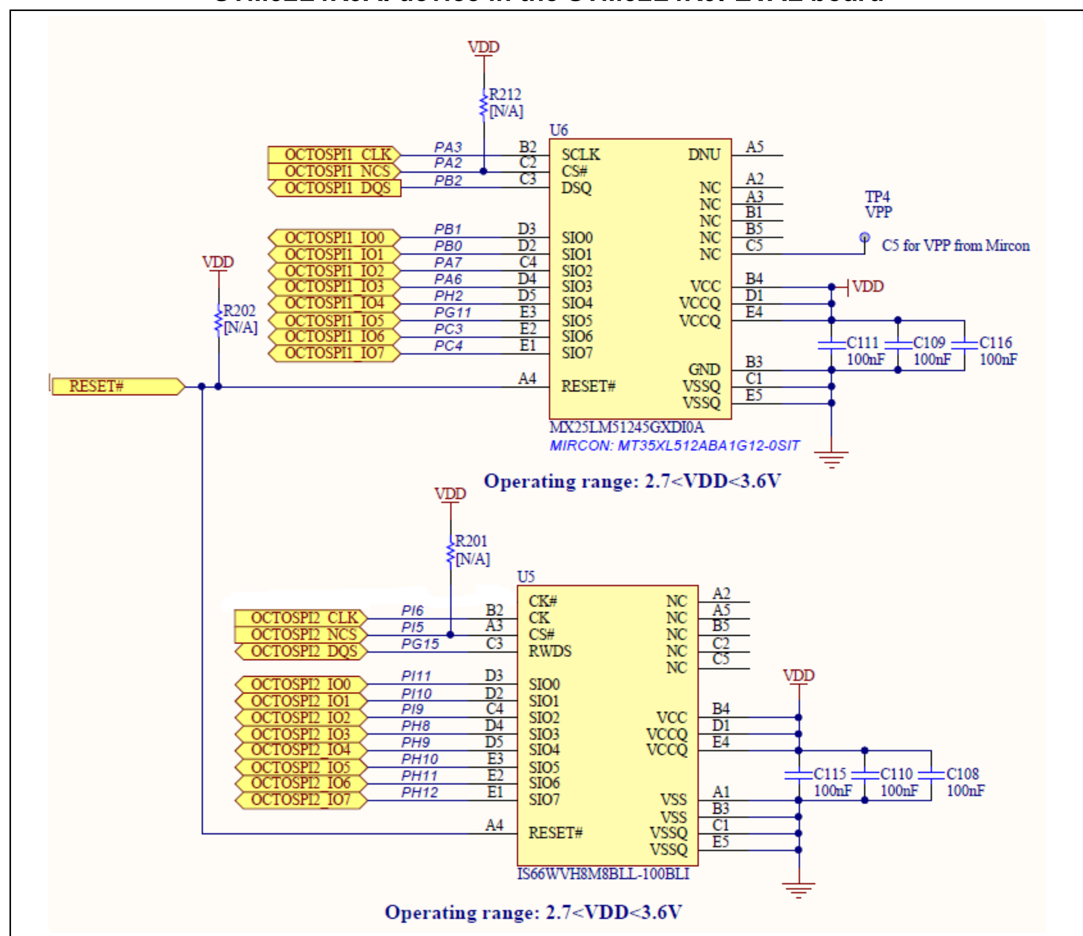
As shown in *Figure 8*, each memory is connected to the STM32L4R9AI device using eleven pins:

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

The OCTOSPI_RESET reset pin permits to reset the memories, it is connected to the global MCU reset pin (NRST).

Figure 8 shows MACRONIX MX25LM51245GXDI0A and ISSI IS66WVH8M8BLL-100BLI memories connected to the STM32L4R9AI MCU.

Figure 8. Octal-SPI Flash and HyperRAM™ memories connection to the STM32L4R9AI device in the STM32L4R9I-EVAL board



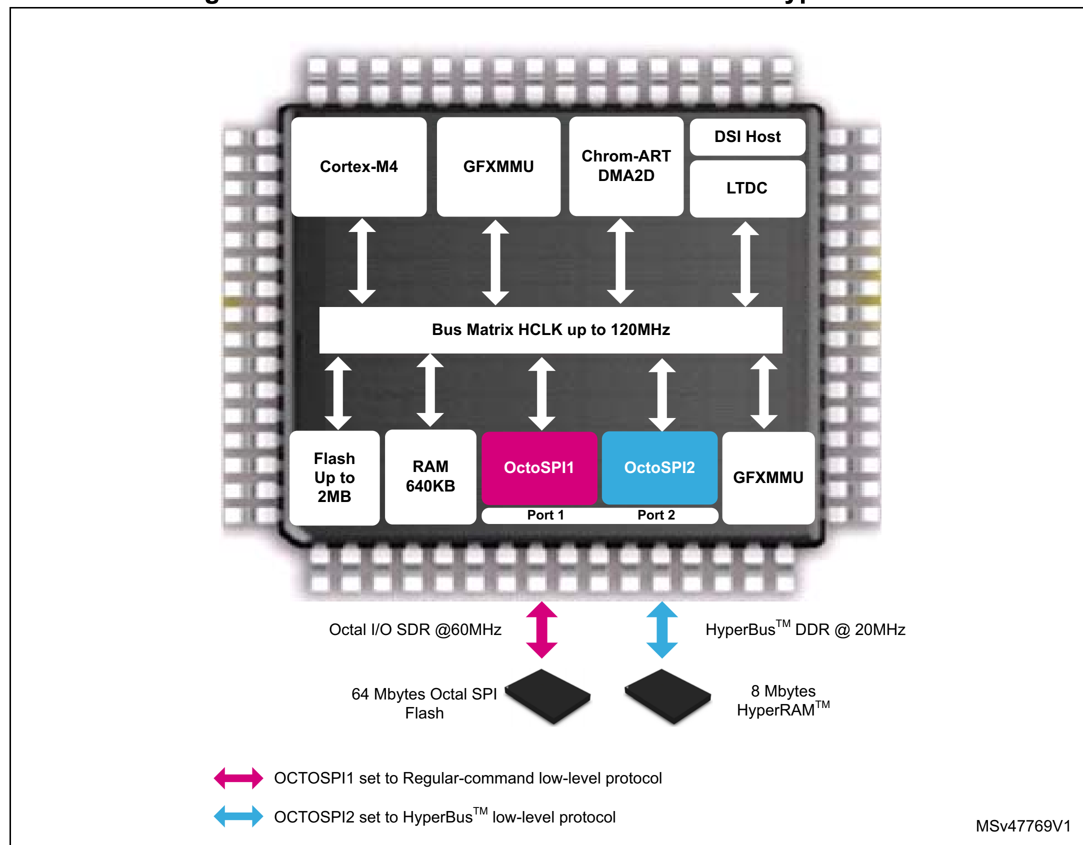
4.2.2 Use case description

The adopted configuration for each example is:

- Regular-command mode example:
  - OCTOSPI1 signals are mapped to port 1 (Nor Flash) so OCTOSPI1 has to be set to regular-command mode.
  - SDR Octal I/O mode (without DQS) with OCTOSPI1 running @ 60 MHz.
  - Programming the memory in indirect mode and reading in memory-mapped mode.
- HyperBus™ mode example:
  - OCTOSPI2 signals are mapped to port 2 (HyperRAM™) so OCTOSPI2 has to be set to HyperBus™ mode.
  - DDR Octal I/O mode (with DQS) with OCTOSPI2 running @ 20 MHz.
  - Memory-mapped mode for reading and writing.

