IP-core Manager for FPGA-based design 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

Maurizio DI LORENZO maurizdl@gmail.com
Simone MACHETTI simonemachetti@gmail.com
Alessandro MONACO alessandro.monaco.94@gmail.com
Flavio PONZINA flavio.ponzina@gmail.com
Paolo PRINETTO *(President, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(CINI Cybersecurity National Lab)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by Blu5 View Pte Ltd. Other brands and names mentioned herein may be the trademarks of their respective owners. No use of these may be made for any purpose whatsoever without the prior written authorization of the owner company.

Disclaimer

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN “AS IS” BASIS AND ITS AUTHORS DISCLAIM ALL WARRANTIES, EXPRESS, OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PURPOSE. THE SOFTWARE IS PROVIDED TO YOU “AS IS” AND WE MAKE NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER WITH RESPECT TO ITS FUNCTIONALITY, OPERABILITY, OR USE, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PURPOSE, OR INFRINGEMENT. WE EXPRESSLY DISCLAIM ANY LIABILITY WHATSOEVER FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR SPECIAL DAMAGES, INCLUDING, WITHOUT LIMITATION, LOSS REVENUES, LOST PROFITS, LOSSES RESULTING FROM BUSINESS INTERRUPTION OR LOSS OF DATA, REGARDLESS OF THE FORM OF ACTION OR LEGAL THEREUNDER WHICH THE LIABILITY MAY BE ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY LIKELIHOOD OF SUCH DAMAGES.

Contents

1 Features

This chapter presents the features of the CPU-FPGA communication system we have in mind. The idea is to create a flexible channel between the CPU and the IP cores accommodated inside the FPGA of **SE***cube*™ . Particular attention is given to performance, parallelism and hardware design. The cooperation between the two actors of the communication is intended to follow these specifications:

- The CPU must communicate with the IP cores through transactions, i.e., sets of data exchanged from an initial packet which opens the connection to a final one which closes it
- Transactions are exclusive: the CPU cannot handle a transaction with more than a single IP core at a time
- Inside the FPGA, several IP cores might work simultaneously
- Each core is not aware of the presence and the status of other cores inside the FPGA
- The typical transactions between the CPU and a core is the following: the CPU opens a transaction with a core and writes on a portion of dual-side-accessed shared memory its inputs, which are read and processed by the core that then writes results again on the same block of memory; the response can arrive either within the transaction or after a certain time after the transaction has been closed: in this case, the core requests to interrupt the CPU before writing the results
- The FPGA should be efficiently designed, whatever the number of cores inside
- The cores can be designed in whatever way, but they must present a standard interface for the communication
- The cores can be implemented in such a way to support an error signalling system to the central manager and thus to the CPU.

2 System Architecture

2.1 Global Architecture

The core of the **SE***cube*™ Hardware device family is a 3D multi-module SoC (System-on-Chip), integrated in a 9mm x 9mm BGA package. The single chip embeds three hardware components: a powerful processor, a flexible FPGA, and an EAL5+ certified smart card. In this section, we want to give an overview of the global system we are dealing with, in order to have clear in mind the communication possibilities between the CPU and the FPGA: for such reason, we are going to present with more details the CPU and the FPGA blocks and their relative interface.

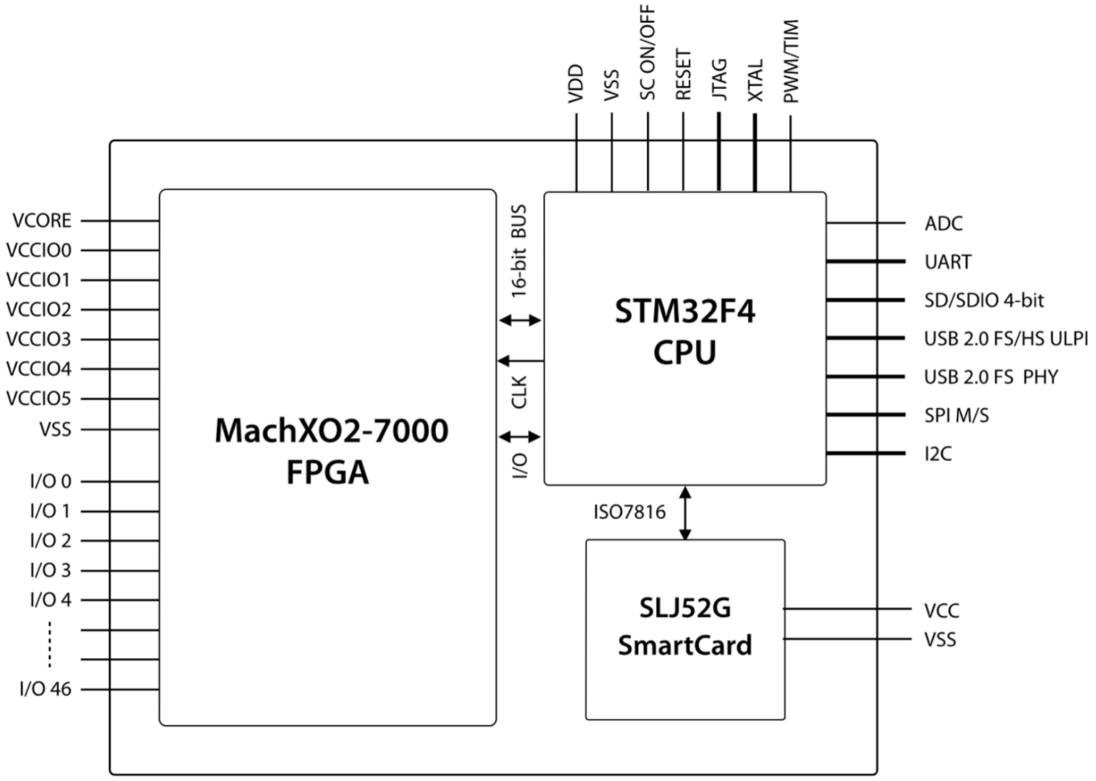


Figure 1: The **SE***cube*™ Hardware Architecture

2.1.1 The CPU

The processor adopted within the **SE***cube*™ is the STM32F429, produced by ST Microelectronics™ , which includes a high-performance ARM Cortex M4 RISC core and provides the following features:

- 2 MiB of Flash memory
- 256 KiB of SRAM
- 32-bit parallelism
- Operating frequency of 180 MHz
- Low power consumption.

This CPU has been selected among many ARM-based microcontrollers, since it offers several features that make it suitable for high-performance and security-oriented solutions. For example, it supports the Cortex CMSIS implementation that provides, among the others, the CMSIS-DSP libraries: a collection with over 60 DSP functions for various data types. The CMSIS-DSP library allows developers to implement complex, real time operations using the embedded hardware Floating Point Unit.
In addition, the CPU provides several peripherals such as SPI, UART, USB2.0 and SD/MMC, which ease the hardware integration in diverse devices. For example, a secure USB device can be easily realized using the USB2.0 and the SD card interfaces, respectively.
On the security side, a TRNG (True Random Noise Generator) embedded unit, hardware mechanisms like MPU (Memory Protection Unit), and privileged execution modes allow implementing the security strategies required by a certified secure controller (e.g., privileged memory areas, key generation, etc.).
For programming, debug, and testing operations, the CPU provides a standard JTAG interface that can be permanently disabled once the development cycle is over, protecting all the sensitive information through a physical hardware lock.

2.1.2 The FPGA

The FPGA element, a Lattice MachXO2-7000 device, is based on a fast, non-volatile logic array providing the following main features:

- 7,000 LUTs
- 240 Kib embedded block RAM
- 256 Kib user flash memory
- Ultra low-power device.

The FPGA exposes 47 general-purpose I/Os which may be used as a 32-bit external bus able to transfer data at 3.2 Gib/s.
As outlined in Figure 1, within the **SE***cube*™ Chip the FPGA is connected to the CPU through a 16-bit internal bus, providing a data transfer rate of up to 1.6 Gib/s.
A CPU-FPGA clock line is provided to simplify the clock domains synchronization. To limit the number of pins and the BGA package size, the FPGA JTAG is connected just to the embedded CPU, which manages both the debug and the programming operations. Therefore, the FPGA configuration can be implemented by means of customized, high-security techniques. For example, the programming bitstream can be encrypted and signed through dedicated algorithms. The CPU and/or the smartcard elements can then be used to decrypt and verify it before its injection into the FPGA.

2.2 FPGA-CPU connection

Here is presented the pin-to-pin interface between the CPU and the FPGA, which represent a fundamental point for our work.
The set of interconnections is used for exchanging data, control, and status signals, including:

- Clock, reset and interrupt signals
- JTAG interface for programming the FPGA
- Other control lines

The configuration within the **SE***cube*™ treats the FPGA as an external memory device (PSRAM), leveraging the Flexible Memory Controller (FMC) available on the processor. The FMC is an interfacing peripheral of the microcontroller used to connect external memories such as NOR Flash, NAND Flash, SRAM, and PSRAM[1], as in this case. With this configuration, each pin assumes a specific behaviour, its value and transitions being directly managed by the FMC.

