Quad-SPI core for FPGA on SEcube™

Project Documentation

Release: October 2019

Proprietary Notice

The following document offers information, which is subject to the terms and conditions described hereafter.
While care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinions expressed are in good faith and while every care has been taken in preparing this document, some typographical errors, error or omissions may have occurred. We reserve the right to make changes to the content and information described herein or update such information at any time without notice. The opinion expressed are in good faith and while every care has been taken in preparing this document.

Authors

Vahid EFTEKHARI MOGHADAM vahid.eftekharimoghadam@studenti.polito.it
Nicoló MAUNERO *CINI Cybersecurity National Lab)* nicolo.maunero@polito.it
Paolo PRINETTO *(President, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *CINI Cybersecurity National Lab)* gianluca.roascio@polito.it
Antonio SCIALDONE antonio.scialdone@studenti.polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Contents

1. Work presentation

The idea behind this project is to enhance the communication capabilities of the **SE*cube*™** board, making it able to communicate through a new interface, not supported natively by the **SE*cube*™** Chip.
The developed IP core is synthetizable onto the **SE*cube*™** FPGA, and makes use of its availale I/O pins to communicate with other devices using Quad-SPI. The transmission rate that can be achieved with the provided design is 110 MB/s, which constitutes a great improvement (2x faster) over the classic SPI interface already available on the **SE*cube*™** .
This document is meant to present all the necessary information about the core. Starting from a brief explanation of the protocol, the architecture of the core will be presented and explained, followed by a step-by-step guide explaining how to embed it in the multi-IP environment already available for the FPGA. Finally, an example project is presented, describing how it is possible to use the core to communicate with another device through Quad-SPI.

2. Overview

Quad-SPI is a communication standard interface which extends the classical SPI, widely spread in the embedded system domain. As for Quad-SPI, it is mainly used for interfacing with embedded memories, as it gives the possibility to write/read data at a sustained rate. It allows an half-duplex communication between two devices, one *master* and one *slave*, which can exchange 4 bits per clock cycle, as the name suggests, rather than a single one as in SPI.
More specifically, the lines involved in the transmission are 6. These are:

- **SCLK**: it is the clock of the communication, always generated by the master, used for synchronization;
- **CS**: it is the chip-select signal, connected to the corresponding slave, and it is active low. In case there is only one slave, this line is not strictly required;
- **SDIO[3:0]**: the 4-bit bidirectional data bus, which carries the bits to be transferred.

Beside the physical interface, there are few required *parameters*, which are introduced here through a simple example.
At first, when there is no transmission occurring, the SCLK signal remains idle on a constant level, depending on the **clock polarity** parameter (hereinafter referred as CPOL):

- **CPOL = 0** means that the clock is low when idle;
- **CPOL = 1** means that the clock is high when idle.

Whenever the master wants to communicate with a slave device, the chip-select signal is driven low by the master, and kept low for the entire transmission window. The master then generates **n** clock cycles. Every clock cycle, 4 bits (1 nibble) are sent, either by the master or by the slave. Therefore, the other two parameters of the transmission that must be known are the number of nibbles to transmit (because the master has to generate the clock signal for a precise number of cycles), and the frequency of the clock signal. This last is needed in order to carry out the transmission correctly, because the slave device must be able to support the frequency of the clock generated by the master. During the transmission, data changes at every clock edge, and is sampled on the opposite clock edge. This last feature is defined by the last parameter, the **clock phase** (hereinafter referred as CPHA):

- **CPHA = 0** means that data changes on traling edge and it is sampled on leading edge;
- **CPHA = 1** means that data changes on leading edge and it is sampled on trailing edge.

Finally, to complete the transmission, CS signal is deactivated and the SCLK returns idle. In Table 1, all the possible combination of clock phase and clock polarity are summarized, with the resulting behavior.

CPOL	CPHA	Idle CLK	Data change	Sampling
0	0	0	Falling	Rising
0	1	0	Rising	Falling
1	0	1	Rising	Falling
1	1	1	Falling	Rising

Table 1: Behavior for each combination of CPOL-CPHA

Let us assume a device needs to transmit 6 nibbles to another one. Before starting, the previously discussed parameters are to be defined according to desired behavior. In this case, we assume that CPOL is set to 0 and CPHA to 1. Then, the CS signal is deactivated, and on each rising edge of the clock, a nibble is sent. On the falling edge, the receiver samples the I/O lines. In Figure 1, the waveforms for the transmission are shown.

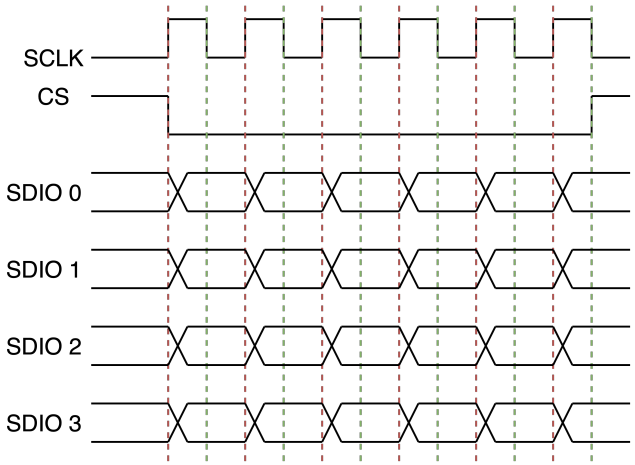

Figure 1: Waveforms related to a transmission of 24 bits with CPOL = 0, CPHA = 1

3. System Architecture and Behavior

This Section is meant to give an overview of the implemented design. Notice that the core is meant to work with the multi-IP architecture already designed for the **SE*cube*™** FPGA[1], based on the **IP Manager** component. To keep things brief, the structure will not be presented nor explained again, except in case of some examples, where it is necessary.

3.1. Design overview

The design is composed of two cores:

QSPI CORE Identified by ID = 0x01, it is the core providing QSPI capabilities;

LED-BLINKER CORE Identified by ID = 0x02, it provides the possibility to control the LEDs of the **SE*cube*™** DevKit for debug purposes, as will be better explained later.

Plus, the FPGA hosts also the IP Manager component and a Data Buffer of 64 16-bit shared memory locations for exchanging information with the CPU. Therefore, the overall architecture that will be mapped onto the FPGA is the one in Figure 2.