Figure 9 illustrates the OctoSPI configuration for each example.

Figure 9. Examples configuration: OCTOSPI1 set to regular-command mode and OCTOSPI2 set to HyperBus™



The two examples described later on, regular-command and HyperBus™, have some common configurations based on STM32CubeMX:

- GPIO and OctoSPI IO manager configuration.
- Interrupts and clock configuration.

Each example has the following specific configurations:

- OctoSPI peripheral configuration.
- Memory device configuration.

4.2.3 OctoSPI GPIOs and clocks configuration

This section describes the needed steps to configure the OCTOSPI1 and OCTOSPI2 GPIOs and clocks. These steps are the same whatever the low-level protocol used (regular-command mode or HyperBus™ mode).

I. STM32CubeMX: GPIOs configuration

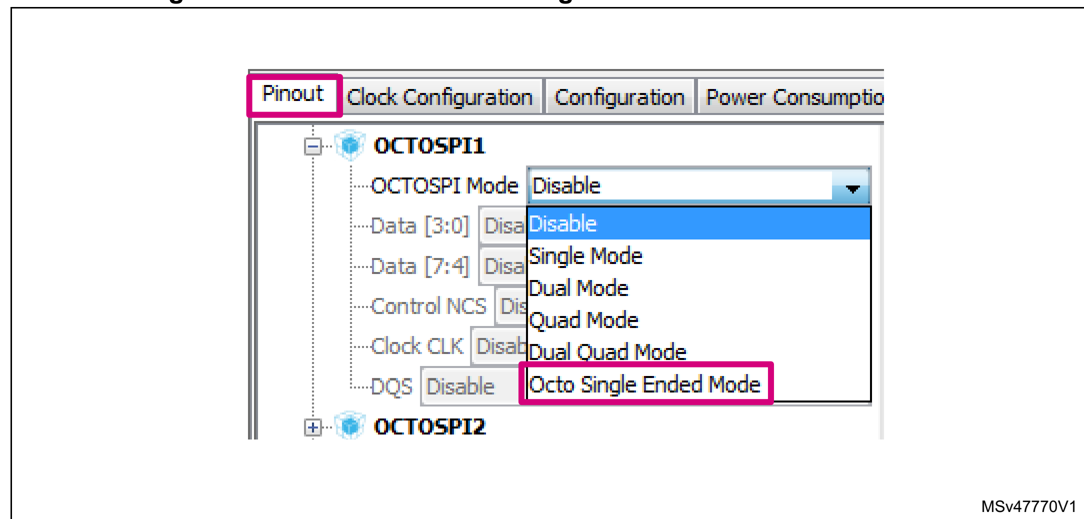
As shown in *Figure 8*, the MACRONIX MX25LM51245GXDI00 Octal-SPI Flash memory is connected to the STM32L4R9 MCU through the OctoSPI IO manager port 1 while the ISSI IS66WVH8M8BLL-100BLI HyperRAM™ is connected through port 2. Based on this hardware implementation the user should configure all the GPIOs shown in *Figure 8*.

A. STM32CubeMX: OCTOSPI1 GPIOs configuration

Once that the STM32CubeMX project is created for the STM32L4R9AI product, the user must follow the steps below:

1. Select the pinout tab and uncollapse the OCTOSPI1 as shown in *Figure 10*.
2. Configure the Octal I/O mode for OCTOSPI1 by selecting **"Octo Single Ended Mode**" in the listed hardware configurations shown in *Figure 10*.

Figure 10. STM32CubeMX: setting Octal I/O mode for OCTOSPI1

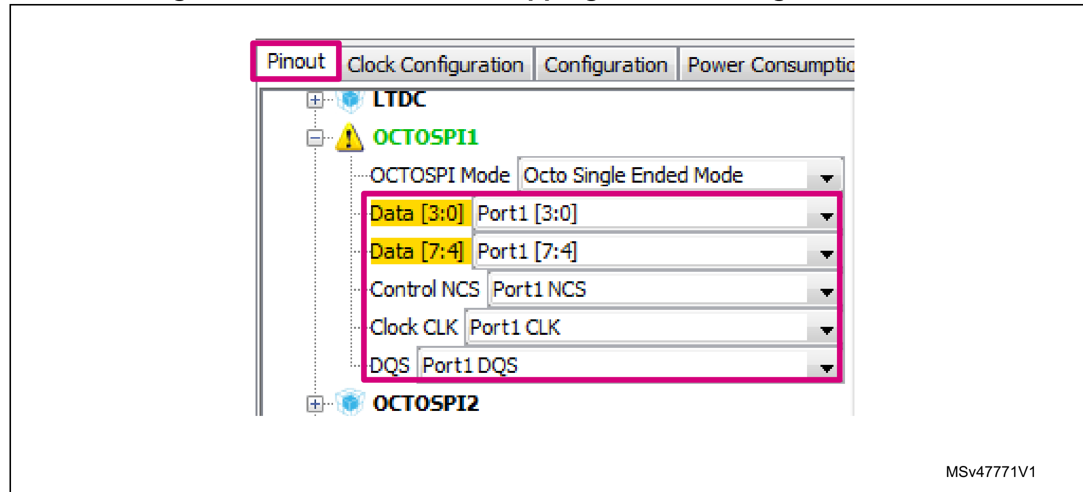


1. Pink color highlights the key items in the figure.

3. Mapping OCTOSPI1 signals to port 1:
   – Set the OCTOSPI1 LSB data signals Data[3:0] to port 1 [3:0]
   – Set the OCTOSPI1 MSB data signals Data[7:4] to port 1 [7:4]
   – Set the OCTOSPI1 chip select to port 1 NCS
   – Set the OCTOSPI1 clock to port 1 CLK
   – Set the OCTOSPI1 data strobe to port 1 DQS.[a]

Figure 11 shows how to map OCTOSPI1 signals to port 1.

Figure 11. STM32CubeMX: mapping OCTOSPI1 signals to Port1



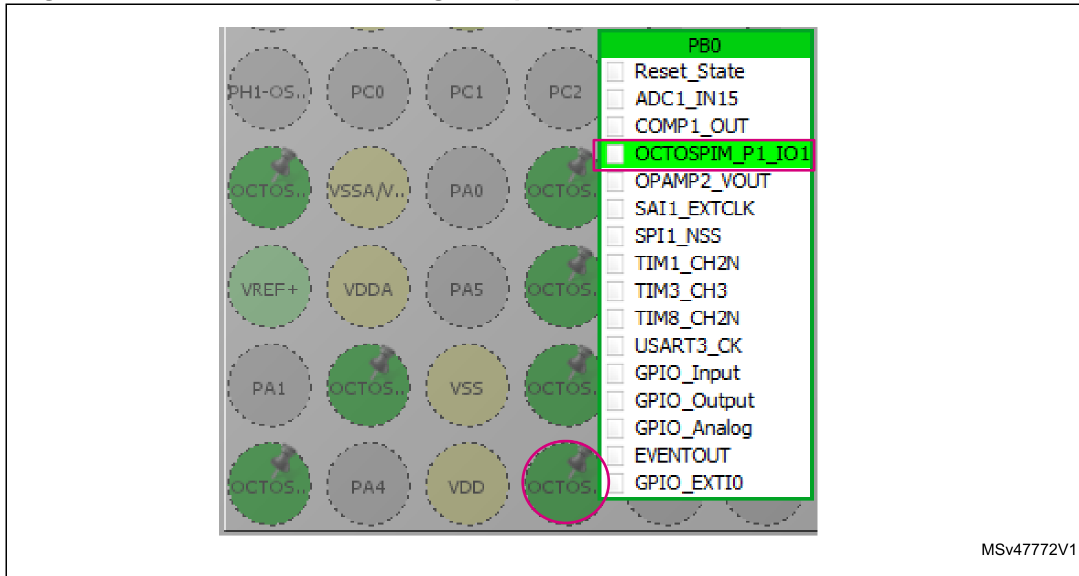
1. Pink color highlights the key items in the figure.