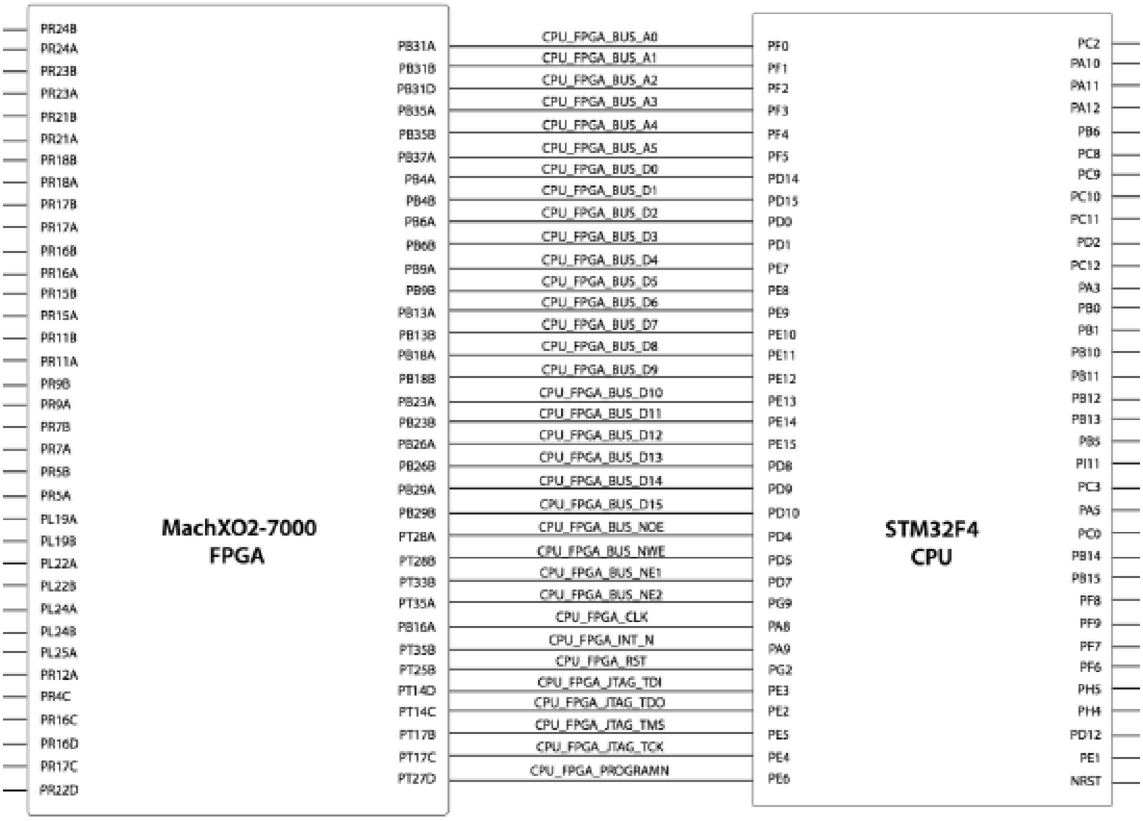


Figure 2: FPGA-CPU pin configuration within **SE***cube*™

The available pins within the bus are:

- Address – 6 pins (CPU_FPGA_BUS_A0:5)
- Data – 16 pins (CPU_FPGA_BUS_D0:15)

[1] For additional details, please refer to STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs Reference Manual, Chapter 37: https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#page=1602&zoom=100,0,116

- Chip select – 2 pins (CPU_FPGA_BUS_NE1:2)
- Clock – 1 pin (CPU_FPGA_CLK)
- Controls – 2 pins (CPU_FPGA_{INT_N, RST})
- JTAG – 5 pins (CPU_FPGA_JTAG_{TDI, TDO, TMS, TCK}, CPU_FPGA_PROGRAMN).

2.3 FPGA internal structure

The internal architecture to be accommodated on the FPGA embeds three main blocks: a dual-ported data buffer where inputs and outputs are exchanged, the IP cores customized for executing a given task, and a central IP manager for routing, arbitrating and configuring the communication.

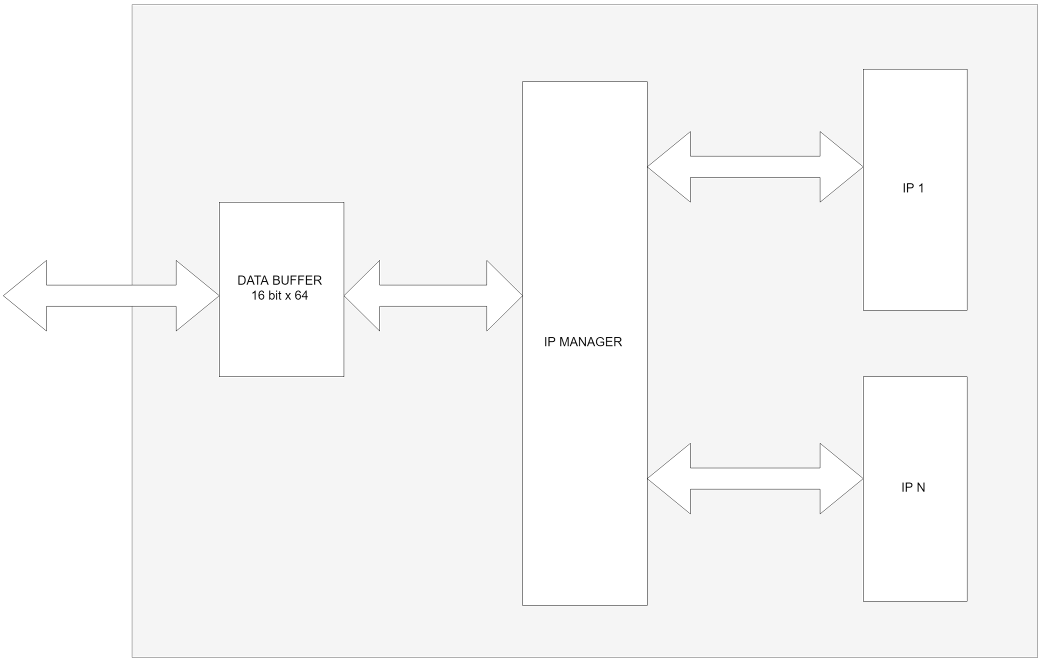


Figure 3: FPGA internal structure

2.3.1 The data buffer

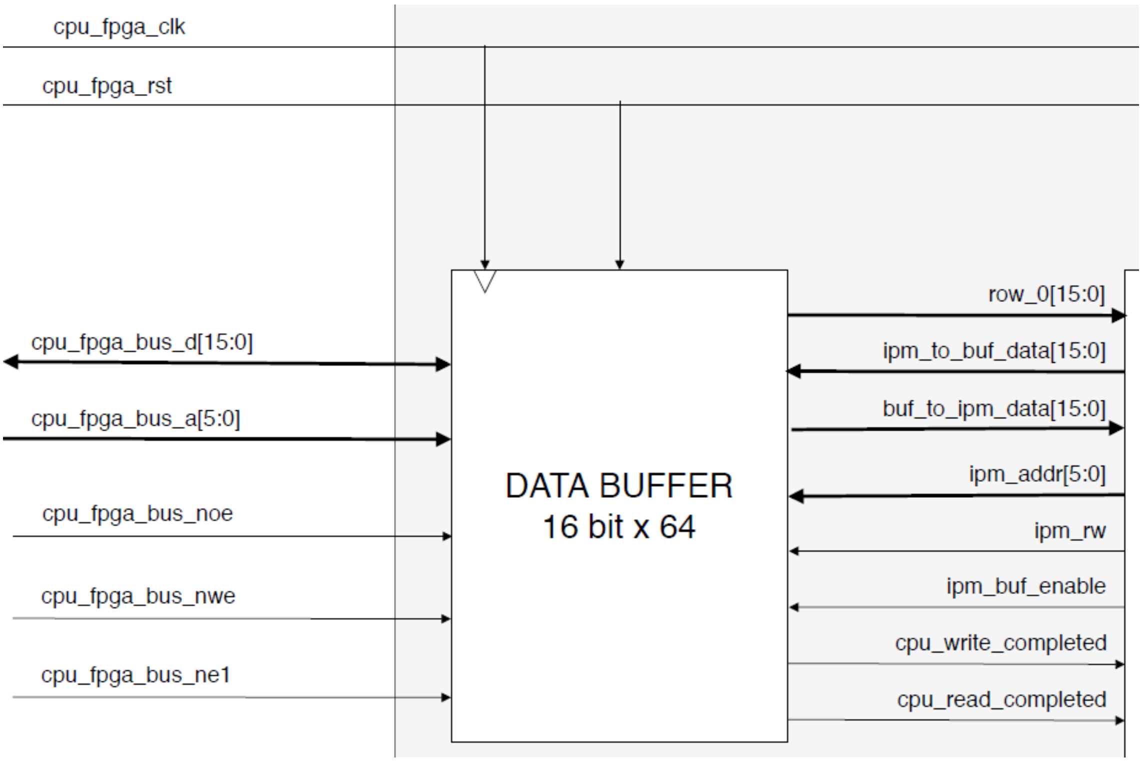


Figure 4: Data buffer schematic

The data buffer represents the memory interfacing module with the CPU, where commands and inputs for the FPGA arrive and where outputs are stored at the end of the operations. It is a dual-ported portion of 128 B of shared synchronous SRAM, organized in 64 16-bit words. The buffer presents two different interfaces at the two sides through which is accessed, so it is understandable the presence of a conspicuous logic that surrounds and controls the memory array. From the CPU point of view, because of the employ of the FMC, the entire FPGA is seen as a block of PSRAM, so signals and timings are accorded to this abstraction. Therefore, besides the common data, address, clock and reset signals, the buffer presents three additional specific signals for the FMC protocol: the generic enable (NE1), the output enable (NOE) and the write enable (NWE). Below here we report values and timing of these signals in a timing diagram which shows the behaviour of the protocol followed by the Flexible Memory Controller to interface the FPGA for reading and writing[2].

