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 two MCU products in the market with the new integrated high-throughput Octo-SPI interface: the STM32L4+ Series and the STM32L5 Series devices.

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

Thanks to the Octo-SPI 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 or SDMMC) 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 peripheral in STM32 MCUs and explains how to configure it in order to write and read external Octo-SPI and HyperBus™ memories. This document describes some typical use cases to use the Octo-SPI interface and provides some practical examples on how to configure the OCTOSPI peripheral depending on the type of the targeted memory.

Table 1. Applicable products

<table>
<thead>
<tr>
<th>Type</th>
<th>Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontrollers</td>
<td>STM32L4+ Series, STM32L5 Series</td>
</tr>
</tbody>
</table>

Related documents

Available from STMicroelectronics web site [www.st.com](http://www.st.com):

- reference manuals for STM32L4+ and STM32 L5 Series (RM0432 and RM0438)
- datasheets for STM32L4+ and STM32 L5 Series
- application note *Quad-SPI interface on STM32 microcontrollers* (AN4760)
Contents

1 Overview of the OCTOSPI in STM32 MCUs .............................. 6
  1.1 OCTOSPI main features .............................................. 6
  1.2 OCTOSPI in a smart architecture ................................ 6
    1.2.1 STM32L4+ Series system architecture ....................... 7
    1.2.2 STM32L5 Series system architecture ....................... 9

2 Octo-SPI interface description ........................................... 10
  2.1 OCTOSPI hardware interface ...................................... 10
    2.1.1 OCTOSPI pins and signal interface ......................... 10
    2.1.2 OCTOSPI IO manager in STM32L4+ Series .................. 10
  2.2 Two low-level protocols ......................................... 11
    2.2.1 Regular-command protocol .................................. 11
    2.2.2 HyperBus protocol ........................................... 12
  2.3 Three operating modes ............................................ 13
    2.3.1 Indirect mode ................................................. 13
    2.3.2 Status-flag polling mode ................................... 13
    2.3.3 Memory-mapped mode ......................................... 14

3 OCTOSPI configuration .................................................... 16
  3.1 OCTOSPI common configuration .................................... 16
    3.1.1 Connecting two octal memories to one Octo-SPI interface 16
    3.1.2 GPIOs and OCTOSPI I/Os configuration ..................... 17
    3.1.3 Interrupts and clocks configuration ....................... 18
  3.2 OCTOSPI configuration for regular-command protocol ........... 19
  3.3 OCTOSPI configuration for HyperBus protocol ................... 19
  3.4 Memory configuration ............................................. 20
    3.4.1 Octo-SPI memory device configuration ..................... 20
    3.4.2 HyperBus memory device configuration ..................... 20

4 OCTOSPI application examples ........................................... 21
  4.1 Implementation examples .......................................... 21
    4.1.1 Using OCTOSPI in a graphical application ................. 21
    4.1.2 Executing from external memory: extend internal memory size 22
4.2 OCTOSPI configuration with STM32CubeMX .................................. 23
  4.2.1 Hardware description ......................................................... 23
  4.2.2 Use case description ....................................................... 25
  4.2.3 OCTOSPI GPIOs and clocks configuration ............................ 26
  4.2.4 Regular-command mode ..................................................... 33
  4.2.5 HyperBus protocol ......................................................... 45

5 Performance and power ........................................................... 50
  5.1 How to get the best read performance ................................... 50
  5.2 Decreasing power consumption ............................................. 50
    5.2.1 STM32 low-power modes ................................................. 51
    5.2.2 Decreasing Octo-SPI memory power consumption ................. 51

6 Supported devices ................................................................. 53

7 Conclusion ............................................................................. 54

8 Revision history ................................................................. 55
List of tables

Table 1. Applicable products ................................................................. 1
Table 2. OCTOSPI main features ......................................................... 6
Table 3. OCTOSPI states in different power modes on STM32L4+ and STM32L5 Series ...... 51
Table 4. Document revision history .................................................... 55
List of figures