The user must make sure that the configured GPIOs match the memory connection as shown in *Figure 8*. If the configuration is not correct, he must manually configure all the GPIOs, one by one, by clicking on each pin directly.

a. DQS pin is used in the STM32L4R9I-EVAL board and configured in this example, but it is optional for regular-command mode. Some memory commands require DQS while some others do not require it (see MACRONIX MX25LM51245GXDI00 datasheet).

Figure 12 shows how to manually configure the PB0 pin to OCTOSPIM_P1_IO1 alternate function.

Figure 12. STM32CubeMX: setting PB0 pin to OCTOSPIM_P1_IO1 alternate function

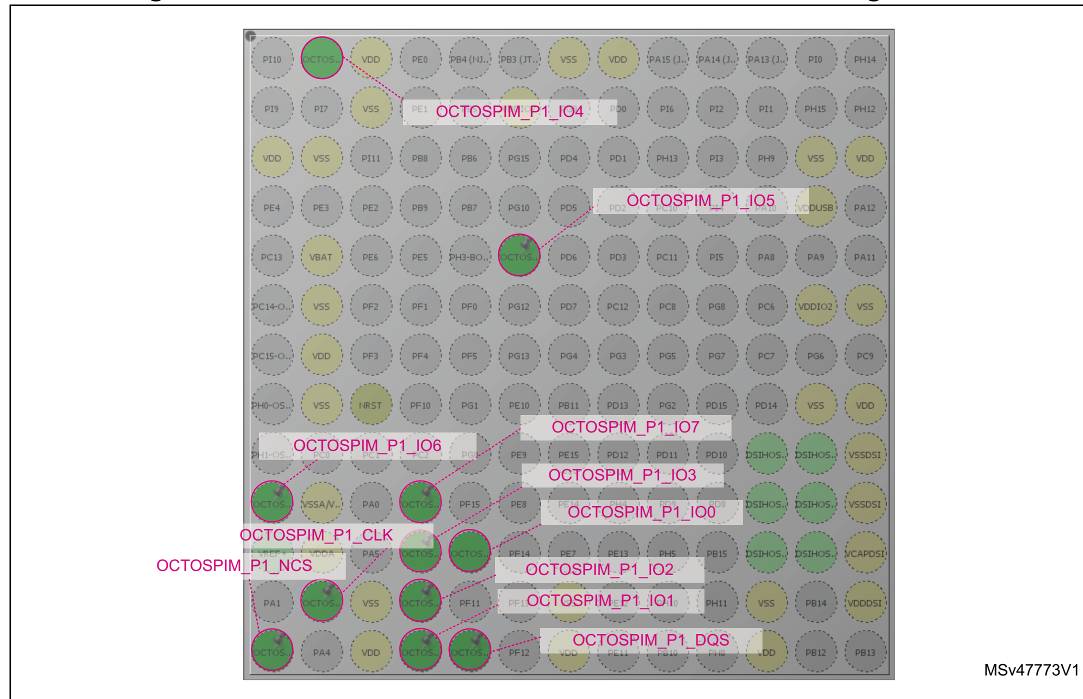


1. Pink color highlights the key items in the figure.

4. OCTOSPI1 GPIOs correct configuration

Figure 13 shows the OCTOSPI1 GPIOs properly configured to port 1.

Figure 13. STM32CubeMX - OCTOSPI1 GPIOs correct configuration

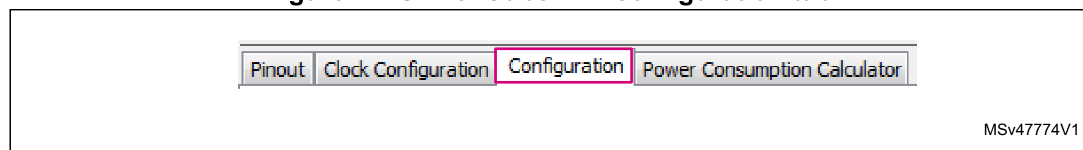


1. Pink color highlights the key items in the figure.

Once that all the OctoSPI GPIOs are properly set, the status of the OCTOSPI1 should be OK in the OCTOSPI1 configuration button as illustrated in *Figure 15*. Else a red cross is displayed in the configuration tab and an error message appears.

5. Configuring OCTOSPI1 GPIOs to very high speed
   a) Select the configuration tab shown in *Figure 14*.

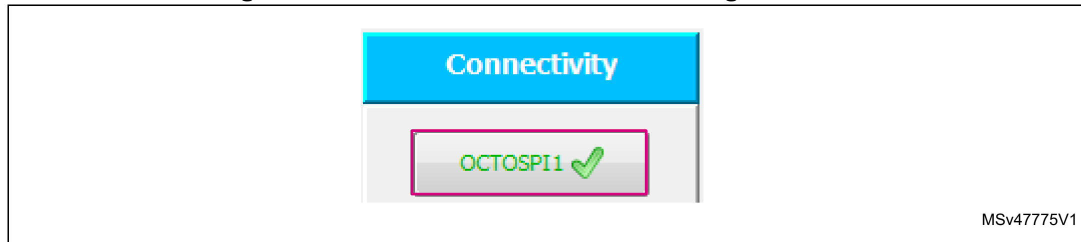
Figure 14. STM32CubeMX - configuration tab



1. Pink color highlights the key items in the figure.

   b) Click on the OCTOSPI1 button in the configuration tab as shown in *Figure 15*.