[2] STMicroelectronics, Reference Manual RM0090 for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs, Chapter 37, "Flexible Memory Controller (FMC)": https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#page=1602&zoom=100,0,116

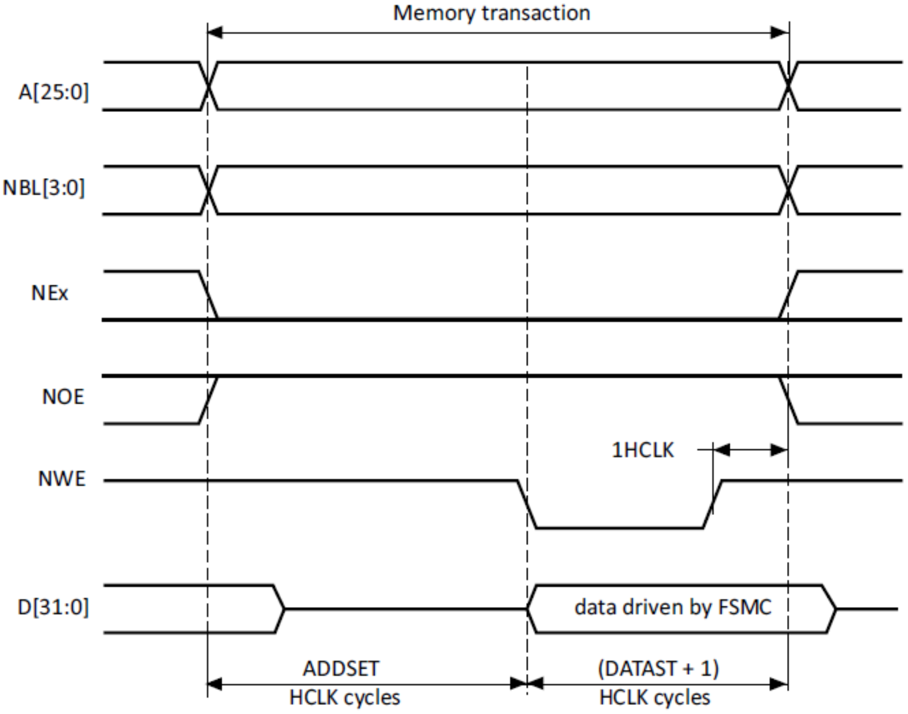


Figure 5: Mode A write access waveforms of the FMC protocol for STM32F4xx MCU

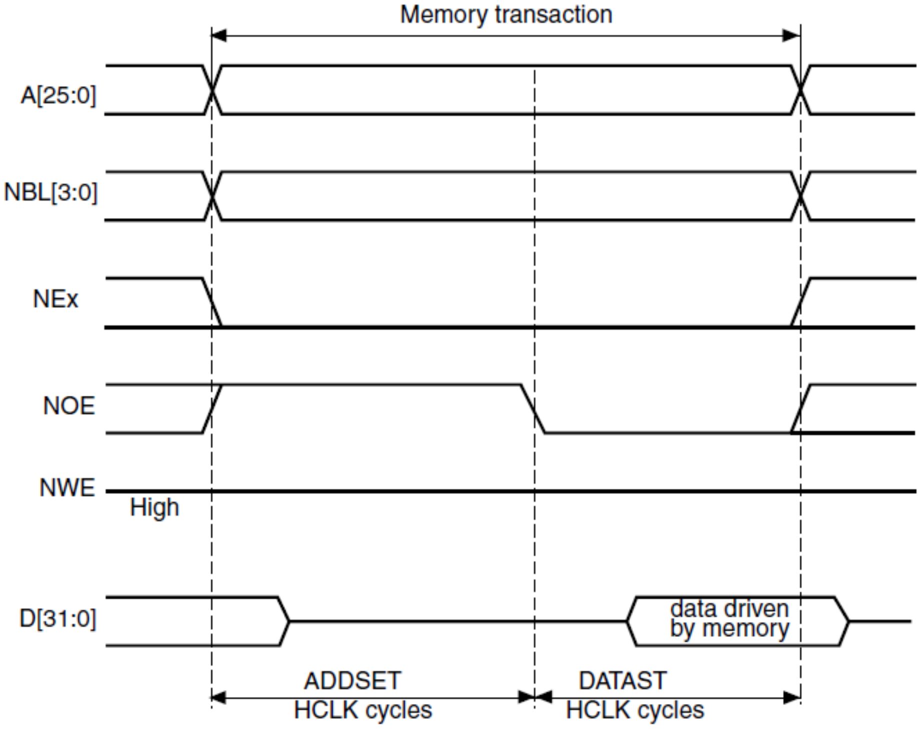


Figure 6: Mode A read access waveforms of the FMC protocol for STM32F4xx MCU

The Flexible Memory Controller is able to manage 4 different banks of memory within a specific address range. Bits [27:26] of the AHB address bus (HADDR) are interpreted by the FMC as the identifier for 1 of the 4 banks, and enable the activation of the corresponding NEx signal (NE1, NE2, NE3 or NE4), which is active low. The CPU-FPGA interconnection allows then to accommodate just 2 of these 4 banks of memory, presenting both NE1 and NE2 pin in the CPU interface.
Bits [25:0] of the HADDR bus are instead interpreted as the actual external memory address, but actually just the lowest 6 of them are used.
When NEx is low, a memory operation is taking place. The address is forwarded and a setup time for its stabilization is awaited. Then there is the data phase: if the NWE not asserted, the address is intended to be a read address, and the CPU waits for a data response from the memory for a given data setup time. If a write is to be performed, NWE is asserted and a word to be written is forwarded and maintained for the data setup time. These setup times are decided by the software configuration of the FMC and must be provided to the FPGA design as generic VHDL parameters. The machine within the buffer recognizes when the FMC enters the address setup time and the data setup time looking at the values of the interface signals, and once understood sets an internal decrementing counter with the generic parameter values and waits for the end of the different phases before acquiring a valid address and a valid data, or before providing it.
At the end of any CPU operations, the internal machine is able to send to the right-side interface two strobes which indicate the completion of a read of or a write, as it is possible to see in Figure 4. These signals are useful to synchronize the activity of the internal modules, as we will see.
From the internal side, the buffer is accessed in a simpler way as a standard bank of synchronous fast memory, with an enable signal, two data busses for input and output, an address bus and a read/write signal. These signals are in control of the module which is enabled to communicate by the Manager.
The first row of the buffer is reserved to the command word for the Manager, as we will see in the following chapters. Since this word is to be continuously monitored by the Manager to sense any CPU command, the row is directly reflected to an output port.

2.3.2 The IP cores

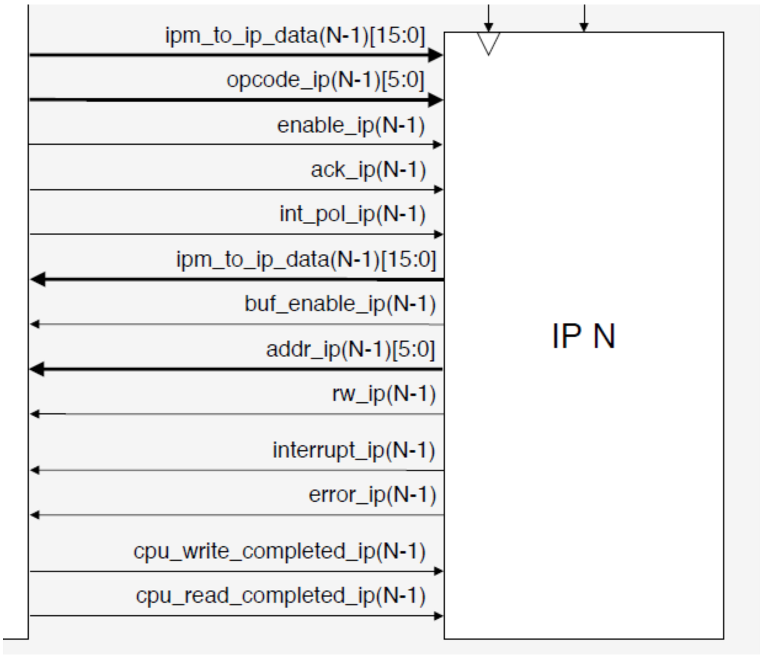


Figure 7: Example of IP core with its interface

The IP cores are the target modules of the CPU-FPGA communications, as they represent the hardware accelerators needed by the CPU to enhance speed in executing some dedicate algorithms and tasks. The designer is thus free to internally design the core as preferred, but must present at least the following mandatory ports (Figure 7):

- Clock and reset input signals
- An enable signal as starting strobe for the internal machine
- A dedicate 6-bit signal for the possible operative code
- A single-bit input for the communication mode (interrupt/polling)
- 2 separate 16-bit buses for data input and data output
- 6-bit address bus
- 2 command signals for the data buffer (enable, read/write)
- 2 dedicate interrupt and error output lines
- An acknowledgment (ACK) input line
- CPU read and write completion input strobes

A core remains inactive until the enable signal coming from the Manager is asserted: this must be the event that triggers its internal state machine. The enable signal ensures the core that the internal-side signals of the buffer corresponds one-to-one to its memory interfacing signals: the buffer answer now to its commands. Therefore, the IP core can start a pipelined communication with the master thank to the strobe signals which indicate the end of a read or of a write operation by the CPU.