Figure 1. STM32L4+ Series system architecture .................................................. 8
Figure 2. STM32L5 Series system architecture .................................................. 9
Figure 3. Example of connecting an Octo-SPI Flash memory and
an HyperRAM memory to an STM32 device .................................................. 11
Figure 4. Regular-command mode: octal DTR read operation example
in Macronix mode ............................................................................................... 12
Figure 5. HyperBus protocol: example of reading operation from HyperRAM ......... 13
Figure 6. Connecting two memories to an Octo-SPI interface ................................. 17
Figure 7. OCTOSPI1 and OCTOSPI2 clock scheme ............................................ 18
Figure 8. OCTOSPI graphic application use case .................................................. 22
Figure 9. Executing code from memory connected to OCTOSPI2 ......................... 23
Figure 10. Octo-SPI Flash and HyperRAM memories connection to the STM32L4R9AI
on the STM32L4R9I-EVAL board ........................................................................ 24
Figure 11. Examples configuration: OCTOSPI1 set to
regular-command protocol and OCTOSPI2 set to HyperBus protocol ................ 25
Figure 12. STM32CubeMX: setting Octal SPI mode for OCTOSPI1 ...................... 26
Figure 13. STM32CubeMX: mapping OCTOSPI1 signals to Port1 ......................... 27
Figure 14. STM32CubeMX: setting PB0 pin to OCTOSPIM_P1_IO1 alternate function .. 27
Figure 15. STM32CubeMX - OCTOSPI1 GPIOs correct configuration .................... 28
Figure 16. STM32CubeMX - configuration tab ..................................................... 28
Figure 17. OCTOSPI1 button in the configuration tab ........................................... 29
Figure 18. STM32CubeMX - setting GPIOs to very high speed ............................ 29
Figure 19. STM32CubeMX: setting OCTOSPI2 signals to Port 2 ......................... 30
Figure 20. STM32CubeMX: OCTOSPI2 GPIOs correct configuration .................... 31
Figure 21. STM32CubeMX: enabling OCTOSPI1 global interrupt ......................... 31
Figure 22. STM32CubeMX: enabling OCTOSPI2 global interrupt ......................... 32
Figure 23. STM32CubeMX: system clock configuration ......................................... 32
Figure 24. STM32CubeMX: OCTOSPI1 and OCTOSPI2 clock source configuration .. 33
Figure 25. OCTOSPI1 peripheral configuration in regular-command mode ............ 34
Figure 26. OCTOSPI2 peripheral configuration in HyperBus protocol ................... 46
1 Overview of the OCTOSPI in STM32 MCUs

This section provides an overview of the OCTOSPI peripheral availability across the STM32L4+ and STM32L5 Series, Arm® Cortex® core-based devices.

1.1 OCTOSPI main features

The table below summarizes the OCTOSPI main features.

<table>
<thead>
<tr>
<th>STM32 Series</th>
<th>Max AHB frequency (MHz)</th>
<th>Maximum OCTOSPI speed (MHz)(1)</th>
<th>Number of instances</th>
<th>Differential clock mode (1V8)</th>
<th>OCTOSPI IO manager</th>
<th>Maximum addressable space(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Regular-command protocol</td>
<td>HyperBus protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SDR</td>
<td>DTR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM32L4+</td>
<td>120</td>
<td>86</td>
<td>60</td>
<td>32</td>
<td>2</td>
<td>Available</td>
</tr>
<tr>
<td>STM32L5</td>
<td>110</td>
<td>90</td>
<td>60</td>
<td>76</td>
<td>1</td>
<td>Available</td>
</tr>
</tbody>
</table>

1. For more details on the OCTOSPI maximum speed, refer to the relevant product datasheet.
2. 32-bit address mode must 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.

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

- All masters can access autonomously to the 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.
- DMA can do transfers from the OCTOSPI to other internal or external memories.
- Graphical DMA2D can directly build framebuffer using graphic primitives from the connected Octo-SPI Flash or HyperFlash™ memory.
- DMA2D can directly build framebuffer in Octo-SPI SRAM or HyperRAM™.
- GFXMMU as a master can autonomously access the OCTOSPI.