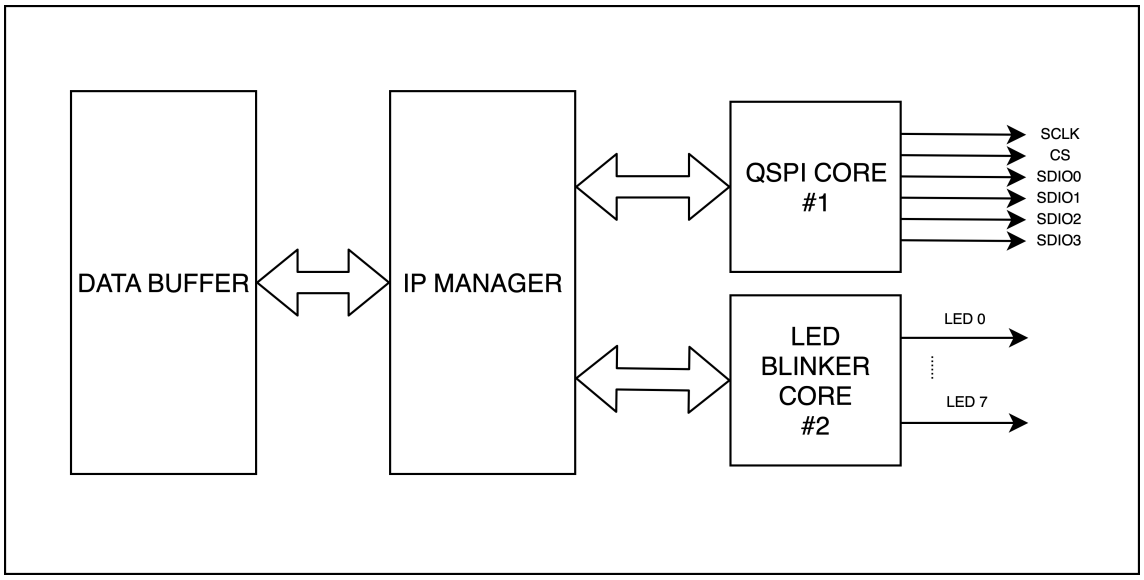

Figure 2: FPGA internal architecture.

3.2. LED-BLINKER Core

This section describes the additional core that has been inserted for testing purposes in the design, since it is not necessary for Quad-SPI to work.
The core is identified by the ID = 0x02. It is initially inactive, waiting for a request coming from the CPU. When it comes, the core reads a byte from the address 0x01 of the data buffer. These 8 bits are then used to drive the 8 LEDs of the FPGA accordingly. In practice, if a bit is zero, the corresponding led will be turned off, otherwise it will be turned on. The bit are mapped as shown in Figure 3.

[1] https://www.secube.eu/resources/open-source-projects/, under **IP-core Manager for FPGA-based design**.

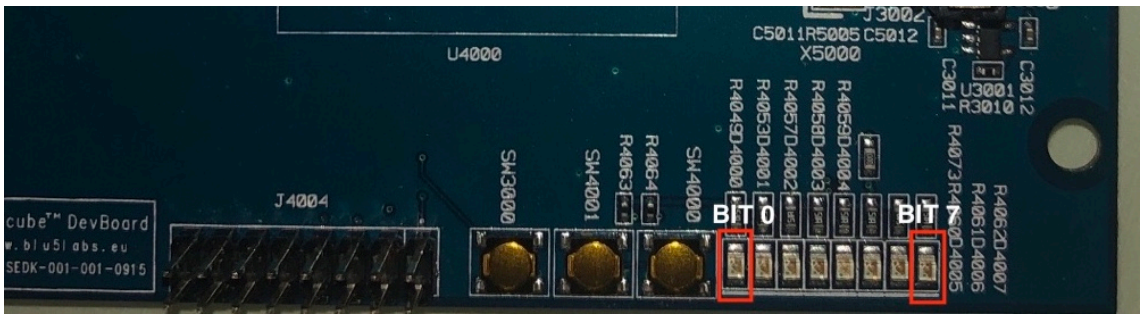

Figure 3: The mapping between the bit of the word read from the data buffer and the led on the **SE*cube*™** .

The core can be used to visualize the word written by the CPU. Therefore, it will be particularly useful when testing the reception of bits over Quad-SPI. As a matter of fact, when the **SE*cube*™** receives data over Quad-SPI, it can be forwarded from the Quad-SPI core to the IP-BLINKER, checking whether they are the expected bits or not. The core is really simple and low-area, so it does not affect the overall design significantly.

3.3. Quad-SPI Core

The core, whose architecture is shown in Figure 4, is structured as an FSM-D. Its activities can be summarized into two main groups: reading/writing from/to the data buffer to communicate with the CPU, and sending/receiving data through Quad-SPI to communicate with a slave device. In the following, the communication with the CPU is dealt from an high-level point of view. For a detailed explanation of the protocol, how transactions are opened and similar, please refer to the documentation of the general IP Manager project.

After the reset, or after any operation is finished, the core enters in an IDLE state. At each