Figure 15. OCTOSPI1 button in the configuration tab

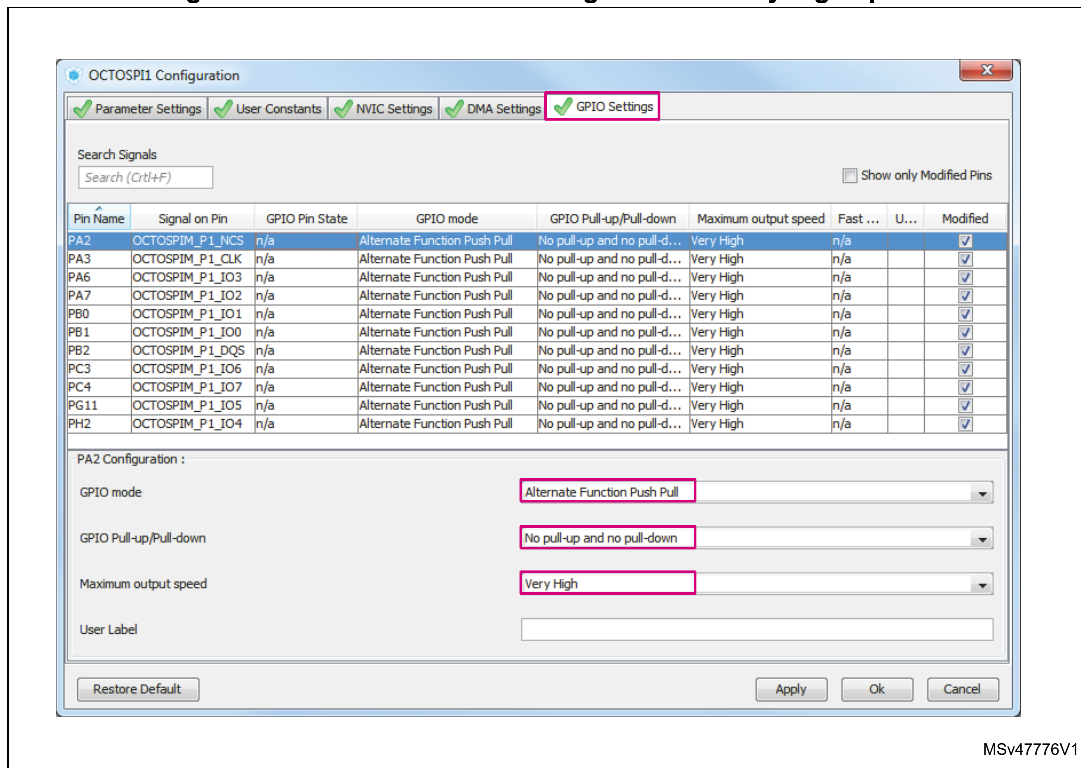


1. Pink color highlights the key items in the figure.

c) In the OCTOSPI1 configuration window, select the GPIO settings tab then make sure that all the GPIOs output speed is set to “very high” as shown in *Figure 16*.

d) Click on Apply and then OK.

Figure 16. STM32CubeMX - setting GPIOs to very high speed



1. Pink color highlights the key items in the figure.

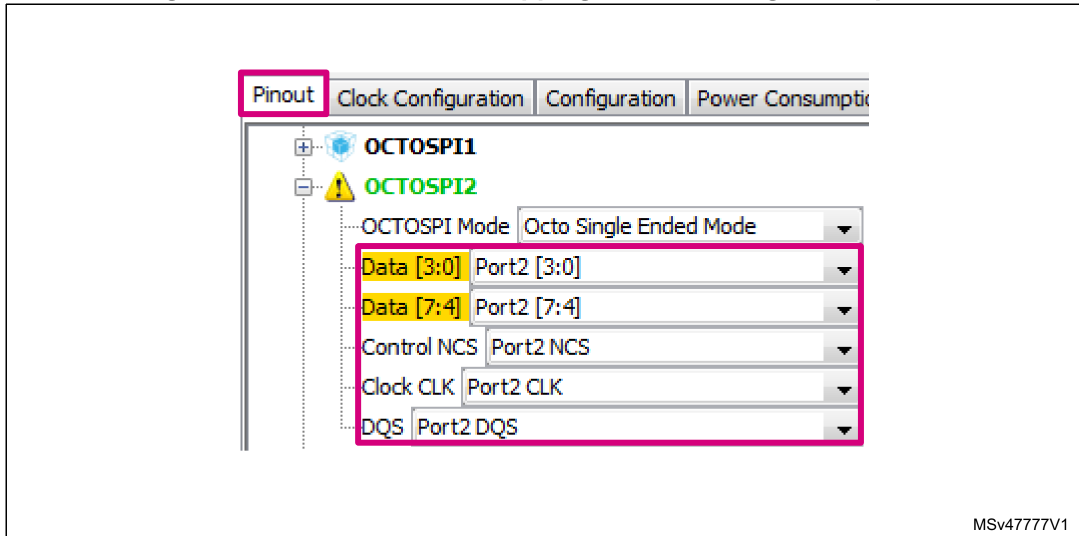
B. STM32CubeMX: OCTOSPI2 GPIOs configuration

To configure OCTOSPI2 GPIOs, follow the following steps:

1. Select the pinout tab and uncollapse the OCTOSPI2 as shown in *Figure 17*.
2. Configure the Octal I/O mode for OCTOSPI2 by selecting **“Octo Single Ended Mode”** in the listed hardware configurations shown in *Figure 17*.
3. Mapping OCTOSPI2 signals to port 2
   - Set the OCTOSPI2 LSB data signals Data[3:0] to port 2 [3:0]
   - Set the OCTOSPI2 MSB data signals Data[7:4] to port 2 [7:4]
   - Set the OCTOSPI2 chip select to port 2 NCS
   - Set the OCTOSPI2 clock to port 2 CLK
   - Set the OCTOSPI2 data strobe to port 2 DQS

Figure 17 shows how to map OCTOSPI2 signals to port 2.

Figure 17. STM32CubeMX: mapping OCTOSPI2 signals to port 2



1. Pink color highlights the key items in the figure.

Note: The DQS pin (RWDS) must be enabled since it is mandatory for HyperBus™ mode.

4. OCTOSPI2 GPIOs correct configuration

To configure OCTOSPI2 GPIOs, follow the same steps as for OCTOSPI1. *Figure 18* shows the OCTOSPI2 GPIOs properly configured to port 2.

Figure 18. STM32CubeMX: OCTOSPI2 GPIOs correct configuration

1. Pink color highlights the key items in the figure.

5. Configuring OCTOSPI2 GPIOs to very high speed

To configure OCTOSPI2 GPIOs to very high speed, follow the same steps as for OCTOSPI1.

II. STM32CubeMX: Enabling interrupts

As previously described in *Section 3.1.2*, each OctoSPI peripheral has its dedicated global interrupt connected to the NVIC, so each peripheral interrupt should be enabled separately.

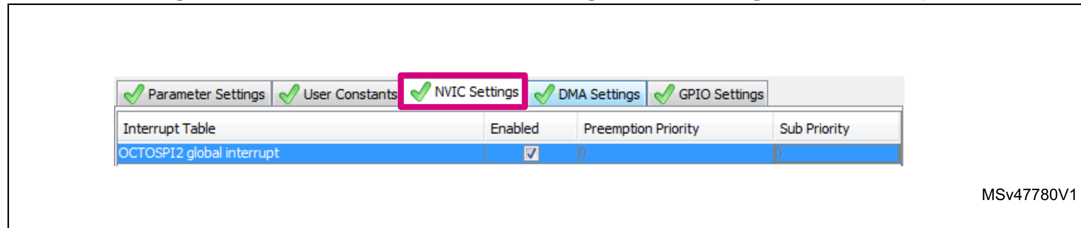
In the OCTOSPI1 configuration window (see *Figure 19*) select the NVIC settings tab, check the OCTOSPI1 global interrupts then click on the OK button.

Figure 19. STM32CubeMX: enabling OCTOSPI1 global interrupt

1. Pink color highlights the key items in the figure.

In the OCTOSPI2 configuration window (see *Figure 20*) select the NVIC settings tab, check the OCTOSPI2 global interrupts then click on the OK button.