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
• LTDC can fetch framebuffer directly from the memory that is connected to the OCTOSPI.
• SDMMC master interface can transfer data between the OCTOSPI and SD/MMC/SDIO cards without any CPU intervention.

1.2.1 STM32L4+ Series system architecture

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

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

Figure 1. STM32L4+ Series system architecture
1.2.2 STM32L5 Series system architecture

The STM32L5 Series system architecture consists mainly of a 32-bit multilayer AHB bus matrix that interconnects six masters and seven slaves.

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

- One OCTOSPI slave (OCTOSPI1) mapped on a dedicated AHB layer and accessible independently by all the masters connected to 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, the OCTOSPI1 addressable space is from 0x9000 0000 to 0x9FFF FFFF.
- The external memory connected to the OCTOSPI1 can be accessed (for code execution or data) by the Cortex-M33 either through S-Bus or through C-bus when physical remap is enabled.
- The CPU can benefit from the 8-Kbyte ICACHE for code execution when accessing the OCTOSPI by remap. Thanks to the 8-Kbyte ICACHE, the CoreMark® execution from the external memory can reach a highly close score to the internal Flash memory.

The figure below shows the OCTOSPI1 in the STM32L5 Series system architecture.

Figure 2. STM32L5 Series system architecture
2 Octo-SPI 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 (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, that enables the support of multiple hardware configurations. It supports the single-SPI (traditional SPI), dual-SPI, quad-SPI, dual quad-SPI and Octo-SPI. It also supports HyperBus protocol with single ended clock (3 V signals) or differential clock (1V8 signals). The flexibility of the OCTOSPI hardware interface permits the connection of most serial memories available in the market.

2.1.1 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
- OCTOSPI_DQS line for data strobe
- OCTOSPI_IO[0...7] eight lines for data

Note: The HyperBus differential clock mode (1V8) is supported only with STM32L5 Series devices.

*Figure 3* shows Octo-SPI interface signals.

2.1.2 OCTOSPI IO manager in STM32L4+ Series

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 the OCTOSPI1 or the OCTOSPI2.

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

For instance when two external memories are used, an HyperRAM can be connected to Port1 and an Octo-SPI Flash memory can be connected to Port2 as shown in *Figure 3*. 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 3* shows an Octo-SPI Flash and an HyperRAM memories connected to the STM32 MCU using the Octo-SPI interface. Thanks to the OCTOSPI IO manager, the HyperRAM memory can be linked to the OCTOSPI1 and the Flash memory can be linked to the OCTOSPI2, and vice versa.
2.2 Two low-level protocols

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

2.2.1 Regular-command protocol

The regular-command protocol 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 a single-SPI, a dual-SPI, a quad-SPI, a dual quad-SPI or an Octo-SPI memory.

Flexible-frame format and hardware interface

The Octo-SPI 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 or octo-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 or octo-SPI 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 or octo-SPI mode.

- **Dummy-cycles 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, dual-SPI, quad-SPI or octo-SPI mode. 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.

**Figure 4. Regular-command mode: octal DTR read operation example in Macronix mode**

---

### 2.2.2 HyperBus protocol

The OCTOSPI supports the HyperBus protocol that 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:** 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:** 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 (tACC initial access). If RWDS was **high** during the CA period, an additional latency count is inserted (2*tACC).

The initial latency count (tACC) 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.

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

Figure 5. HyperBus protocol: 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: must be used to access to the memory register space, that 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 detailed below.

2.3.1 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 masters to access autonomously the OCTOSPI peripheral (available in memory-mapped mode)
- for all the operations to be performed through the OCTOSPI data register, using CPU or 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 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 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 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:

- read and write operations
- to use the external memory device exactly like an internal memory (so that any AHB 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 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:

- In STM32L4+ Series:
  - OCTOSPI1 addressable space: from 0x9000 0000 to 0x9FFF FFFF (256 Mbytes)
  - OCTOSPI2 addressable space: from 0x7000 0000 to 0x7FFF FFFF (256 Mbytes)
- In STM32L5 Series:
  - OCTOSPI1 addressable space: from 0x9000 0000 to 0x9FFF FFFF (256 Mbytes)

Starting memory-mapped read or write operation

A memory-mapped operation is started as soon as there is an AHB 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: When programming a Flash memory in memory-mapped mode, the application must wait until the programming operation is finished. 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.