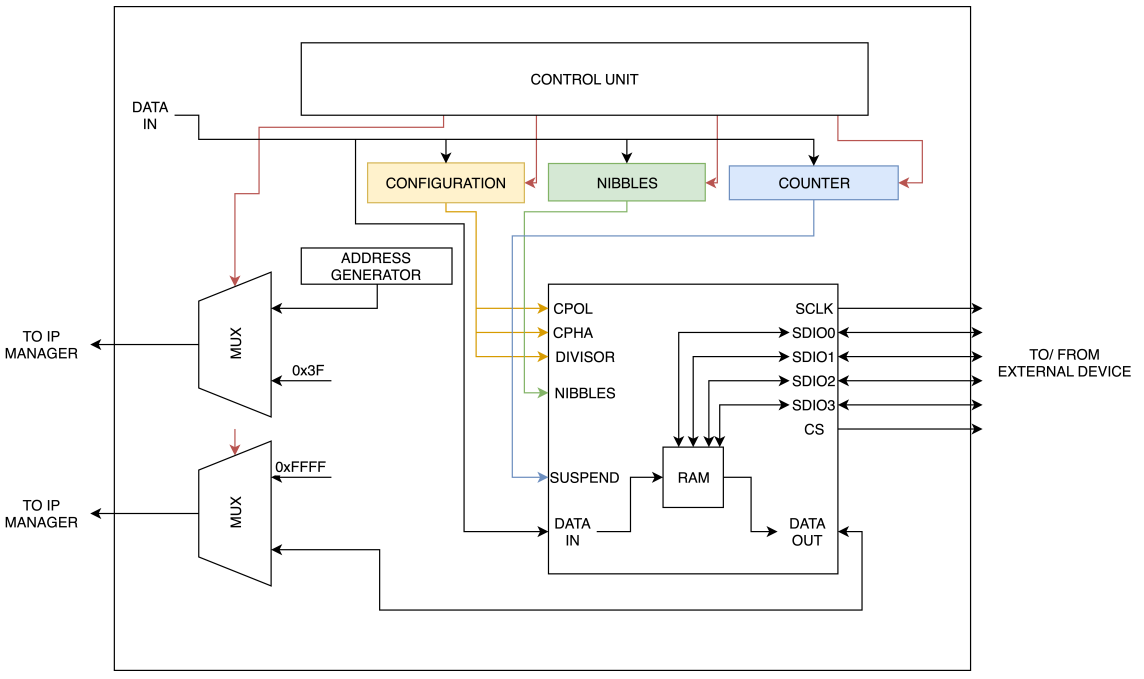

Figure 4: Overview of the architecture of the QSPI core.

clock cycle, it monitors the enable signal coming from the IP Manager. When that signal is asserted, meaning that the CPU has requested the functionalities of the core, this last checks for the received OPCODE. According to it, three are the possible choices. Either the CPU wants to configure the core, or it wants to send/receive some data. Each one of them is discussed right after.

3.3.1. Configuration

The phase is entered whenever the core in IDLE state receives a request from the CPU with the OPCODE = 0x00. As discussed in Section 2, Quad-SPI requires some parameters before the transmission can take place: the clock frequency, the clock polarity and the clock phase. This state is committed just to their definition. Therefore, as soon as it is entered, the core reads from the data buffer two 16-bit words, from addresses 0x01 and 0x02 respectively. They are supposed to contain all of the three parameters. More specifically, the structure of the two words is in Figure 5 and Figure 6 respectively.

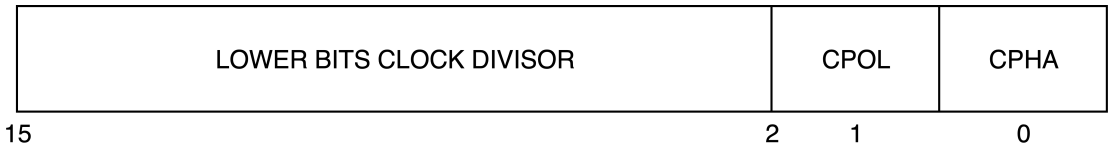

Figure 5: Structure of the control word read from address 0x01 of the data buffer.

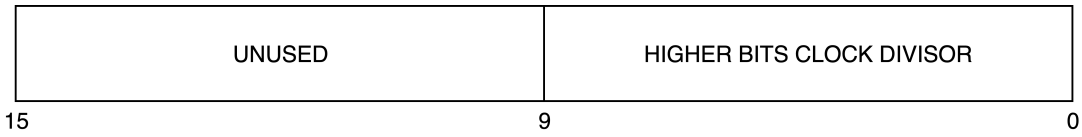

Figure 6: Structure of the control word read from address 0x02 of the data buffer.

As it can be seen, the clock divisor is expressed on 24 bits, allowing to use a clock frequency which is really low (that will be really useful for debugging purposes). The minimum value this parameter can assume is 1. Once they have been read, they are stored inside the dedicated registers, to be used for all the future communications up to a new reconfiguration. Notice that this configuration must be performed *at least once before any other operation*, otherwise the core will not work properly. Once the configuration registers are updated, the core exits the CONFIGURATION state and goes back to IDLE, waiting for the next request.

3.3.2. Send

This state is in charge of establishing a communication with the slave device that is supposed to be connected to the pin of the FPGA, as we will see in the next Section. It is entered whenever the core in IDLE state receives a request from the CPU with OPCODE = 0x01. In order to handle the transmission, the core needs to know how many nibbles are to be sent. Thus, first the core reads from address 0x01 of the data buffer this value. This is loaded in the corresponding register, where it will be kept until the end of the transmission. Once the number of nibbles is known, the core reads from the data buffer the exact number of words, considered that one word is composed by four nibbles.
The size of the transmission has been fixed to 1024 as a technical choice. This means that for each

transaction the CPU opens with the core, a transfer of 2KB at maximum is allowed. In order to make this possible, the core reads the buffer circularly. Therefore, once the end is reached, if the number of words to send is greater than 63, the core starts reading again from address 0x01.
An internal storage is needed to store the data read from the data buffer. To this end, the RAM available on the FPGA has been exploited. This choice helps reducing the space occupied by the core, leaving space for the co-presence of other cores. The dimension of the RAM has been fixed to 2KB (maximum size of the transfer). To sum up, when this state is reached, one word is read from the data buffer at CPU write cycle, and is stored inside the internal RAM. In case the end of the data buffer is reached, the core starts reading again from address 0x01. This process repeats until the words are over, and we have all of them stored inside the internal storage of the core. At that point, according to how to communication with the core has been opened, there are two possible scenarios:

- The CPU opened a transaction in polling mode, so it waits for the transmission to be over, without closing the transaction;
- The CPU opened a transaction in interrupt mode, so the transaction is closed as soon as the last word is written inside the internal RAM.

In both cases, when the reading process is terminated, the core enters in the **transmission** phase. Hence, the CS signal is driven low, signaling to the slave device that a transmission is about to start. According to the parameters, which have been configured as explained in Section 3.3.1, the core starts generating the clock signal (SCLK) for the transmission with the proper frequency, and the data (read from the internal RAM) is sent on the four data lines (SDIO[3:0]) according to the CPOL-CPHA configuration, as explained in Table 1. Obviously, when the number of sent nibbles reaches the limit, the core asserts the CS signal, so that the slave knows that the transmission is over, the SCLK returns idle, and the core exits from the transmission state. Then:

- If the CPU opened a transaction in polling mode, the core writes the unlock code at address 0x3F. The CPU, which is continuously reading from that address, reads the unlock code as soon as it is written, and disables the core;
- If the CPU opened a transaction in interrupt mode, the core raises an interrupt and wait for an acknowledgement. Sooner or later, the CPU will respond, and the core will be disabled.