Figure 20. STM32CubeMX: enabling OCTOSPI2 global interrupt



1. Pink color highlights the key items in the figure.

III. STM32CubeMX: clocks configuration

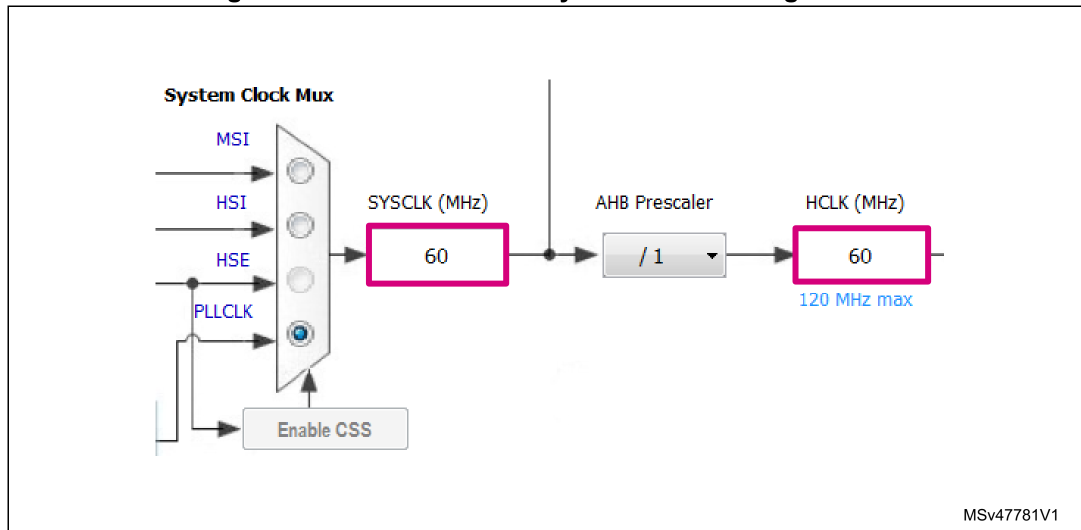
In this example the system clock is configured as shown below:

- Main PLL is used as system source clock.
- SYSCLK and HCLK set to 60 MHz, so Cortex®-M4 and AHB are operating @ 60 MHz.

  As previously described in *Section 3.1.2: Interrupts and clocks configuration*, both OctoSPI peripherals have the same clock source but each one has its dedicated prescaler allowing to connect two memories running at different speeds.

  In this example the SYSCLK is used as clock source for both OCTOSPI1 and OCTOSPI2 peripherals.
- System clock configuration:
  - Select the clock configuration tab.
  - In the clock configuration tab, set the PLLs and the prescalers to get the system clock @ 60 MHz as shown in *Figure 21*.

Figure 21. STM32CubeMX: system clock configuration



1. Pink color highlights the key items in the figure.

- OctoSPI clock source configuration:
  – In the clock configuration tab, select the SYSCLK clock source (see *Figure 22*).

Figure 22. STM32CubeMX: OCTOSPI1 and OCTOSPI2 clock source configuration

1. Pink color highlights the key items in the figure.

With this configuration, both OctoSPI peripherals are clocked by SYSCLK@60 MHz. Then for each OctoSPI peripheral a prescaler is configured to get the 60 MHz and 20 MHz targeted speed (see *Section 4.2.4: Regular-command mode* and *Section 4.2.5: HyperBus™ mode*).

4.2.4 Regular-command mode

Once that all of the OCTOSPI1 GPIOs and the clock configuration have been done, the user should configure the OCTOSPI1 peripheral to the regular-command mode.

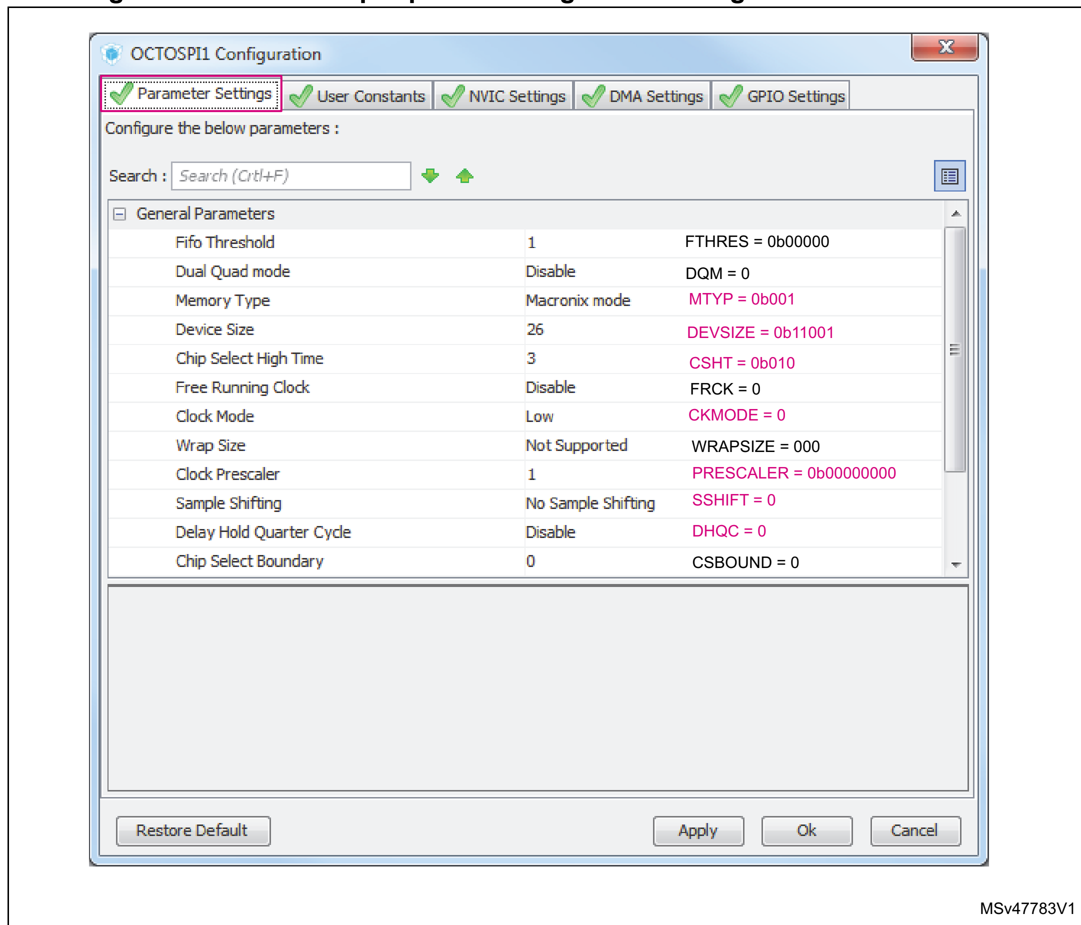
STM32CubeMX: OCTOSPI1 peripheral configuration in regular-command mode

Referring to the Macronix MX25LM51245GXDI00 datasheet, the OCTOSPI1 parameters should be configured as following:

- Memory type set to Macronix mode.
- Device size set to 26: memory size is 64 Mbytes = $2^{[DEVSIZE+1]}$ = $2^{[25+1]}$.
- Chip select high time (CSHT) set to 3: 3 OctoSPI clock cycles = 50 ns since in the datasheet the minimum CS# Deselect time from write/erase/program to read status register is 40 ns.
- Clock mode set to low (Mode 0).
- Clock prescaler set to 1: 60 MHz/1 = 60 MHz.
- Sample shifting (SSHIFT) is disabled.
- Delay hold quarter cycle (DHQC) must be disabled in SDR mode.