The core can be designed to support more than one operation, and both polling and interrupt communication mode: the mode and the possible operative code are written by the CPU in the command word of start transaction and are forwarded to the module at the enable time by the manager through two dedicated signals. When in polling mode, the CPU does not close the transaction until the output are returned on the buffer. It just awaits the completion of the computation polling a dedicate address or waiting for some predefined time necessary to the core to end its job. When in interrupt mode, the core and the CPU have an input-transmitting transaction first, then the communication is closed, and the core starts computing. When done, the core asks to the manager the interruption of the CPU. When it is acknowledged, the core has a second transaction during which the outputs are written on the shared memory. The system also provides support for error signalling from the IP cores, for example after an internal failure detected by a built-in self-test procedure. A dedicate error signal is present in the interface to advise the Manager, and consequently the CPU, of the problem.

2.4 The IP Manager

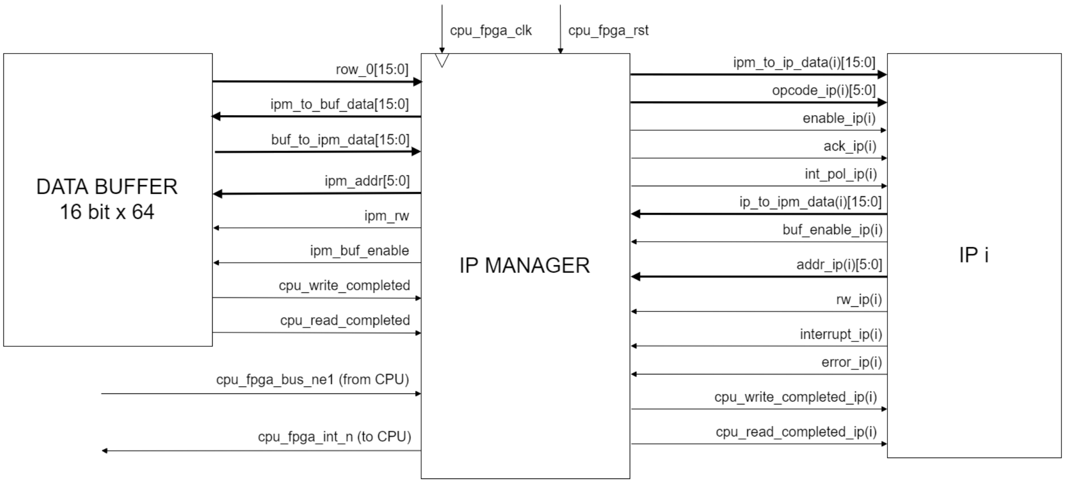


Figure 8: The IP Manager connections

The IP Manager is the central main block of the architecture. It is the intelligent multiplexer/demultiplexer of the system. Its main intent is to put in communication the IP core addressed by the CPU with the buffer where data are exchanged. The IP Manager-IP core interface is replicated for each core placed in the design.

After the global reset, the module is in its IDLE state and waits for a request from the CPU. This is forwarded under the form of a control word written in the first row of the buffer, row which is immediately reflected towards the Manager not to lose time for addressing it before reading. At each clock cycle during the IDLE state, the IP Manager monitors this signal, detecting at some point the word of start transaction. As we will see in the next sections, this word contains the ID number of the IP core desired by the CPU plus an operative code and other configuration bits. One of them indicates the begin of a transaction. The IP Manager then enters in its MULTIPLEXING state, putting into direct contact the signals on its IP interface with the ones on its buffer interface, asserting the enable signal to the correct core, and forwarding to the corresponding interface the operative code and the communication mode (interrupt/polling). The Manager detaches this link only when the word of end transaction is written at row 0 of the buffer, disabling the core and taking back the control of the signals to the buffer. During the multiplexing state, the module is blind to any other request from the cores not involved in the communication and even to CPU commands different from the end of transaction one, such that the exclusivity of the transactions is preserved.

Once exited the MULTIPLEXING state, the Manager returns IDLE. When IDLE, it is sensitive to any interrupt or error request coming from the IP cores. If the interrupt line of one IP core is high, the Manager raises the global interrupt signal of the FPGA towards the CPU, writing at address 0x00 of the buffer the ID of the core that desires to interrupt the CPU. The CPU, after having read it, possibly sends a control word of acknowledge for that request, which triggers the raising of the corresponding ACK line of the Manager-core interface along with the enable. An acknowledgment transaction thus begins, during which the core probably communicates its results to the CPU and the Manager is in its MULTIPLEXING state, from which exits when the end of acknowledgment transaction command arrives, as usual. The CPU can also decide of not caring the interrupt from the FPGA, starting after some time another transaction without acknowledging the request, but loses the right to know the ID of the interrupting IP and its results.

If an error line is raised by a core, the Manager forwards an interrupt request from the CPU by itself, i.e., writing at address 0x00 its own ID, which is 0. An acknowledge for IP 0 from the CPU equals the begin of a transaction during which the Manager itself communicates on the buffer (at address 0x01) the ID of the IP which has signalled an internal fault. The faulty IP is not involved in this exchange of information as in the interrupt case, and its error line remains up until the problem is persistent. This cause no problem to the Manager, that once acknowledged by the CPU becomes blind to that line until its next rising edge.

The CPU sees the Manager as an actor of the communication in this case and in this case only. It is not possible, for the CPU, to start a transaction with the IP with ID 0 outside this situation, because that ID is normally considered out of range and, for this reason, ignored.

Although, thank to this ID reservation, future updates of the project could include a Manager that, for example, handles a transaction with the CPU to dynamically set which cores have the right to interrupt and which must be silenced, or to dynamically set the priority order to be followed in accepting core requests.

3 Communication Protocol

3.1 Overview

The following section is intended to describe the communication protocol that has been thought for the project. It is necessary to keep in mind that the goal is to allow a flexible communication channel between the CPU and the FPGA. The direct connection between these two entities seems to suggest that there is no need of a dedicated protocol. The point is that more cores are accommodated within the design. With a single core, the communication would be very straightforward, since the two sides would implement a point-to-point communication. In the common foreseen scenario, the CPU programs more than a single core, but the actual interface only allows the processor to read and write the buffer of the FPGA. Hence the need of an additional entity (the IP Manager) which has the aim of controlling the information exchange, and both the CPU and the cores need to follow a given protocol for their communications.
To manage exclusivity of communication between a single core at a time and the CPU, such protocol is based on transactions: the CPU must send a specific data packet to open or to close a transaction, which addresses one and one only IP. Once the transaction is open, the CPU and the selected core can perform the exchange of information. At the end, the CPU must send a packet to close the transaction.
As already mentioned before, the communication can be held in polling or in interrupt mode.

3.1.1 Polling

Polling is a very common way of communication between a master of a computing system and a slave such as a peripheral or a coprocessor. The master continuously checks the status of the interlocutor to monitor its state (which can be “something to communicate”, or “nothing to communicate”). The monitoring is done classically with a continuous read of the status on a shared location, a portion of a memory or a control register.
In our case, the CPU can decide to communicate with a core in polling mode when the core computation is thought to take a short time and can be waited idly without impactive loss of performance. Inputs are then sent to the core and the transaction is maintained opened until the results are ready.
When the communication is handled in this way, after the opening of the transaction, a phase of handshaking with delivering of the inputs follows. Such inputs can be read by the core as soon as they are written thanks to the write strobe sent by the buffer. The core, once read the last input, starts working, while the CPU sets the polling location to a locking value and starts reading it. When the core is done, it writes the outputs in the buffer without waiting any enabling signal, since the transaction is open and it has rights to access the buffer. As last write, it writes the unlocking code in the polling location. The CPU is thus unlocked and starts reading the outputs, then closes the transaction. The driver of the core must be obviously aware of the location in the buffer of the results, of the word to be polled and of the value to expect for the unlocking.