At that point, it returns to the IDLE state, waiting for another transaction.

3.3.3. Receive

This state is entered whenever the core in IDLE state receives a request from the CPU with OPCODE = 0x02. As already discussed for the sending part, the core needs to know how many nibbles must be received, because the SCLK signal must be generated for that precise number of times. Hence, the core expects the CPU to write at address 0x01 of the Data Buffer the expected number of nibbles. Once it is read, it is stored inside the corresponding register. In case the transaction was opened in polling mode, the CPU waits, otherwise it is closed. At that point, the core enters the RECEIVING state by driving low the CHIP SELECT signal, to notify the slave that it is ready to receive data. It generates the SCLK with the frequency set during the configuration phase, and samples the bits on the four data lines, according to CPOL and CPHA, as explained in Table 1. Also in this case, the received data is stored inside the internal RAM, one nibble per time. Once the end of the expected nibbles has been reached, the clock returns idle, the CS signal is asserted and the communication stops. At that point, all the data are in the internal RAM of the core, but they must be read from the CPU. According to how the transaction was opened, there are two possible scenario:

- The CPU opened the transaction in polling mode, so the core unlocks the CPU writing the unlock code at address 0x3F, and then it resumes;
- The CPU opened the transaction in interrupt mode, so the core raises an interrupt and waits for the CPU to respond. As soon as it does, the CPU sends an acknowledgement to the core, so that it can resume.

Whatever the modality, once the core resumes its activity, the CPU starts waiting for a fixed amount of time. This is necessary because the core needs some time to copy the content of the internal RAM inside the Data Buffer. Once this time is elapsed, the CPU reads from the Data Buffer the data received. However, as already stated, the maximum size per transfer is 2KB (1024 words of the data buffer), which means that once the core has copied 63 words to the Buffer, it has to stops. Meanwhile, the CPU, which was waiting for a certain amount of time (long enough to allow the core writing 63 words), reads the data. Each time a read is completed, the core receives a signal proving that the CPU has read a word. Once the core has received 63 times this kind of notification, it resumes the copying process (starting again from address 0x01), whereas the CPU stops reading, and it starts waiting again. This process repeats until all the words received through Quad-SPI from the slave are read by the CPU. Notice that, even though the number of words is lower than 63, to keep things simple, the CPU still waits for the time necessary to write 63 words. Anyhow, when the process is over, the CPU disables the core, which returns to the IDLE state waiting for the next transaction.

4. Application Program Interface

This Chapter describes the communication driver developed for interfacing the user application and the Quad-SPI core on the FPGA.
For a correct communication, the complete driver should be composed of two layers: a high- level one, composed of the specific functions for managing the task of each IP core, and a low-level one, containing the low-level functionalities for the communication with the FPGA. The developed APIs take into account the mutual concurrency of high-level drivers.
The project presented here focuses on providing the application programmer a reliable high-level layer, over which it is possible to create the program exploiting all the functionalities of the core. Before using any of the functions described in this Chapter, the FPGA should be configured to support the IP Manager environment, without forgetting to insert calls to

```
B5_FPGA_Programming();
FPGA_IPM_init();
```

in order for the FPGA to be ready. `FPGA_IPM_init()` also initializes a semaphore to resolve concurrency issues, as we will see later. Once it is done, you can start writing your own program.

4.1. High-level driver

The high-level communication with the core is implemented by the functions declared in the header file “qspi_fpga.h”. Such functions are supposed to be called by the user application code, without having almost any knowledge of the specific hardware implementations and of the details about the micro-controller and the FPGA provided by **SE*cube*™** . Because of the nature of the concerned core , three types of APIs have been provided. These are:

Configuration - Used to configure the core;

Transmit - Used to send data from the **SE*cube*™** to a slave device;

Receive - Used to receive data from a slave device.

Due to foreseen usability and the user convenience, the transmission function supports three different width: 8, 16 and 32 bit. The user should only pass the address where the data is stored, and then it will be treated differently (as 8, 16, or 32 bits wide) according to the called function. The function is responsible of making the data compatible with the interface of the core. As for the reception, data is considered only 16-bit wide, and it is written to the address provided by the user. When finished, it is up to the user converting the data to the appropriate data length before using it. Right-after, the APIs and their explanation are listed.

```
int FPGA_QSPI_CONF(uint32_t clk_divisor, uint8_t clk_polarity,
   uint8_t clk_phase)
```

This function is used to configure the Quad-SPI core, therefore it is mandatory to call it at the beginning of any application that makes use of the core, otherwise it will not work as expected. The parameters are:

- `clk_divisor`, which sets the clock divisor, that will be used to obtain the desired clock frequency starting from the one of the FPGA. It can be any number between 1 and $2^{24} - 1$;
- `clk_polarity`, which sets the polarity of the clock. It can be 0 or 1;
- `clk_phase`, which sets the phase of the clock. It can be 0 or 1.

For a detailed explanation of these parameters, please refer to Section 2.

```
int FPGA_QSPI_SEND_8bI(uint16_t n_nibbles, uint8_t* data, int
   interruptMode);
```

This function allows to open a transaction with the core in send mode, to transmit 8-bit wide data. The following parameters are required:

- `n_nibbles`: it indicates how many groups of 4 bits the user is willing to send. For example, 16 bits are composed of 4 nibbles;
- `data`: it is the initial address in memory where the data to be sent is stored;
- `interruptMode`: it indicates how the transaction should be opened: polling if 0, interrupt if 1.

The opening of the transaction is possible only if there are no active transactions. Any other transaction is blocked by the software semaphore. The procedure to open a transaction follows these steps:

1. Check if the transaction can be established: if not, the request is rejected;
2. Lock the resource (FPGA) by decrementing the semaphore;
3. Update control variables to the new values;
4. Perform a write operation at address 0x02 of the buffer;
5. Send a positive response to the calling function if everything went fine.