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 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 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 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.
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 or HyperBus protocol).

- 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 or HyperBus).

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 Connecting two octal memories to one Octo-SPI interface

This section describes how to address two external memories using one Octo-SPI interface with a pseudo-static communication.

Example: connecting one HyperRAM and one HyperFlash to an STM32L5 Series MCU 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 following figure.

**Figure 6. Connecting two memories to an Octo-SPI interface**

### 3.1.2 GPIOs and OCTOSPI 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 or octo-SPI).

In Octal SPI mode, ten GPIOs are needed when only one external memory is connected. An additional GPIO for DQS is optional for regular-command mode and mandatory for HyperBus protocol. A differential clock mode (nCLK) is needed only in HyperBus mode 1V8.

In STM32L4+ Series, when two external octal memories are connected, each memory must be connected to an IO manager port, that requires up to 22 GPIOs.

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 mode.*

**GPIOs configuration using STM32CubeMX**

Thanks to the STM32CubeMX tool, the OCTOSPI peripheral and its GPIOs can be configured very simply, easily and quickly. The STM32CubeMx is used to generate a project with a preconfigured OCTOSPI. *Section 4.2.3: OCTOSPI GPIOs and clocks configuration* details how to configure the OCTOSPI GPIOs.

**OCTOSPI I/Os configuration**

By default, after reset, all OCTOSPI1 and OCTOSPI2 signals are mapped respectively to Port 1 and to Port 2.
3.1.3 Interrupts and clocks configuration

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

Enabling interrupts

In STM32L4+ Series 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 must enable the OCTOSPI1 and/or OCTOSPI2 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

In STM32L4+ Series 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.

\[ \text{OCTOSPI}_{x, \text{CLK}} = \frac{\text{F}_{\text{clock, source}}}{\text{(PRESCALER [+ 1])}} \]

For instance, when the PRESCALER[7:0] is set to 2, \( \text{OCTOSPI}_{x, \text{CLK}} = \frac{\text{F}_{\text{clock, source}}}{3} \).

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

OCTOSPI 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 7. OCTOSPI1 and OCTOSPI2 clock scheme**
3.2 OCTOSPI configuration for regular-command protocol

The regular-command protocol must be used when an external single-SPI, quad-SPI, dual quad-SPI or Octo-SPI memory is connected to the STM32.

The user must configure the following OCTOSPI parameters:

- memory type: Micron mode, Macronix mode or Macronix RAM mode
- device size: number of bytes in the device \( = 2^{[\text{DEVSIZE}+1]} \)
- 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.
- 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 clock cycle enabling more relaxed input timings.

3.3 OCTOSPI 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 OCTOSPI parameters:

- memory type: HyperBus
- device size: number of bytes in the device \( = 2^{[\text{DEVSIZE}+1]} \)
- 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)
- DTR (DDR) mode: must be enabled for HyperBus
- 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.
- SSHIFT: must be disabled since HyperBus operates in DDR mode
- read-write recovery time \( (t_{\text{RWR}}) \): used only for HyperRAM and must be configured according to the memory device
- initial latency \( (t_{\text{ACC}}) \): 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
3.4 Memory configuration

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

3.4.1 Octo-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:

1. Set the dummy cycles according to the operating speed (see relevant memory device datasheet).
2. Enable the Octal SPI mode that enables the communication in Octo-SPI mode.
3. Enable DTR mode that enables the communication in 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.

3.4.2 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 register.