In the OCTOSPI1 configuration window, select the **"Parameter Settings"** tab and configure the parameters as shown in *Figure 23*. Then click on “Apply” and “OK” button.

Figure 23. OCTOSPI1 peripheral configuration in regular-command mode



1. Pink color highlights the key items in the figure.

STM32CubeMX: project generation

Once that all of the GPIOs, the clock and the OCTOSPI1 peripheral configurations have been done, the user should generate the project with the desired toolchain (SW4STM32, EWARM, MDK-ARM….).

Indirect mode and memory-mapped mode configuration

At this stage the project should be already generated with GPIOs and OCTOSPI1 peripheral properly configured following steps in *Section 4.2.3: OctoSPI GPIOs and clocks configuration* and*Section 4.2.4: Regular-command mode*. In order to configure the OCTOSPI1 peripheral in indirect/memory-mapped mode and to configure the external memory allowing communication in SDR Octal I/O mode (without DQS), some functions have to be added to the project.

- 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 allows to avoid losing the user code in case of project regeneration.

1. Insert variables declarations in the adequate space indicated in green bold below.

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------
---*/
uint8_t aTxBuffer[]=" Programming:indirect mode -Reading:mem-mapped mode ";
__IO uint8_t *nor_memaddr = (__IO uint8_t *)(OCTOSPI1_BASE);
__IO uint8_t aRxBuffer[BUFFERSIZE] ="";
/* USER CODE END PV */
```

2. Insert the functions prototypes in the adequate space indicated in green bold below.

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

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

```
/* USER CODE BEGIN 1 */
uint16_t index;
/* USER CODE END 1 */
/* USER CODE BEGIN 2 */
/*----------------------------------------------------------------------*/
/*-------------- MX25LM51245G memory configuration --------------*/
/* Configure MX25LM51245G memory to SDR Octal I/O mode */
OctalSDR_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 */
OctalSDR_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 */
```

4. Insert the functions definitions, called in the main(), in the adequate space indicated 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(&hospi1, &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(&hospi1, &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(&hospi1, &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_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 Octal mode */
  if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
  {
    Error_Handler();
  }
}
/* This function Configures Software polling to wait until WEL=1 */
void OctalPollingWEL(void)
{ uint8_t tmp;
  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_DISABLE;
  sCommand.Address            = 0x0;
  sCommand.AddressMode        = HAL_OSPI_ADDRESS_8_LINES;
  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_8_LINES;
  sCommand.DataDtrMode        = HAL_OSPI_DATA_DTR_DISABLE;
  sCommand.NbData             = 1;
  sCommand.DummyCycles        = DUMMY_CLOCK_CYCLES_READ_REG;
  sCommand.DQSMode            = HAL_OSPI_DQS_DISABLE;
  sCommand.SIOOMode           = HAL_OSPI_SIOO_INST_EVERY_CMD;
  /* Start Octal Software Polling to wait until WEL=1 */
  do
  {
    if (HAL_OSPI_Command(&hospi1, &sCommand,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
    {
      Error_Handler();
```

```
    }
    /* Read memory's Status register */
    if (HAL_OSPI_Receive(&hospi1, &tmp, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
    {
      Error_Handler();
    }
  } while((tmp & WRITE_ENABLE_MASK_VALUE) != WRITE_ENABLE_MATCH_VALUE);
}

/* 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(&hospi1, &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(&hospi1, &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)
{ uint8_t tmp;
  OSPI_RegularCmdTypeDef  sCommand;
  /* Initialize Indirect read mode for Software Polling 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_DISABLE;
  sCommand.Address             = 0x0;
  sCommand.AddressMode         = HAL_OSPI_ADDRESS_8_LINES;
  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_8_LINES;
  sCommand.DataDtrMode         = HAL_OSPI_DATA_DTR_DISABLE;
  sCommand.NbData              = 1;
  sCommand.DummyCycles         = DUMMY_CLOCK_CYCLES_READ_REG;
  sCommand.DQSMode             = HAL_OSPI_DQS_DISABLE;
  sCommand.SIOOMode            = HAL_OSPI_SIOO_INST_EVERY_CMD;
  /* Start Octal Software Polling to wait until WIP=0 */
  do
  {
    if (HAL_OSPI_Command(&hospi1, &sCommand,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
    {
      Error_Handler();
    }
    /* Read memory's Status register */
    if (HAL_OSPI_Receive(&hospi1, &tmp, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
    {
      Error_Handler();
    }
  } while((tmp & MEMORY_READY_MASK_VALUE) != MEMORY_READY_MATCH_VALUE);
}

/*** This function configures the MX25LM51245G memory ***/
void OctalSDR_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(&hospi1, &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(&hospi1, &tmp, HAL_OSPI_TIMEOUT_DEFAULT_VALUE) !=
HAL_OK)
  {
    Error_Handler();
  }
  /* Enable writing to memory in order to set Octal 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(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
  {
    Error_Handler();
  }
  /* Write Configuration register 2 with with Octal mode */
```

```
  tmp = CR2_STR_OPI_ENABLE;
  if (HAL_OSPI_Transmit(&hospi1, &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_DISABLE;
  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;
  sCommand.AddressMode         = HAL_OSPI_ADDRESS_8_LINES;
  sCommand.AddressSize         = HAL_OSPI_ADDRESS_32_BITS;
  sCommand.Address             = 0;
  sCommand.DataMode            = HAL_OSPI_DATA_NONE;
  sCommand.DummyCycles         = 0;
   /* Send Octal Sector erase cmd */
  if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
  {
  Error_Handler();
  }
}
/* This function writes the memory */
void OctalSDR_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_DISABLE;
  sCommand.AddressDtrMode          = HAL_OSPI_ADDRESS_DTR_DISABLE;
  sCommand.DataDtrMode             = HAL_OSPI_DATA_DTR_DISABLE;
  sCommand.DQSMode                 = HAL_OSPI_DQS_DISABLE;
  if (HAL_OSPI_Command(&hospi1, &sCommand, HAL_OSPI_TIMEOUT_DEFAULT_VALUE)
!= HAL_OK)
  {
    Error_Handler();
  }
  /* Memory Page programming */
  if (HAL_OSPI_Transmit(&hospi1, 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_READ_CMD;
  sCommand.InstructionDtrMode      = HAL_OSPI_INSTRUCTION_DTR_DISABLE;
  sCommand.AddressDtrMode          = HAL_OSPI_ADDRESS_DTR_DISABLE;
  sCommand.DataDtrMode             = HAL_OSPI_DATA_DTR_DISABLE;
  sCommand.DQSMode                 = HAL_OSPI_DQS_DISABLE;
  if (HAL_OSPI_Command(&hospi1, &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(&hospi1, &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(&hospi1, &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 as indicated in green bold below.