The function returns a value(1) that notifies whether the operation was correctly performed.

```
int FPGA_QSPI_SEND_16bI(uint16_t n_nibbles, uint16_t* data, int
   interruptMode);

int FPGA_QSPI_SEND_32bI(uint16_t n_nibbles, uint32_t* data, int
   interruptMode);
```

These two functions are basically the same of the previous one, except for the fact that they deal with 16-bit and 32-bit wide data respectively.

```
int FPGA_QSPI_RECEIVE_16bI(uint16_t n_nibbles, uint16_t* data,
   int interruptMode);
```

This function is used to received data from a slave device. As already stated, data is treated as 16-bit wide. The parameters are:

- `n_nibbles`: it indicates how many groups of 4 bits the user is willing to receive;
- `data`: it is the initial address in memory where the user wants to store the received data;
- `interruptMode`: it indicates how the transaction should be opened: polling if 0, interrupt if 1.

4.2. Concurrency issues

Concurrency is a big concern that may affect the correct behavior of the system if not correctly managed. As we said, the management is operated through the implementation of a semaphore inside the driver that allows the execution of one and one only transaction at the time. The semaphore is managed by the following functions:

1. `FPGA_QSPI_CONF()`, that configures it;
2. `FPGA_QSPI_SEND_XbI()`, that checks the value of the semaphore. In case the resource is unlocked, the function zeroes the semaphore and allows the beginning of the current transaction. Otherwise, the function immediately returns with an error;
3. `FPGA_QSPI_RECEIVE_16bI()`, that increments the semaphore releasing the resource if and only if there is an active transaction and the caller of the function is the caller that has opened the active transaction.

5. User Manual

The following Chapter is committed to explaining how to use the core, starting from its insertion in the development environment.

5.1. Core installation

To start with, you need to add the two cores to the working environment, that is the one containing the IP Manager and the Data Buffer. Hence, you need a custom Device-Side **SE*cube*™** project opened in Eclipse, ready to work with the FPGA available on the **SE*cube*™** . To do so:

1. Download the files related to the IP Manager and the data buffer from the **SE*cube*™** site[2];
2. Create a new Device-Side project by following the steps that are listed in the related documentation. Be sure to add all the necessary files to communicate with the FPGA;
3. Create a new Lattice Diamond™ project. When you are asked to import the VHDL files, be sure to include the files named “CONSTANTS.vhd”, “DATA_BUFFER.vhd”, “IP_MANAGER.vhd”, which you downloaded at point 1. Moreover, you must add all the VHDL files related to the Quad-SPI and the Led Blinker. These files are located inside the folder named ”VHDL”;
4. Proceed with the creation of the project, as explained in the following subsections of the IP Manager documentation;
5. Once the project is created, synthesize it (check that no timing errors are present) and produce the files containing the bitstream by following the steps contained in the following subsections of the IP Manager documentation. You should have obtained two files with name ending with “_algo.c” and “_data.c”, which will be used later;
6. Proceed by adding everything to the Device-Side project, as explained in Section 6.4 of the IP Manager documentation. When you are asked to substitute the content of the two arrays in the file “TEST_FPGA.h”, use the files obtained at point 5;

Now the bitstream describing the architecture is statically saved on the flash memory image to be programmed into the device. To use its functionalities, you need to include the files containing the correlated APIs, with the following steps:

1. To import the necessary files in your Device-Side project, select ”File » Import…”, then “Filesystem” and press “Next”
2. Browse to the directory where the API libraries for the Quad-SPI core are located, which is called ”API”
3. Select all the files inside the folder (”qspi_fpga.c”, ”qspi_fpga.h”).
4. Open the file ”Fpgaipm.h” and include the file ”qspi_fpga.h” by inserting the following line after all the others inclusions:

```
#include "qspi_fpga.h"
```

5. In the same file, modify the function `EXTI9_5_IRQHandler()`, which is located at the bottom of the file, adding the following lines to the switch-case statement, necessary to call the interrupt handler when an interrupt request comes from the core:

[2] https://www.secube.eu/resources/open-source-projects/, under **IP-core Manager for FPGA-based design**.

```
case 1:
handle_IR();
break;
```

Now the core is correctly inserted in the environment. You can now create your own program, in the file ”main.c”, and use the core to send and receive data over Quad-SPI, as described in section 4. Once finished, you may want to program the **SE*cube*™** , by following these simple steps:

1. Save the changes to all files
2. Go to ”project » Build Configuration » Set Active” and ensure the tick is on ”Release”
3. Build the project
4. Connect the **SE*cube*™** to the PC
5. Flash the produced executable on the device by right-clicking on it in the Project Explorer and selecting the Release binary under ”Target » Program Chip” (i.e., select the label containing the string “/Release”)

When the process starts, the LEDs associated to the FPGA will be set in a weak pull-up state, meaning that the FPGA is being programmed. At the end, they should turn off. As soon as they turn off, the program you wrote starts executing.
You will need a second device which should be programmed accordingly, connected to the **SE*cube*™** , for sending and receiving data over Quad-SPI. Read Section 5.2 for an example of how a possible communication can be established.

5.2. A first project: communicating with Arduino

To check whether the procedure in Section 5.1 has been executed correctly without errors, and prove that the core is actually working, you may want to carry out a simple experiment. The purpose of this Section is to provide a tutorial that illustrates, step-by-step, how it is possible to send and receive data through the core. The slave device that will be used in the communication is an **Arduino**™ UNO board, from Arduino™ . It has been chosen because it is easy-to-use, cheap, and can be reused for a lot of other projects. Moreover, it is supported by all the operating systems, its setup takes few minutes, and there is a lot of documentation online, which makes the development really easy.
As already said, Quad-SPI allows to send and receive data. Hence in the following, we will test the core in both operating conditions. First, we will be using the **Arduino**™ board to read data sent by the **SE*cube*™** , and then we will use it as a transmitter, sending the data to the **SE*cube*™** .

5.2.1. Hardware resources

The hardware you will need is:

- PC
- **SE*cube*™** DevKit
- **Arduino**™ UNO board (it can be bought online for 20€)[3]
- 6 F/M jumpers, to connect the **Arduino**™ UNO to the **SE*cube*™** .

[3] https://store.arduino.cc/arduino-uno-rev3

5.2.2. Software resources

In addition to the software necessary to create the Device-Side project and synthesize the VHDL code, you will need the Arduino IDE, to program the Arduino UNO board. It can be downloaded from the Arduino official site[4]. The supported operating systems are Windows, Linux and Mac OS.