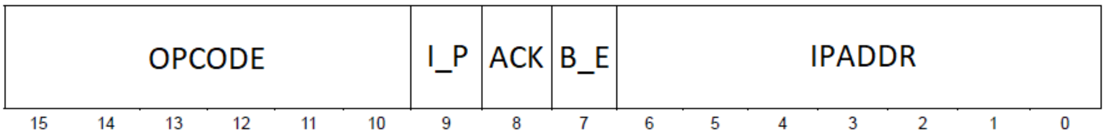
3.1.2 Interrupt

Interrupt is a good alternative to polling when working in a multitask environment. In this scenario, the master first programs the slave and then continues its own computation, while the slave reads the input, makes its job and at the end triggers an interrupt request to the master. This is typically the most common way of communication in modern systems, because it offers higher performance with respect of polling, because of the possibility for processor and accelerators of working in parallel. The hardware overhead consists of a single additional line for each slave (interrupt line).

In the **SE***cube*™ FPGA, the cores do not have a dedicate interrupt line directly towards the CPU, but the requests are all intermediated by the central IP Manager.
When the communication is held in interrupt mode, the CPU opens the transaction and writes the input as usual, while the core is enabled by the Manager and reads them in parallel as soon as they are written. Once written the last input, the CPU closes the transaction disabling the core, which probably has already started its computation once read the last input. When finished, the core raises its interrupt line transmitting the request to the Manager first, then the Manager writes at location 0x00 the ID of the core and forwards its request to the interrupt output line of the FPGA. The CPU receives the interruption and, sooner or later, may decide to respond. First, it reads location 0x00 where it retrieves the ID of the interrupting IP, then opens an acknowledgment transaction with this IP. At this point, the interrupt lines are cleared and the exchange of the output on the buffer starts. The CPU has the only limitation of leaving some time to the core to complete this write-out. Once completed this phase, the CPU closes the transaction.

3.2 The control word

The control word is the fundamental opening and closing word of any CPU-FPGA transaction. It is located at address 0x00 of the data buffer, which is immediately reflected to the IP Manager through a dedicate port, as previously said. The structure of the control word is reported below.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPCODE						I_P	ACK	B_E	IPADDR						

Figure 9: Bit fields of the control word

Bits 15:10	**OPCODE**	Operative Code	Hosts the possible operative code instructing the core on the task to be performed.
Bit 9	**I_P**	Communication Mode	0 : Polling Mode 1 : Interrupt Mode
Bit 8	**ACK**	Acknowledgment Transaction	0 : The transaction opened/-closed is a normal transaction 1 : The transaction opened/-closed acknowledges an interrupt request
Bit 7	**B_E**	Begin/End of transaction	0 : End Transaction 1 : Begin Transaction
Bits 6:0	**IPADDR**	Address of the IP core	Hosts the IP core identifier. Can assume any value from 0 to 127.

Whenever the Manager is IDLE and senses setting of bit 7, it enables the core specified by the field **IPADDR** and reflects on its interface the other fields. When bit 7 is cleared by the CPU, all the signals towards the cores are cleared consequently.
As will be presented in the following sections, we have developed a set of APIs to manage the communication via software, including also functions for opening and closing a transaction with given parameters that avoid the low-level bit-by-bit set of the control word.

In this format, the control word is always set and cleared via software. However, the location 0x00 can be used in a different format from the inside, when the IP Manager uses it to write the identifier of the core that requests to interrupt the CPU. In that case, bits 15 to 7 are cleared by the IP Manager.

0	0	0	0	0	0	0	0	0	IPADDR						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 10: Bit fields of the control word set via hardware during interrupt

3.3 Sequence diagrams

This section is intended to provide sequence diagrams of communication modes between the CPU and the cores.
Typical reading and writing operations are presented. As already mentioned, the communication is "pipelined", in the sense that there is no need of waiting that an entire block of data has been written/read before making the following read/write operation. This is achieved through the write strobe asserted by the FSM present in the data buffer.

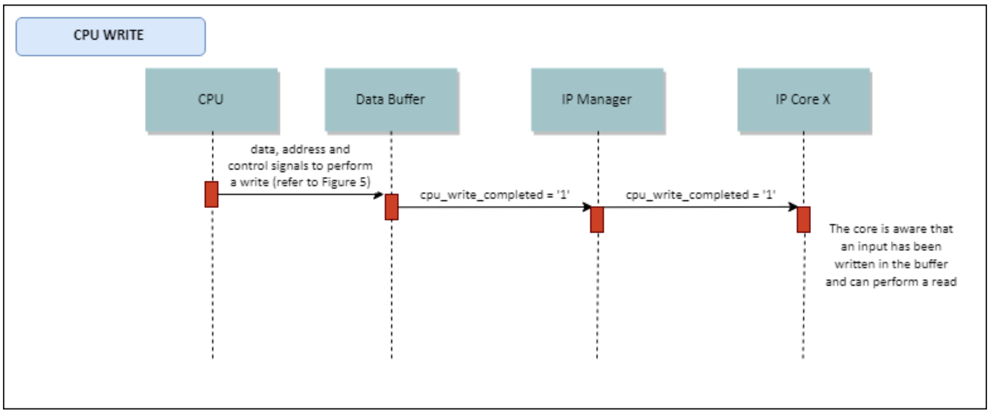

CPU WRITE
CPU
Data Buffer
IP Manager
IP Core X
data, address and control signals to perform a write (refer to Figure 5)
cpu_write_completed = '1'
cpu_write_completed = '1'
The core is aware that an input has been written in the buffer and can perform a read

Figure 11: Typical CPU write operation during a transaction with an IP core

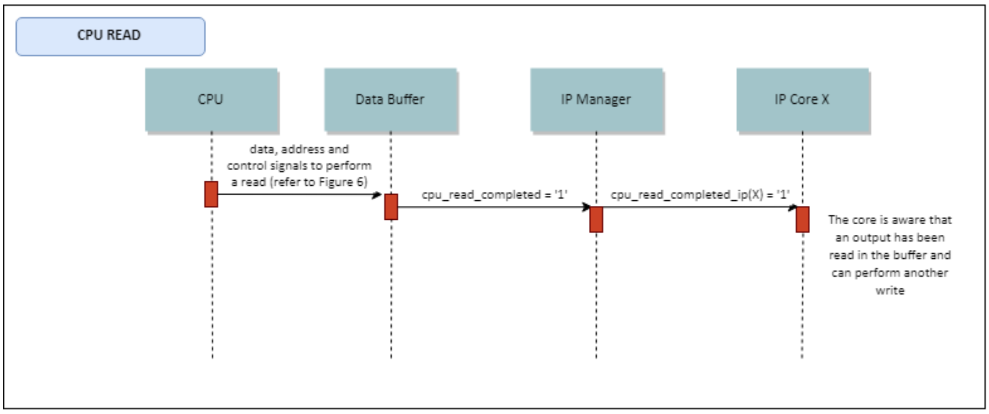

CPU READ
CPU
Data Buffer
IP Manager
IP Core X
data, address and control signals to perform a read (refer to Figure 6)
cpu_read_completed = '1'
cpu_read_completed_ip(X) = '1'
The core is aware that an output has been read in the buffer and can perform another write

Figure 12: Typical CPU read operation during a transaction with an IP core

The typical polling and interrupt transactions between the CPU and the FPGA are presented below.