```
/* USER CODE BEGIN Private defines */
/* MX25LM512ABA1G12 Macronix memory */
/* Flash commands */
#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 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 */
```

Building and running the project

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

4.2.5 HyperBus™ mode

Once all of the OCTOSPI2 GPIOs and the clock configuration have been done, the user should configure the OCTOSPI2 peripheral to the HyperBus™ mode.

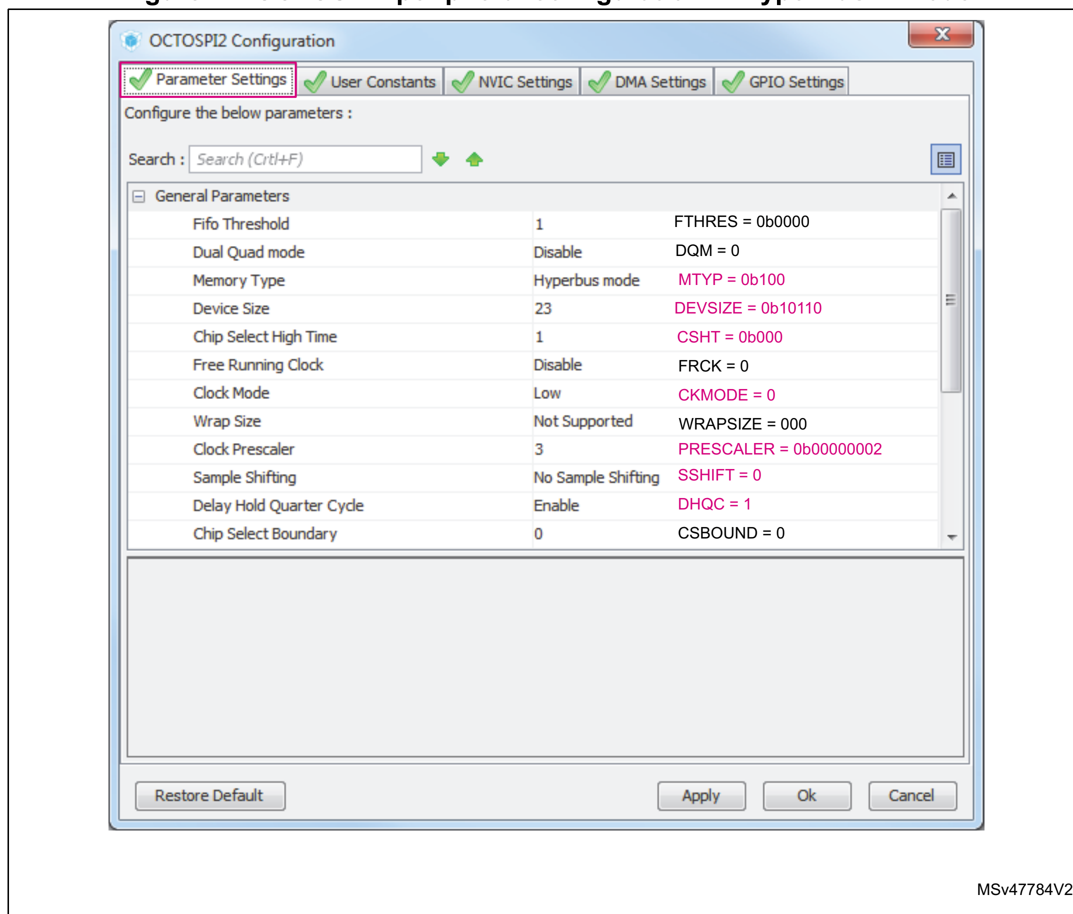
STM32CubeMX: OCTOSPI2 peripheral configuration in HyperBus™ mode

Referring to the IS66WVH8M8BLL-100BLI datasheet, the OCTOSPI2 parameters should be configured as following:

- Memory type: HyperBus™ mode.
- Device size set to 23: the memory size is 8 Mbytes = $2^{[DEVSIZE+1]} = 2^{[22+1]}$.
- Clock prescaler set to 3: 60 MHz/3 = 20 MHz.
- Chip select high time (CSHT) set to 1 OctoSPI clock cycle (50 ns), the minimum based on the datasheet is 10 ns when operating @ 3.0 V.
- Clock mode set to low (mode 0).
- Sample shifting (SSHIFT) must be disabled when DTR (DDR is enabled in OCTOSPI_CCR register) is enabled.
- Delay hold quarter clock (DHQC) enabled.

In the OCTOSPI2 configuration window, select the **“Parameter Settings”** tab then configure the parameters as shown in *Figure 24*.

Figure 24. OCTOSPI2 peripheral configuration in HyperBus™ mode



1. Pink color highlights the key items in the figure.

STM32CubeMX: project generation

Once that all of the GPIOs, the clock and the OCTOSPI2 configuration have been done, the user should generate the project with the desired toolchain (SW4STM32, EWARM, MDK-ARM….).

Memory-mapped mode configuration

At this stage the project should be already generated with the GPIOs and the OCTOSPI2 peripheral properly configured following steps in *Section 4.2.3: OctoSPI GPIOs and clocks configuration* and *Section 4.2.5: HyperBus™ mode*.

Some code lines, allowing the following configurations, have to be added to the project:

– The default memory configuration is used. No need to configure the HyperRAM™ memory.

– OCTOSPI2 peripheral timings and latency mode configuration according to the HyperRAM™ datasheet.

– OCTOSPI2 peripheral configuration to memory-mapped mode.

- Adding code to the main.c file

1. Insert variables declarations in the adequate space indicated in green bold below.

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------
---*/
uint8_t aTxBuffer[]="Writing/Reading to/from HyperRAM in mem-mapped mode";
__IO uint8_t *ram_memaddr;
__IO uint8_t aRxBuffer[BUFFERSIZE] ="";
/* USER CODE END PV */
```

2. Insert the code lines in the main() function, in the adequate space indicated in green bold below.

```
/* USER CODE BEGIN 1 */
OSPI_HyperbusCfgTypeDef sHyperbusCfg;
OSPI_HyperbusCmdTypeDef sCommand;
OSPI_MemoryMappedTypeDef sMemMappedCfg;
uint16_t index;
/* USER CODE END 1 */
/* USER CODE BEGIN 2 */
/*--- Setting the OCTOSPI2 peripheral timings and latency mode ----*/
/* TRWR Latency=3 (TRWR=50ns);TACC Latency=6;Latency on write
accesses=0(Enabled);Latency mode=1(Fixed 2 times initial Latency)*/
sHyperbusCfg.RWRecoveryTime   = 3;
  sHyperbusCfg.AccessTime           = 6;
  sHyperbusCfg.WriteZeroLatency   = HAL_OSPI_LATENCY_ON_WRITE;
  sHyperbusCfg.LatencyMode          = HAL_OSPI_FIXED_LATENCY;
  if (HAL_OSPI_HyperbusCfg(&hospi2, &sHyperbusCfg,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
  {
    Error_Handler();
  }
  /*--------- Setting OCTOSPI2 to memory-mapped mode ----------*/
  /* Setting OCTOSPI2 to HyperBus memory space mode: MTYP=0b100 */
  sCommand.AddressSpace = HAL_OSPI_MEMORY_ADDRESS_SPACE;
  sCommand.AddressSize    = HAL_OSPI_ADDRESS_32_BITS;
  sCommand.DQSMode        = HAL_OSPI_DQS_ENABLE;
  sCommand.NbData            = 1;
  if (HAL_OSPI_HyperbusCmd(&hospi2, &sCommand,
HAL_OSPI_TIMEOUT_DEFAULT_VALUE) != HAL_OK)
  {
    Error_Handler();
  }
  /* Disabling the Timeout Counter */
  sMemMappedCfg.TimeOutActivation = HAL_OSPI_TIMEOUT_COUNTER_DISABLE;
  if (HAL_OSPI_MemoryMapped(&hospi2, &sMemMappedCfg) != HAL_OK)
  {
```

```
    Error_Handler();
  }
/*----------------------------------------------------------*/

/*-------------- Writing to the HyperRAM memory --------------*/
ram_memaddr = (__IO uint8_t *)(OCTOSPI2_BASE);
for (index = 0; index < BUFFERSIZE; index++)
{
*ram_memaddr = aTxBuffer[index];
 ram_memaddr++;
}
/* Insert a delay between writing and reading */
HAL_Delay(1);
/*---------------------------------------------------------*/

/*------------------ Reading from the HyperRAM memory -------------*/
ram_memaddr = (__IO uint8_t *)(OCTOSPI2_BASE);
for(index = 0; index < BUFFERSIZE; index++)
{
/* Reading back the written aTxBuffer in memory-mapped mode */
aRxBuffer[index] = *ram_memaddr;
if(aRxBuffer[index] != aTxBuffer[index])
{
/* Can add code to toggle a LED when data doesn't match */
}
*ram_memaddr++;
}
/*---------------------------------------------------------*/
/* USER CODE END 2 */
```