5.2.3. Arduino setup

Let's start by gaining some familiarity with the **Arduino**™ board and its development environment.

1. Start by downloading the Arduino IDE from the official website, choosing the version according to your operating system, and install it
2. Once the installation is terminated, open it
3. Connect the **Arduino**™ UNO board to the USB port of your PC
4. In the Arduino IDE, go to "Tools » Board", a list of boards should appear. Select the "Arduino/Genuino UNO" board, as in Figure 7

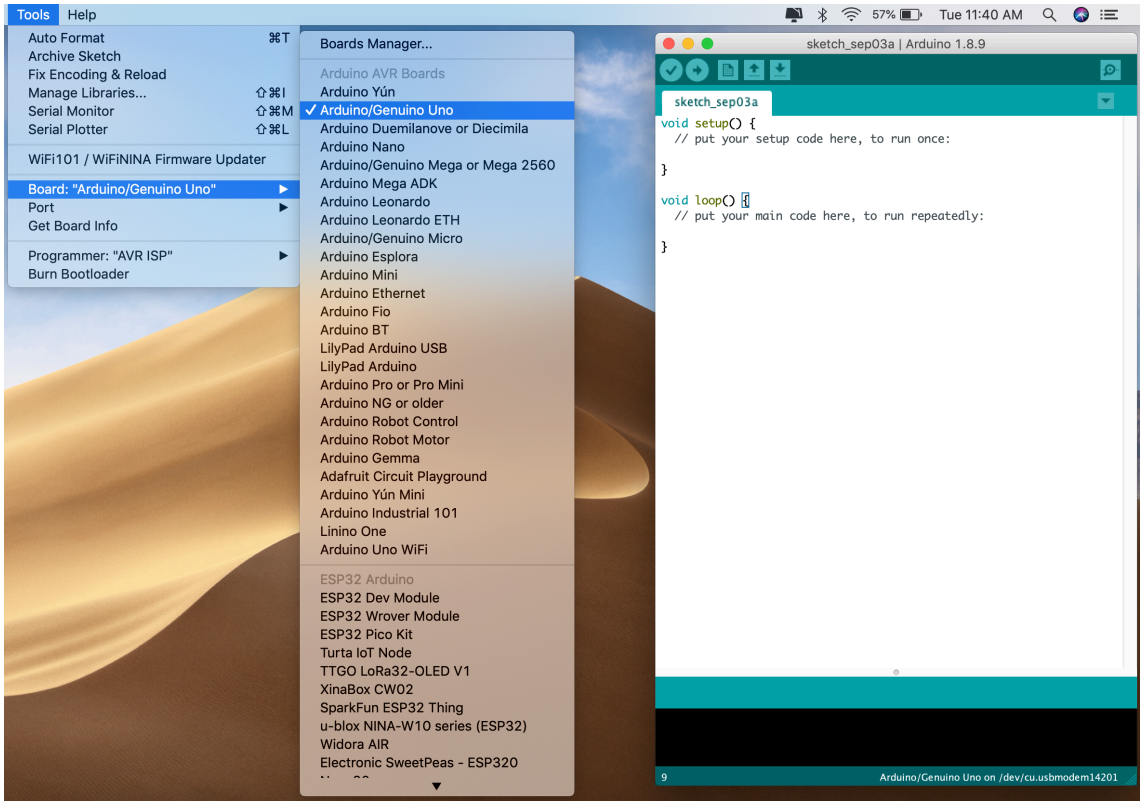

Figure 7: Selecting the **Arduino**™ Uno Board

5. Select "File » Examples » 01.Basics » Blink". A new window, containing some code should open. It is a script that makes the LED of your **Arduino**™ UNO blink

[4] https://www.arduino.cc/en/Main/Software

6. Select ”Sketch » Upload”. Now the code is being downloaded on the board. At the end of the process, the LED should start blinking, meaning that everything has been setup correctly.

You may notice that the script is made of two functions: `setup()` and `loop()`. The first one should contain code that is executed only one time, at the beginning. On the contrary, the code inside the `loop()` function, is executed repeatedly after the `setup()` has finished. This must be the structure of every Arduino program.
Another important feature that we will use for our communication example is the Serial Monitor. You can open it by clicking on the icon in the top-right corner of the window, see Figure 8. It shows the value that are being printed by your program. We will use it to print the data we are receiving, or sending.

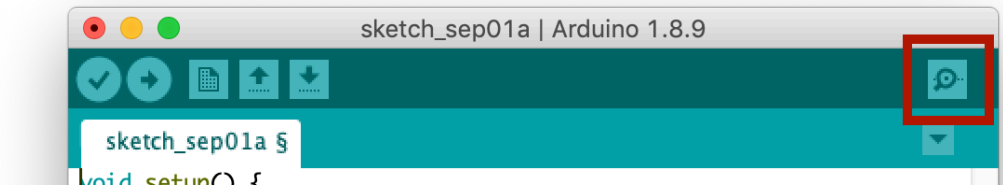

Figure 8: The serial monitor where data is printed.

If there are no errors, you can proceed to the next section, otherwise, be sure to resolve any issue before moving on.

5.2.4. Connecting the two devices together

This section contains information related to the connection of **Arduino**™ to the **SE*cube***™ .
As stated in Section 2, the communication over Quad-SPI uses 6 lines: 1 for the clock, 1 for the chip select, and 4 lines for the data. In Figure 9, you can look at which are the pins of the **SE*cube***™ that are connected to the Quad-SPI core, and how they are used. These lines have to be attached to 6 pins on the **Arduino**™ UNO. More precisely, if you are going to use the code associated to this document, the pins are configured in this way:

- D12 for the clock
- D13 for the chip select
- D8-D7-D4-D2 for SDIO[3:0] respectively.

However, this configuration is not mandatory. You can use whatever digital pins of the **Arduino**™ UNO, as long as you change the code accordingly. When you are finished, the connection should look like in Figure 10. Finally, connect both the **SE*cube***™ and the **Arduino**™ to your PC.