When memory voltage range is 1.8 V, HyperBus requires differential clock mode 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
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 describes the following typical OCTOSPI use case examples:

- using OCTOSPI in a graphical application
- code execution from OCTOSPI memory

4.1.1 Using OCTOSPI in a graphical application

The STM32L4+ Series devices embed two independent OCTOSPI peripherals that enable the connection of two external memories.

This configuration is ideal for graphical applications where:

- An Octo-SPI Flash memory is connected to OCTOSPI1 that is used to store graphical primitives.
- An HyperRAM memory is connected to OCTOSPI2 that is used to build framebuffer.
- OCTOSPI1 must be set to regular-command protocol in order to communicate with the Octo-SPI Flash memory.
- OCTOSPI2 must be set to HyperBus protocol in order to communicate with the HyperRAM.
- Both OCTOSPI1 and OCTOSPI2 must be configured in memory-mapped mode.
- Any AHB master (such as CPU, LTDC, DMA2D or GFXMMU) can autonomously access to both memories, exactly like an internal memory.
4.1.2 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 9. Executing code from memory connected to OCTOSPI2**

### 4.2 OCTOSPI configuration with STM32CubeMX

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 Octo-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 Discovery kit STM32L4R9I-DISCO that embeds a Macronix MX25LM51245GXI00 Octo-SPI Flash memory (same memory as STM32L4R9I-EVAL board).

For more details on STM32L4R9I-EVAL and STM32L4R9I-DISCO, refer to the user manuals Evaluation board with STM32L4R9AI MCU (UM2248) and Discovery kit with STM32L4R9AI MCU (UM2271) respectively.

#### 4.2.1 Hardware description

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

- Macronix MX25LM51245GXI0A Octo-SPI Flash memory connected to Port1
- ISSI IS66WVH8M8BLL-100BLL HyperRAM memory connected to Port2
As shown in the figure below, 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).

The figure below shows Macronix MX25LM51245GXD10A and ISSI IS66WVH8M8BL100BLI memories, connected to the STM32L4R9AI MCU.

Figure 10. Octo-SPI Flash and HyperRAM memories connection to the STM32L4R9AI on the STM32L4R9I-EVAL board
4.2.2 Use case description

The adopted configuration for each example is the following:

- Regular-command protocol example:
  - OCTOSPI1 signals are mapped to Port 1 (Nor Flash). So OCTOSPI1 must be set to regular-command mode.
  - SDR Octal SPI mode (without DQS) with OCTOSPI1 running at 60 MHz
  - Programming the memory in indirect mode and reading in memory-mapped mode

- HyperBus protocol example:
  - OCTOSPI2 signals are mapped to Port 2 (HyperRAM). So OCTOSPI2 must be set to HyperBus protocol.
  - DDR Octal SPI mode (with DQS) with OCTOSPI2 running at 20 MHz
  - Memory-mapped mode for reading and writing

The figure below illustrates the OCTOSPI configuration for each example.

**Figure 11. Examples configuration: OCTOSPI1 set to regular-command protocol and OCTOSPI2 set to HyperBus protocol**

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).

I. STM32CubeMX: GPIOs configuration

As shown in Figure 10, the Macronix MX25LM51245GXD100 Octo-SPI Flash memory is connected to the STM32L4R9 MCU through the OCTOSPI IO manager Port 1, while the ISSI IS66WVH8M8BLL-100BL1 HyperRAM is connected through Port 2.

Based on this hardware implementation the user must configure all the GPIOs shown in Figure 10.

A. STM32CubeMX: OCTOSPI1 GPIOs configuration

Once 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 the figure below.

2. Configure the Octal SPI mode for OCTOSPI1 by selecting “Octo Single Ended Mode” in the listed hardware configurations shown in the figure below.

Figure 12. STM32CubeMX: setting Octal SPI mode for OCTOSPI1

3. Map OCTOSPI1 signals to Port 1 with the following steps:
   - 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)

   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 MX25LM51245GXD100 datasheet).
The figure below shows how to map OCTOSPI1 signals to Port 1.

**Figure 13. 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 10*. If the configuration is not correct, the user must manually configure all the GPIOs, one by one, by clicking on each pin directly.