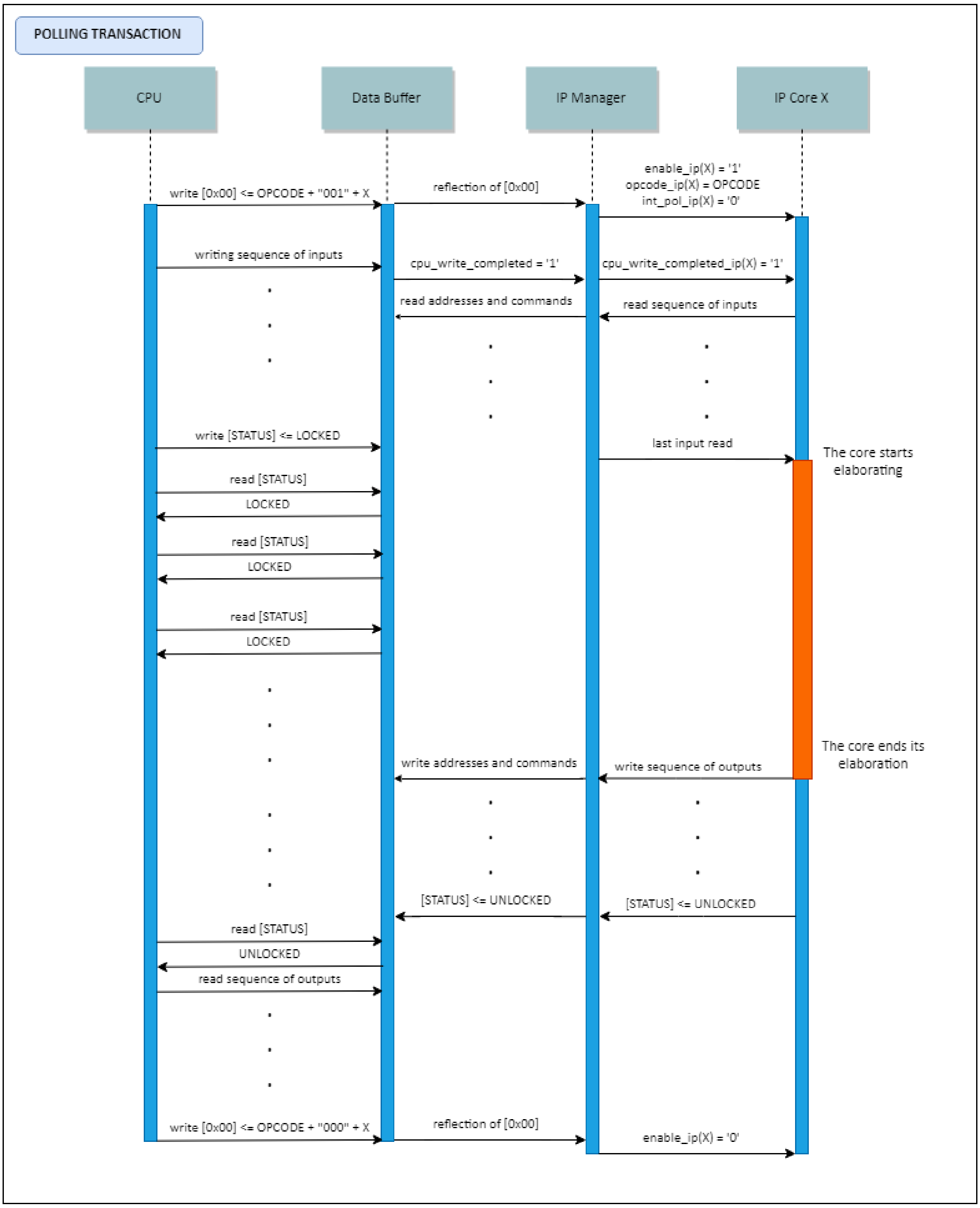


Figure 13: Polling transaction between the CPU and a core

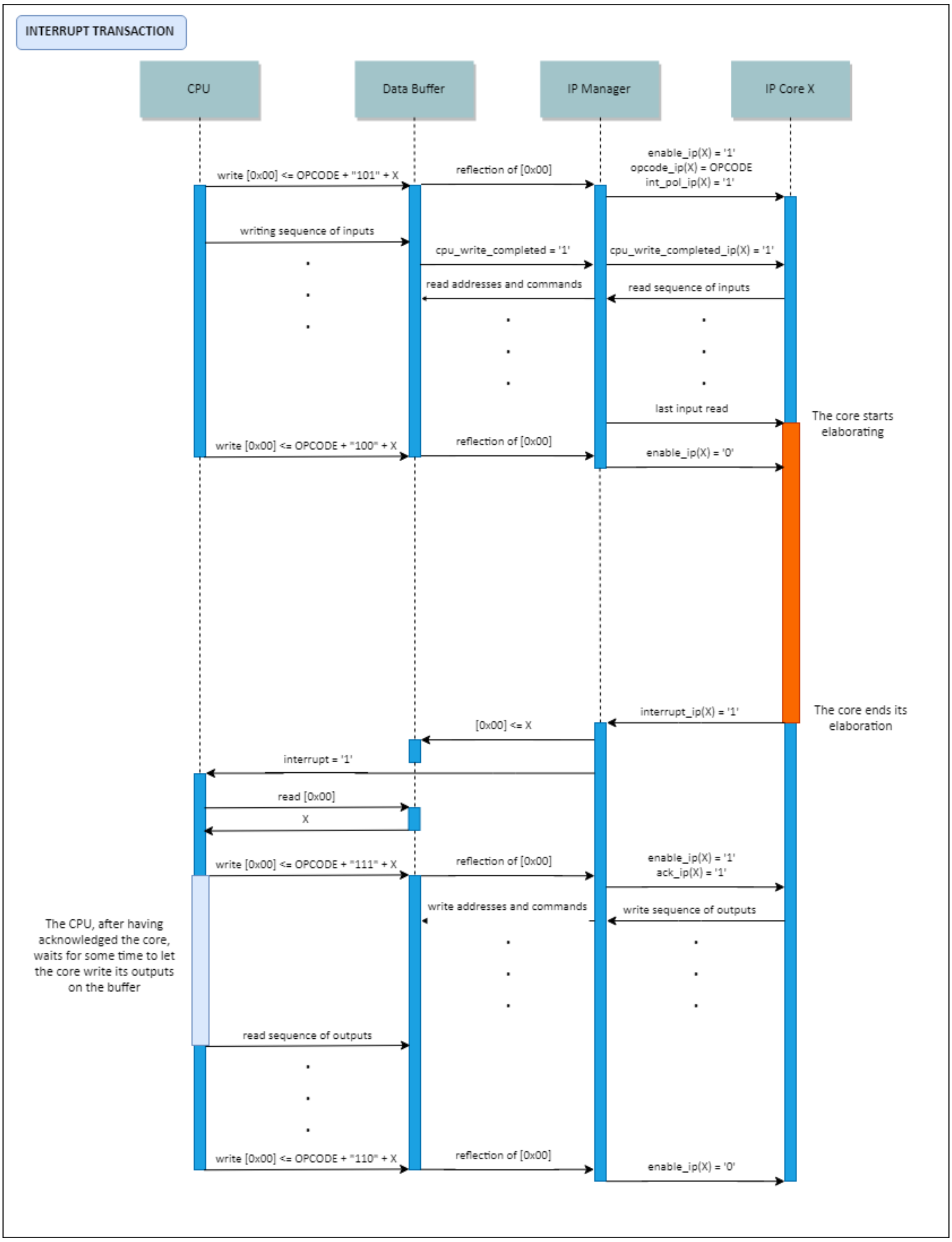
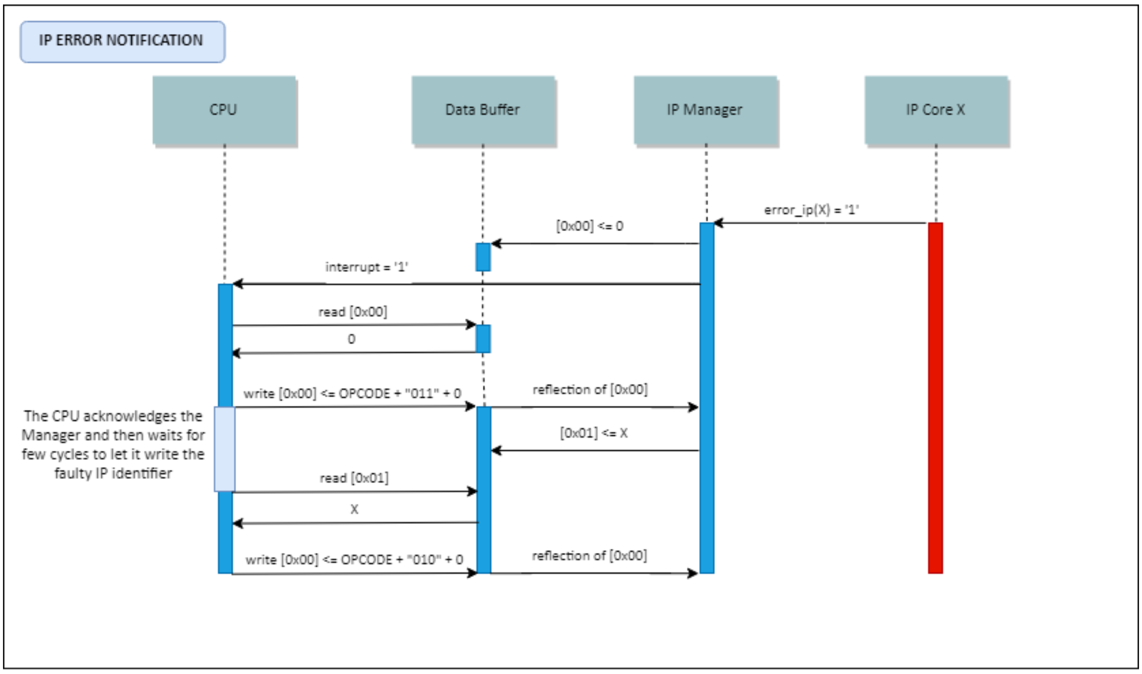

INTERRUPT TRANSACTION
CPU
Data Buffer
IP Manager
IP Core X
write [0x00] <= OPCODE + "101" + X
reflection of [0x00]
enable_ip(X) = '1'
opcode_ip(X) = OPCODE
int_pol_ip(X) = '1'
writing sequence of inputs
cpu_write_completed = '1'
cpu_write_completed_ip(X) = '1'
read addresses and commands
read sequence of inputs
last input read
The core starts elaborating
write [0x00] <= OPCODE + "100" + X
reflection of [0x00]
enable_ip(X) = '0'
interrupt_ip(X) = '1'
The core ends its elaboration
[0x00] <= X
interrupt = '1'
read [0x00]
X
write [0x00] <= OPCODE + "111" + X
reflection of [0x00]
enable_ip(X) = '1'
ack_ip(X) = '1'
write addresses and commands
write sequence of outputs
The CPU, after having acknowledged the core, waits for some time to let the core write its outputs on the buffer
read sequence of outputs
write [0x00] <= OPCODE + "110" + X
reflection of [0x00]
enable_ip(X) = '0'

Figure 14: Interrupt transaction between the CPU and a core



The CPU acknowledges the Manager and then waits for few cycles to let it write the faulty IP identifier

Figure 15: An IP core signals an internal error

4 Driver

This chapter describes the communication driver developed for interfacing the FPGA and the IP cores within it.
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, as opening and closing the transaction, or reading and writing a word on the buffer, but also managing the mutual concurrency of several high-level drivers.
Our work focused on providing the driver programmer a reliable low-level layer over which it is possible to implement a functionality layer to interact with the cores.