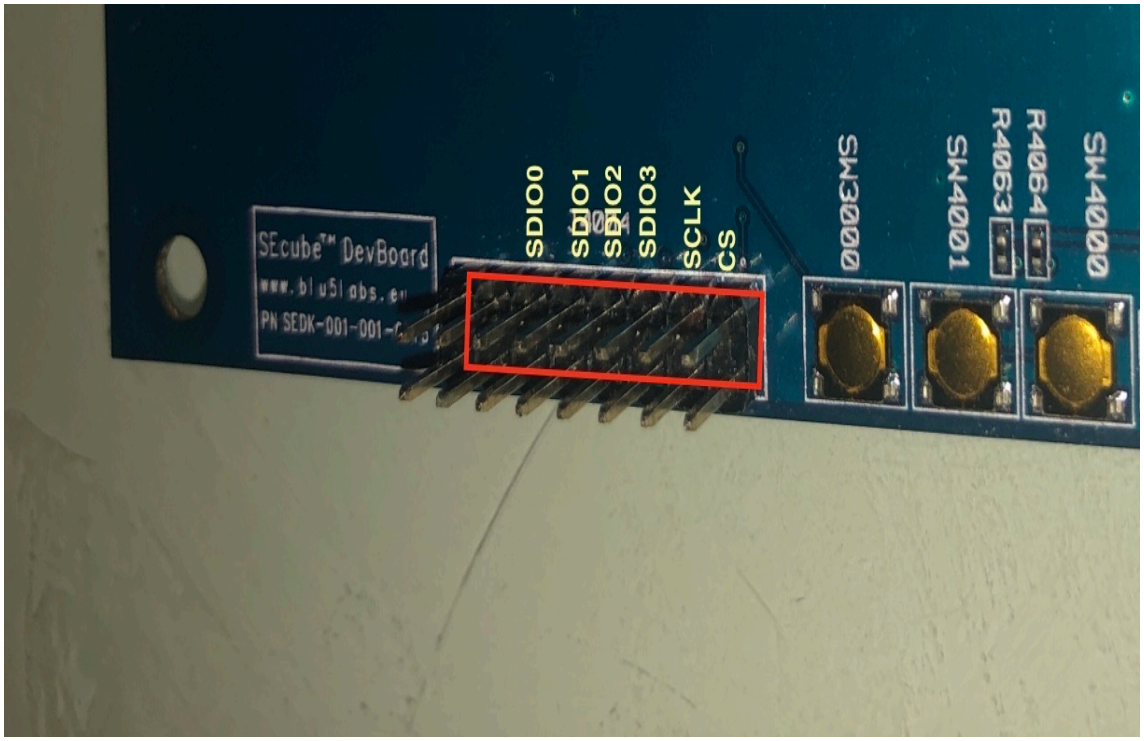

Figure 9: The pins of the **SE*cube*™** that are used by the Quad-SPI core.

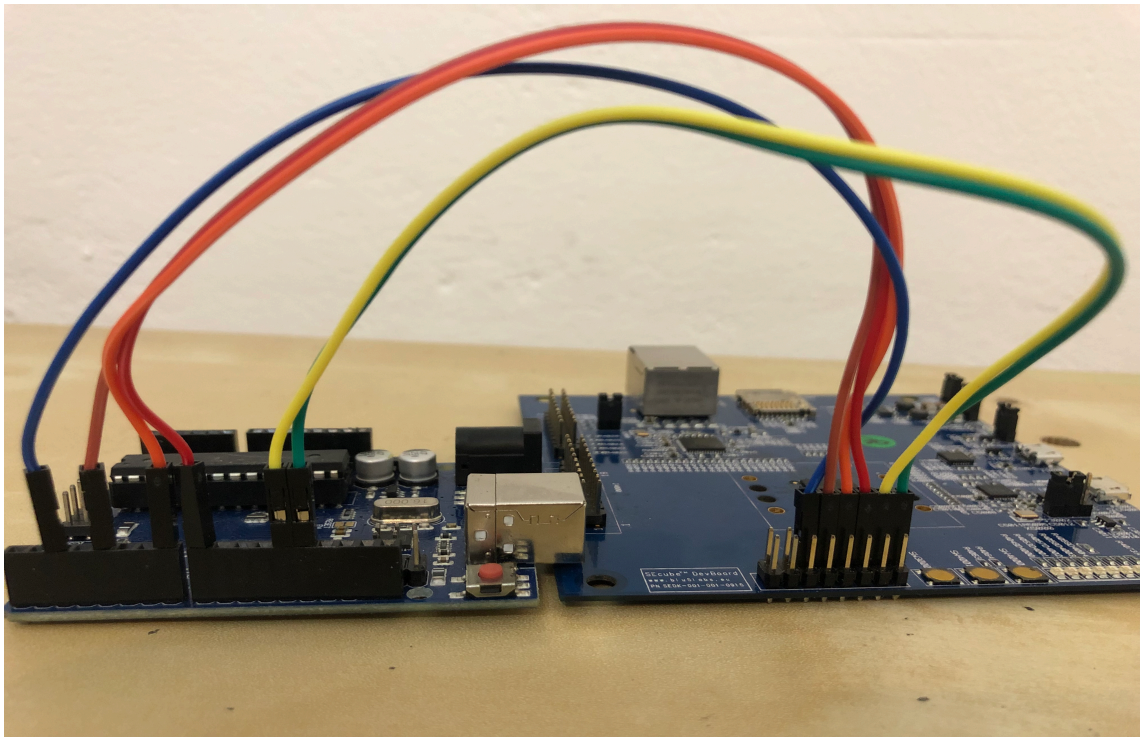

Figure 10: Overview of the connection between the **SE*cube*™** and the **Arduino™** UNO

5.2.5. Testing Quad-SPI - Send

Let's deal with the situation in which we want to use the **SE*cube*™** as master of the communication to send some data to the **Arduino**™ UNO. We are going to work in polling mode, but nothing changes if interrupt is used instead. The necessary code is contained in A.1. An explanation of the code follows, to make things clearer.
First of all, considering how the Quad-SPI works, as explained in Section 2, we must initialize the core by setting its parameters: clock divisor (which will be used to determine the clock frequency), the clock polarity, and the clock phase. In this example, we will be using a really high clock divisor, resulting in a slow transmission. This choice is due to the fact that we are using as a slave device, a board that does not provide native support for Quad-SPI. Therefore, the piece of code we are going to use, is a simple loop that reads the value on the four digital pins connected to the data pins of the **SE*cube*™** . This implies that, in case a lower clock divisor is utilized, we will not be able to read all the data through the **Arduino**™ UNO because of the high speed of the transmission. Therefore, for the purpose of the experiment, do not change it. As for the clock polarity and the clock phase, both have been set to one.
Once the core has been configured, we must declare the array of the data we want to send. In this case, we have declared an array of 10 numbers, each one expressed on 16 bits. Then, using a for loop each element is initialized, from 0 to 9. Once completed, we can send the data through Quad-SPI by simply calling the associated function. Remember that the function requires also the number of nibbles we want to send, that is easily obtained:

$$nibbles = \frac{(words * width)}{4} \tag{1}$$

To embed it in your project, follow these steps:

1. Open the device-side project
2. Open the file "main.c"
3. Copy paste the code right after the call to the function `B5_FPGA_Programming()`
4. Save the changes to all files and build the project
5. Connect the **SE*cube*™** to the PC
6. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under "Target » Program Chip" (i.e., select the label containing the string "/Release")
7. Wait few minutes. The LEDs of the FPGA should be set in a weak pull up state, meaning that it is being programmed, and at the end, all of them should be turned off.

The part related to the **SE*cube*™** is over. Now we need the **Arduino**™ to be able to read the data we are sending. To do so, follow these steps:

1. Open the Arduino IDE
2. Go to "File » Open", and select the file "QSPI-Receive.ino", located in the folder "Arduino/QSPI-Receive"
3. Go to "Sketch » Upload", and wait for the process to finish
4. Open the serial monitor, by clicking on the icon placed in the top-right part of the screen
5. In the serial monitor page, be sure that the baud rate is set to 19200.

The code contained in the Arduino script is straightforward. As explained in Section 2, when the master starts the communication, the CS signal is set to 0.
Therefore, since the arduino board is the slave device, it waits for the chip select signal to go to 0. Once a zero has been read, 4 bits are read on each rising edge of the clock (because we set CPOL = 1 and CPHA = 1, check the Table 1 to see when the sampling shall be performed), and they are printed on the serial monitor.
Now the code for both devices are ready. If they are not connected to each other already, please connect them as described in 5.2.4. When everything is connected, press the RESET button (it is the first button next to the LEDs of the FPGA) on the **SE*cube*™** and wait few minutes. Remember to not close the Serial Monitor. At some point after the reset (as soon as the FPGA is programmed), the **SE*cube*™** should start sending the bits, which should appear on the serial monitor.

5.2.6. Testing Quad-SPI - Receive

The procedure and the motivations required for testing the reception mode, are almost equal to one used for the test of the transmission mode. As first, let's consider the code for sending data from the **Arduino™** board, which is located in "Arduino/QSPI-Send". Initially, the slave is waiting for the master to initiate the communication, therefore it waits until the chip select signal is set to zero. As soon as a zero is detected, each falling edge of the clock, a new value on the digital pins is written. When the chip select will be asserted again from the master, the process will stop.
To upload this code:

1. Open the Arduino IDE
2. Go to "File » Open", and select the file "QSPI-Send.ino", located in the folder "Arduino/QSPI-Send"
3. Go to "Sketch » Upload", and wait for the process to finish
4. Open the serial monitor, by clicking on the icon placed in the top-right part of the screen
5. In the serial monitor page, be sure that the baud rate is set to 19200.

On the other side, the **SE*cube*™** will now be in charge of receiving and storing the data that **Arduino™** UNO will send. The code to add in the main file of the **SE*cube*™** project is reported in A.2. A brief explanation follows.
Once the core has been initialized with its parameters, the function to receive data is called, and the data will be stored in the desired array passed to the function. Once the reception is over, you may want to check what you have received. This is where the additional core, the LED-BLINKER (section 3.2) comes in help. Once the communication with **Arduino™** is over, for each 16-bit word we have received, we split it in two parts of 8-bit each. Each of these two parts is sent to the LED-BLINKER core. In this way, the LEDs on the **SE*cube*™** will turn on/off, according to the data received, and the correctness can be proved. To embed the code in your project, follow these steps:

1. Open the device-side project
2. Open the file "main.c"
3. If you previously added the code related to Section 5.2.5, please remove or comment it
4. Copy paste the code right after the call to the function `B5_FPGA_Programming()`
5. Save the changes to all files and build the project

6. Connect the **SE*cube*™** to the PC
7. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under «Target » Program Chip» (i.e., select the label containing the string "/Release")
8. Wait few minutes. The LEDs of the FPGA should be set in a weak pull up state, meaning that it is being programmed, and at the end, all of them should be turned off.

Notice that, if you have already uploaded your code on the Arduino UNO, during the programming phase the CS pin is set to 0. This is recognized from **Arduino™** as if it were the start of the transmission, even if it is not. Therefore, you should notice on the serial monitor that while the FPGA is being programmed, the transmission start. If this is the case, once the FPGA has been programmed, re-upload the program to **Arduino™** .
At some point, you should see on the serial monitor of Arduino that the transmission is started. When the transmission will be over, the LEDs of the **SE*cube*™** will reflect that data sent by Arduino.

A. Appendix

Here are reported all the piece of code used in the example.

A.1. SE*cube*™ - Send

```
FPGA_QSPI_CONF(16777215, 1, 1);
uint16_t data_to_send[10];
for (int i = 0; i < 10; i++) {
  data_to_send[i] = i;
}
FPGA_QSPI_SEND_16bI(40, &data_to_send[0], false);
```

A.2. SE*cube*™ - Receive

```
FPGA_QSPI_CONF(16777215, 1, 1);
uint16_t data_to_receive[10];
FPGA_QSPI_RECEIVE_16bI(9,&data_to_receive[0], false);

HAL_Delay(10000);

FPGA_IPM_DATA temp;
for (int i = 0; i < 3; i++) {
  temp = data_to_receive[i] & 0xFF;
  FPGA_IPM_open(2,0,0,0);
  FPGA_IPM_write(2,1,&temp);
  FPGA_IPM_close(2);
  HAL_Delay(2000);
  temp = data_to_receive[i] >> 8;
  FPGA_IPM_open(2,0,0,0);
  FPGA_IPM_write(2,1,&temp);
  FPGA_IPM_close(2);
  HAL_Delay(2000);
}
```