The figure below shows how to configure manually the PB0 pin to OCTOSPIM_P1_IO1 alternate function.

**Figure 14. 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

The figure below shows the OCTOSPI1 GPIOs properly configured to Port 1.

**Figure 15. STM32CubeMX - OCTOSPI1 GPIOs correct configuration**

Once all the OCTOSPI GPIOs are properly set, the status of the OCTOSPI1 must be OK in the OCTOSPI1 configuration button as illustrated in **Figure 17**. 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 the figure below.

**Figure 16. 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 below.

**Figure 17. OCTOSPI1 button in the configuration tab**

![OCTOSPI1 button in the configuration tab](image)

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 the figure below.

d) Click on Apply and then OK.

**Figure 18. STM32CubeMX - setting GPIOs to very high speed**

![STM32CubeMX settings GPIOs to very high speed](image)

1. Pink color highlights the key items in the figure.
B. STM32CubeMX: OCTOSPI2 GPIOs configuration

To configure OCTOSPI2 GPIOs, follow the steps below:

1. Select the pinout tab and uncollapse the OCTOSPI2 as shown in the figure below.

2. Configure the Octal SPI mode for OCTOSPI2 by selecting “Octo Single Ended Mode” in the listed hardware configurations shown in the figure below.

3. Map OCTOSPI2 signals to Port 2 with the following steps:
   - 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.

The figure below shows how to map OCTOSPI2 signals to Port 2.

![Figure 19. STM32CubeMX: mapping OCTOSPI2 signals to Port 2](image)

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

Note: The DQS pin (RWDS) must be enabled since it is mandatory for HyperBus protocol.
4. OCTOSPI2 GPIOs correct configuration:

To configure OCTOSPI2 GPIOs, follow the same steps as for OCTOSPI1. The figure below shows the OCTOSPI2 GPIOs properly configured to Port 2.

**Figure 20. STM32CubeMX: OCTOSPI2 GPIOs correct configuration**

5. Configure 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 3.1.3: Interrupts and clocks configuration*, each OCTOSPI peripheral has its dedicated global interrupt connected to the NVIC, so each peripheral interrupt must be enabled separately.

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

**Figure 21. STM32CubeMX: enabling OCTOSPI1 global interrupt**

1. Pink color highlights the key items in the figure.
In the OCTOSPI2 configuration window (see the figure below), select the NVIC settings tab, check the OCTOSPI2 global interrupts, then click on the OK button.

**Figure 22. 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 operate at 60 MHz.

As previously described in *Section 3.1.3: 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 both 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 60 MHz as shown in the figure below.

**Figure 23. 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 the figure below).

Figure 24. STM32CubeMX: OCTOSPI1 and OCTOSPI2 clock source configuration

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 protocol).

### 4.2.4 Regular-command mode

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

STM32CubeMX: OCTOSPI1 peripheral configuration in regular-command mode

Referring to the Macronix MX25LM51245GXD100 datasheet, the OCTOSPI1 parameters must be configured as follows:

- Memory type set to Macronix mode
- Device size set to 26: memory size is 64 Mbytes \(= 2^{[\text{DEVSIZE}+1]} = 2^{[25+1]} \)
- Chip-select high time (CSHT) set to 3: three 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) 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 the figure below. Then click on “Apply” and “OK” button.

Figure 25. OCTOSPI1 peripheral configuration in regular-command mode

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

STM32CubeMX: project generation

Once all of the GPIOs, the clock and the OCTOSPI1 configurations have been done, the user must generate the project with the desired toolchain (such as SW4STM32, EWARM or MDK-ARM).

Indirect mode and memory-mapped mode configuration

At this stage, the project must be already generated with GPIOs and OCTOSPI1 properly configured following the steps detailed in Section 4.2.3: OCTOSPI GPIOs and clocks configuration and Section 4.2.4: Regular-command mode.

In order to configure the OCTOSPI1 in indirect/memory-mapped mode and to configure the external memory allowing communication in SDR Octal SPI mode (without 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 losing the user code in case of project regeneration.