- Adding defines to the main.h file
  Update the main.h file by inserting the ***defines*** in the adequate space as indicated in green bold below.

```
/* USER CODE BEGIN Private defines */
#define COUNTOF(__BUFFER__) (sizeof(__BUFFER__)/sizeof(*(__BUFFER__)))
/* Size of buffers */
#define BUFFERSIZE                  (COUNTOF(aTxBuffer) - 1)
/* USER CODE END Private defines */
```

Building and running the project

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

5 Performance and power

This section provides information in how to get the best performances and how to decrease the application’s power consumption.

5.1 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 at its maximum speed
- Use Octal I/O DDR mode for regular command mode
- Reduce command overhead

  Each new read operation needs a command/address to be sent plus a latency period which leads to command overhead. In order to reduce command overhead and boost the read performance, the user must focus on the three points below:

  – 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 clock speed.

  – Consider timeout counter
    The user should consider that enabling timeout counter in memory-mapped mode may increase command overhead and then decrease the read performance. When timeout occurs, the OctoSPI 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 should 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 register or even disable it.

5.1.1 Read performance

Some reading performance tests have been performed on the STM32L4R9I-EVAL board. The tests consist on reading, using CPU, 10-Kbytes data from the MX25LM51245GXDI00 external Octal SPI memory.

The OctoSPI is configured in regular-command memory-mapped mode with all phases (instruction, address, dummy and data) set in Octal mode.

The test code is executed from the internal Flash memory in single bank mode with ART and PREFETCH enabled.

The used toolchain was IAR Embedded workbench V8.20.1.14188 C Compiler: IAR C/C++ Compiler for Arm Version 8.20.1.14183

Table 2 shows some read performance results when reading from the external Macronix memory where the OctoSPI is set in regular-command memory-mapped mode.

Table 2. OctoSPI memory-mapped mode Reading performance

System frequency HCLK (MHz)	OctoSPI frequency (MHz)	Reading throughput (Mbyte/s)	
		Octal SDR	Octal DDR
120	60	60	106
80	40	40	71
60	60	60	64
48	48	48	51

5.2 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's power-consumption, the user usually puts the STM32 in low-power mode. To reduce even more the current consumption, the connected memory can also be put in low-power mode.

5.2.1 STM32 low-power modes

The STM32 low-power states is an important requirement that must be considered as it has a direct effect on the overall application power consumption and on the OctoSPI interface state. For instance, the OctoSPI has to be reconfigured after wakeup from standby or shutdown mode.

Table 3 summarizes OctoSPI peripheral state for STM32L4Rxxx and STM32L4Sxxx devices in different power modes.

Table 3. OctoSPI peripheral state in different power modes on STM32L4Rxxx and STM32L4Sxxx

<table>
<tr><th>Mode</th><th>Description</th></tr>
<tr><td>Run</td><td rowspan="2">Active</td></tr>
<tr><td>Low-power run</td></tr>
<tr><td>Sleep</td><td>Active. Peripheral interrupts cause the device to exit sleep mode.</td></tr>
<tr><td>Low-power sleep</td><td>Active. Peripheral interrupts cause the device to exit low-power sleep mode.</td></tr>
<tr><td>Stop0</td><td rowspan="3">Frozen. Peripheral registers content is kept.</td></tr>
<tr><td>Stop1</td></tr>
<tr><td>Stop2</td></tr>
<tr><td>Standby</td><td rowspan="2">Powered-down. The peripheral must be reinitialized after exiting standby or shutdown mode.</td></tr>
<tr><td>Shutdown</td></tr>
</table>

5.2.2 Decreasing Octal-SPI memory's 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 powering-up sequence is the standby low-power mode. In standby mode, there is no ongoing operation; the nCS is high and current consumption is relatively less than operating mode.

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

The application can put the memory device in DPD mode before entering STM32 in low-power mode when the memory is not used. This action permits a reduction of the overall application's power-consumption.

- **Entering and exiting DPD mode**

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

To exit DPD mode, some memory devices are requiring to issue a “release from deep power-down” (RDP) command. For some other memory devices a hardware reset leads to exit DPD mode.

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

6 Supported devices

The OctoSPI interface can operate in two different low-level protocols: the regular-command mode and the HyperBus™ mode.

- Thanks to the regular-command mode frame format flexibility, any SPI, Quad-SPI or Octal-SPI memory can be connected to an STM32 device.
  There are several suppliers of Octal-SPI compatible memories, such as Macronix, Adesto, MICRON, Cypress (Spansion) and others.
- Thanks to the HyperBus™ low-level protocol support, several HyperRAM™ and HyperFlash™ memories are supported by STM32 devices. Some memory manufactures like Cypress and ISSI provide HyperRAM™ and HyperFlash™ memories.

As already described in *Section 4.2: OctoSPI STM32CubeMX examples*, the MACRONIX MX25LM51245GXDI0A Octal-SPI Flash memory and the ISSI IS66WVH8M8BLL-100BLI HyperRAM™ are embedded on the STM32L4R9I-EVAL board, while a MACRONIX MX25LM51245GXDI00 is embedded on the STM32L4R9I-DISCO board.

7 Conclusion

STM32 MCUs provide a very flexible OctoSPI 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 STM32L4+ Series excellent OctoSPI interface performance and flexibility allowing lower development costs and faster time to market.

8 Revision history

Table 4. Document revision history

Date	Revision	Changes
20-Oct-2017	1	Initial release.
27-Apr-2018	2	Updated – *Section 1: Overview of STM32L4+ Series OctoSPI interface* – *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 34* – *Section : STM32CubeMX: OCTOSPI2 peripheral configuration in HyperBus™ mode on page 45* – *Section : STM32CubeMX: project generation on page 46* – *Figure 9: Examples configuration: OCTOSPI1 set to regular-command mode and OCTOSPI2 set to HyperBus™* – *Figure 24: OCTOSPI2 peripheral configuration in HyperBus™ mode* – *Table 1: 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*

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2018 STMicroelectronics – All rights reserved