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

```c
/* 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 */
```

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

```c
/* 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 */
```

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

```c
/* 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 */


/* 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_DISABLE;
    sCommand.DataMode           = HAL_OSPI_DATA_DISABLE;
    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 function 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.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.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_ADDR;
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)
{
    OSPISI_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();
    }
}
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 (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 protocol

Once all of the OCTOSPI2 GPIOs and the clock configuration have been done, the user must configure the OCTOSPI2 to the HyperBus protocol.

STM32CubeMX: OCTOSPI2 peripheral configuration in HyperBus protocol

Referring to the IS66WVH8M8BLL-100BLI datasheet, the OCTOSPI2 parameters must be configured as follows:

- Memory type: HyperBus
- 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 the figure below.

Figure 26. OCTOSPI2 peripheral configuration in HyperBus protocol

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

STM32CubeMX: project generation

Once all of the GPIOs, the clock and the OCTOSPI2 configuration have been done, the user must generate the project with the desired toolchain (such as W4STM32, EWARM or MDK-ARM).

Memory-mapped mode configuration

At this stage, the project must be already generated with the GPIOs and the OCTOSPI2 properly configured, following the steps detailed in *Section 4.2.3: OCTOSPI GPIOs and clocks configuration* and *Section 4.2.5: HyperBus protocol*. 
The following configurations are expected:

- The default memory configuration is used. No need to configure the HyperRAM memory.
- OCTOSPI2 timings and latency mode configuration according to the HyperRAM datasheet.
- OCTOSPI2 peripheral configuration to memory-mapped mode

Some code lines must then 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
  
a) Insert variables declarations in the adequate space (in green bold below).

```c
/* 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 */
```

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

```c
/* 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_TIMBBOUT_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 (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 explains how to get the best performances and how to decrease the application 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 SPI DDR mode for regular command mode.
- 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 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 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 register or even disable it.

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 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.
5.2.1 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 OCTOSPI interface state. For instance, the OCTOSPI must be reconfigured after wakeup from Standby or Shutdown mode.

The table below summarizes the OCTOSPI states for STM32L4+ and STM32L5 Series, in different power modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>OCTOSPI state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>Active</td>
</tr>
<tr>
<td>Low-power run</td>
<td></td>
</tr>
<tr>
<td>Sleep</td>
<td>Active. OCTOSPI interrupts cause the device to exit Sleep mode.</td>
</tr>
<tr>
<td>Low-power sleep</td>
<td></td>
</tr>
<tr>
<td>Stop 0</td>
<td>Frozen. OCTOSPI registers content is kept.</td>
</tr>
<tr>
<td>Stop 1</td>
<td></td>
</tr>
<tr>
<td>Stop 2</td>
<td></td>
</tr>
<tr>
<td>Standby</td>
<td>Powered-down. OCTOSPI must be reinitialized after exiting Standby or Shutdown mode.</td>
</tr>
<tr>
<td>Shutdown</td>
<td></td>
</tr>
</tbody>
</table>

5.2.2 Decreasing Octo-SPI 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.
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.
6 Supported devices

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

Thanks to the regular-command mode frame format flexibility, any SPI, Quad-SPI or Octo-SPI memory can be connected to an STM32 device. There are several suppliers of Octo-SPI compatible memories (such as Macronix, Adesto, MICRON or Cypress (Spansion)).

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

As already described in Section 4.2: OCTOSPI configuration with STM32CubeMX, the Macronix MX25LM51245GXI0A Octo-SPI Flash memory and the ISSI IS66WVH8M8BLL-100BLI HyperRAM, are embedded on the STM32L4R9I-EVAL and the STM32L552E-EVAL boards, while a Macronix MX25LM51245GXI00 is embedded on the STM32L4R9i-DISCO Discovery kit.
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 the STM32L4+ and STM32L5 Series excellent OCTOSPI interface performance and flexibility. The STM32 OCTOSPI allows lower development costs and faster time to market.
### 8 Revision history

Table 4. Document revision history

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