4.1 Low-level APIs

The low-level communication with the FPGA is handled by the functions declared in the header file “Fpgaipm.h”. Such functions are supposed to be called by the high-level driver code, without having almost any knowledge of the specific hardware implementations and of the details about the microcontroller and the FPGA provided by **SE***cube*™ .

```
FPGA_IPM_BOOLEAN FPGA_IPM_init(void)
```

This function is used to initialize all the data structures needed by the driver and to configure the required components of the **SE***cube*™ microcontroller. It is mandatory to call it just once at the beginning of any application code that makes use of the FPGA.
In details, the function:

- configures the FMC peripheral control registers to make it handle accesses to a block of external asynchronous PSRAM with data width of 16 bits. Address and data setup times are set as 6 CPU clock cycles long each, both for reading and for writing
- attaches the GPIO to the FMC address, control and data signals
- configures the clock input of the FPGA at 60 MHz (1/3 of the CPU clock)
- insert in the interrupt vector table the interrupt line coming from the FPGA
- initializes a semaphore to resolve concurrency issues between transactions.

Chapter 6 will discuss about the reasons of choices made on setup times and operating frequency.

```
FPGA_IPM_BOOLEAN FPGA_IPM_open(FPGA_IPM_CORE coreID,
   FPGA_IPM_OPCODE opcode, FPGA_IPM_BOOLEAN interruptMode,
   FPGA_IPM_BOOLEAN ack)
```

This function allows to open a transaction with a core on the FPGA, writing at address 0x00 of the buffer the control word with the specified parameters.
The following parameters are required:

- `coreID`: the ID of the required core that will be used. The ID 0 refers to the IP Manager itself
- `opcode`: the possible operative code. Depends on the used core
- `interruptMode`: a boolean indicating the type of transaction: polling (0) or interrupt (1)

- ack: a boolean that is positive when a transaction is answering to the interrupt request from a core.

Opening a 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 0x00 of the buffer
5. Send a positive response to the calling function if everything went fine.

```
FPGA_IPM_BOOLEAN FPGA_IPM_read(FPGA_IPM_CORE coreID,
   FPGA_IPM_ADDRESS address, FPGA_IPM_DATA *dataPtr)
```

This function is used to read a word from the FPGA buffer.
The following parameters are required:

- coreID: this parameter permits to check if the request is compliant with the current transaction (i.e., the active core ID is the same). If there are no transactions opened, the function returns a negative value
- address: the row of the data buffer from which data is read. The address must be in the range from 0x01 (1) to 0x3F (63)
- dataPtr: the main memory pointer where the read word is saved.

The function returns a boolean that notifies if the operation was correctly performed.
This procedure is implemented using internally the native function HAL_SRAM_Read_16b(), which is declared in the library file "stm32f4xx_hal_sram.h". The function only reads a piece of SRAM organized in 2-byte words. The automatic interaction with the FMC is possible thank to the FMC initial settings.

```
FPGA_IPM_BOOLEAN FPGA_IPM_write(FPGA_IPM_CORE coreID,
   FPGA_IPM_ADDRESS address, FPGA_IPM_DATA *dataPtr)
```

This function is used to write a word into the FPGA buffer.
The required parameters are:

- coreID: as the previous function, this parameter is used to check whether the request can be performed in base of the presence of an open transaction or not
- address: the memory offset of the FPGA data buffer in which the word is intended to be stored. Must be in the range from 0x01 (1) to 0x3F (63)
- dataPtr: the main memory pointer that contains the word that is going to be stored in the FPGA data buffer.

The function returns a value that notifies whether the operation was correctly performed or not. The procedure is implemented using HAL_SRAM_Write_16b() function declared in the library file "stm32f4xx_hal_sram.h" that interacts with FMC to send data to the FPGA.

```
FPGA_IPM_BOOLEAN FPGA_IPM_close(FPGA_IPM_CORE coreID)
```

This function is used to close a transaction with the FPGA by just complementing the B_E bit in the control word written at address 0x00. The only parameter that has to be specified is the corresponding IP core ID, that is used to verify if the closing request is compliant with the previous actions.
Closing a transaction releases the FPGA resources for the transactions.

4.2 FPGA interrupt handler

As presented, `FPGA_IPM_init()` also inserts in the interrupt sensitivity list of the processor the pin PA9, which is attached to the global interrupt line of the FPGA. The function `void EXTI9_5_IRQHandler(void)` present in "Fpgaipm.c" is the interrupt service routine automatically called when the FPGA interrupts the CPU. Once entered, the CPU effectively controls if the interrupt flag is set, then performs its operations and finally clears such flag. The body of this routine is to be customized depending on the cores present in the FPGA, this will be discussed in Chapter 6.

4.3 Concurrency issues

Concurrency is a big concern that may affect the correct behaviour 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:

- `FPGA_IPM_init()`, that initializes it
- `FPGA_IPM_open()`, 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
- `FPGA_IPM_close()`, 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

This Chapter is intended to explain you how to build your own application to be run on **SE***cube*™ DevKit exploiting the communication between the CPU and the possible FPGA hardware accelerators with a step-by-step description of the required actions. A complete project involves the customization of both sides: by the microcontroller side, an opportune firmware must be written, that makes use of a combination of the STMicroelectronics™ standard APIs and of the dedicate APIs for interacting with the customized hardware; by the FPGA side, the HDL description of the IP cores must be deployed and synthetized, paying attention to the respect of the protocol and of the physical interface restrictions already described.

5.1 The SE*cube*™ System Setup

In this Section, we outline the set of both hardware and software resources you need to set up the **SE***cube*™ DevKit.
At the end of this Section, you will have acquired a clear overview of the prerequisites to set up the environment.

5.1.1 Hardware resources

The following hardware resources are needed (detailed in the following paragraphs):

1. A PC
2. The **SE***cube*™ Open SDK
3. The **SE***cube*™ DevKit

You do not need a particularly new or powerful PC to get started with the **SE***cube*™ DevKit. Minimal requirements include:

- 2+ GiB[3] of RAM
- 10+ GiB of empty/available space on HDD
- USB ports

To program the STM32F429 processor available on the **SE***cube*™ DevKit you can follow two alternatives, resorting to:

- an in-circuit programmer and debugger, and particularly to the ST-Link/v2[4],
- one board such as the STM Discovery or STM Nucleo, equipped with a ST-Link/v2 programmer, respectively.

The ST-LINK/V2 is an in-circuit debugger and programmer for the STM8 and STM32 microcontroller families. Its JTAG/serial wire debugging-programming (SWD) interface is used to communicate with the STM32 microcontroller comprised within the **SE***cube*™ DevKit. This programmer requires 5V power supplied by a standard USB connector (A to Mini-B cable) compatible with the USB 2.0 interface. We suggest getting the programmer through RS[5], at a price of 19.19 €. Your purchase should comprise the following items (Figure 16):

- The St-Link/v2 programmer

[3] For the purpose of this document 1 GiB = 2^{30} Bytes
[4] http://www.st.com/en/development-tools/st-link-v2.html
[5] http://it.rs-online.com/web/p/kit-di-sviluppo-per-processori-e-microcontrollori/7141701/?sra=pmpn

- USB 2.0 A to Mini-B cable
- JTAG to SWD cable
- SWIM cable (not needed to program the **SE***cube*™ DevKit)

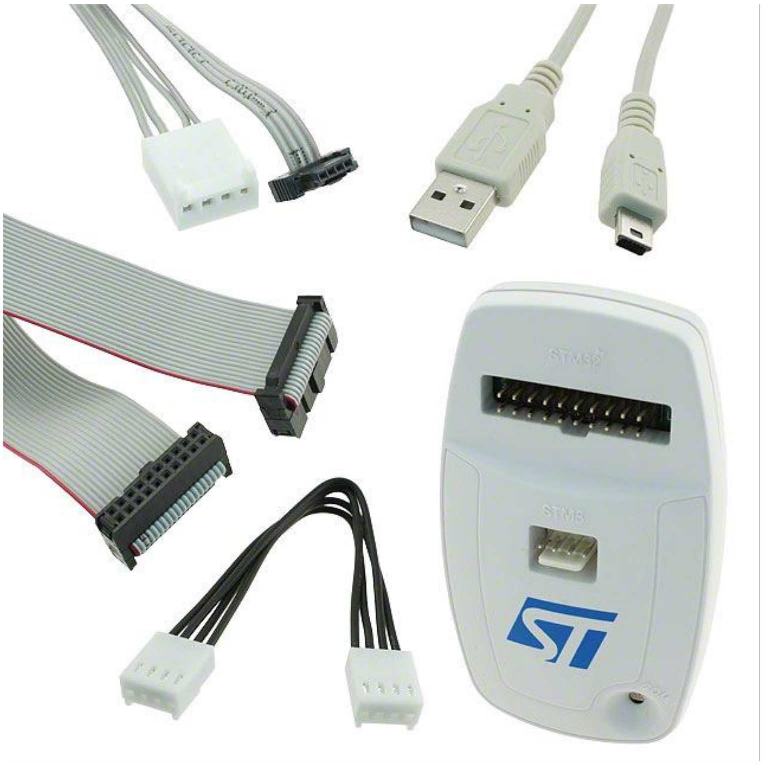


Figure 16: Components purchased with the ST-Link/v2 programmer

The ST Discovery and ST Nucleo boards represent an affordable and flexible way for users to build project with a microcontroller from the STM32 family, choosing from the various combinations of performance, power consumption and features.
These boards do not require any separate probe as they both integrate a ST-Link/V2 programmer/debugger.
The STM32 Nucleo board comes with the STM32 comprehensive software HAL library together with various packaged software examples, as well as direct access to online resources. We suggest getting the boards through RS. It is important to clarify that you do not need to buy them both: you can buy only one board, and your purchase will in any case represent a valid alternative to the ST-Link/v2 programmer.
The recommended Discovery[6] and Nucleo[7] boards can both be bought through RS. In both cases, you should get the board with a USB 2.0 A to Mini-B cable.
The **SE***cube*™ DevKit can be ordered online[8].
Your purchase should comprise the following items, depicted in Figure 17:

- The **SE***cube*™ DevKit;
- A USB 2.0 A to Micro-B cable

[6] http://it.rs-online.com/web/p/kit-di-sviluppo-per-processori-e-microcontrollori/9107951/
[7] http://it.rs-online.com/web/p/kit-di-sviluppo-per-processori-e-microcontrollori/8029425/
[8] http://www.secube.eu

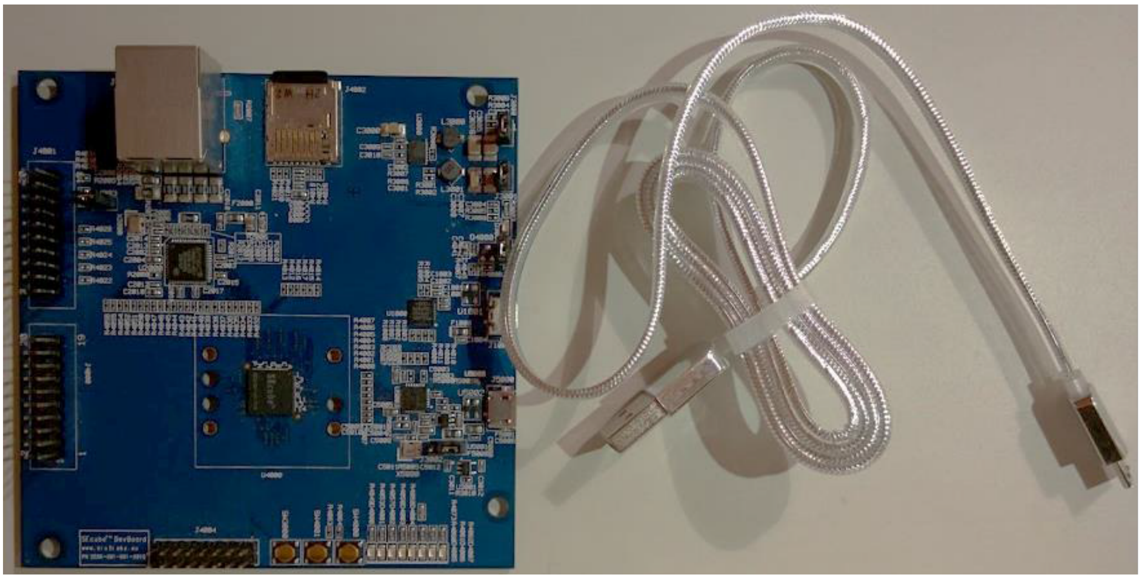


Figure 17: Components purchased with the **SE***cube*™ DevKit

In order to make the DevKit work properly, you should also purchase a MicroSD card with a minimum capacity of 4 GiB. The card must then be inserted in the dedicate socket (J4002).

5.1.2 Software resources

You need the following tools:

1. Operating System
2. Java Runtime Environment
3. Eclipse
4. AC6 Tools: GNU ARM Embedded Toolchain
5. STM32CubeMX - STM32Cube initialization code generator
6. Lattice Diamond Software
7. ST-Link/v2 drivers
8. ST-Link Utility
9. Open Source SDK

Two **Operating Systems** are currently supported:

- Windows 7 (or later)[9]
- Linux with Kernel 2.6 (or later)[10]

[9] This procedure has been tested with Windows 7 Professional x64

[10] This procedure has been tested with both Linux Chakra kernel 4.5.7-1 x64 and Linux Ubuntu 14.04LTS x64

The **Java Runtime Environment (JRE)** is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library. The Oracle Corporation, which owns the Java trade-mark, distributes a Java Runtime environment with their Java Virtual Machine called HotSpot.

Version Required

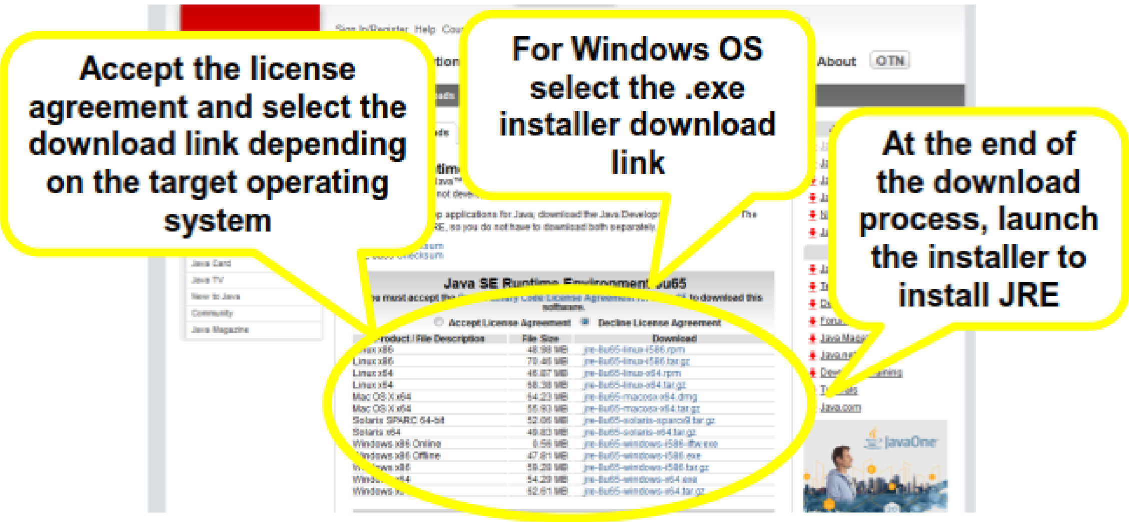
Version 8u111 (or later)

How to get it

The program is available free of charge from the Oracle website[11].

Installation hints

Visit the download link and follow the instructions as in the following screenshot:



Accept the license agreement and select the download link depending on the target operating system

For Windows OS select the .exe installer download link

At the end of the download process, launch the installer to install JRE

Java SE Runtime Environment 8u65

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ● Decline License Agreement

Product / File Description	File Size	Download
Linux x86	48.98 MB	jre-8u65-linux-i586.rpm
Linux x86	70.46 MB	jre-8u65-linux-i586.tar.gz
Linux x64	46.87 MB	jre-8u65-linux-x64.rpm
Linux x64	68.38 MB	jre-8u65-linux-x64.tar.gz
Mac OS X x64	64.23 MB	jre-8u65-macosx-x64.dmg
Mac OS X x64	55.93 MB	jre-8u65-macosx-x64.tar.gz
Solaris SPARC 64-bit	52.06 MB	jre-8u65-solaris-sparcv9.tar.gz
Solaris x64	49.83 MB	jre-8u65-solaris-x64.tar.gz
Windows x86 Online	0.56 MB	jre-8u65-windows-i586-iftw.exe
Windows x86 Offline	47.81 MB	jre-8u65-windows-i586.exe
Windows x86	59.28 MB	jre-8u65-windows-i586.tar.gz
Windows x64	54.29 MB	jre-8u65-windows-x64.exe
Windows x64	62.61 MB	jre-8u65-windows-x64.tar.gz

What is going to be used for

The Java Runtime Environment is required for Eclipse to work properly.

Eclipse is of the most widely used free and open-source integrated development environment (IDE) in computer programming.

It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may be used to develop applications in other programming languages as well, resorting to dedicated plugins.

Version required

Version 4.6 Neon (or later)

How to get it

You need to download the Eclipse IDE for C/C++ Developers[12].

Installation hints

Visit the download link and follow the indications of the website to download the correct version. Pay attention to choose the same architecture (32-bit or 64-bit) for both Eclipse and the Java Virtual Machine in your PC. You can verify which version of Java is present in your machine by launching the command “java -version” in a Command Prompt: its outcome would clearly state if the Java version within your PC is a 64-bit architecture (otherwise you should assume that it is a 32-bit architecture).

If the two architectures do not match, it is possible that Eclipse will show this error on startup: “Can’t start Eclipse - Java was started but returned exit code=13”.

[11] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[12] https://www.eclipse.org/downloads/

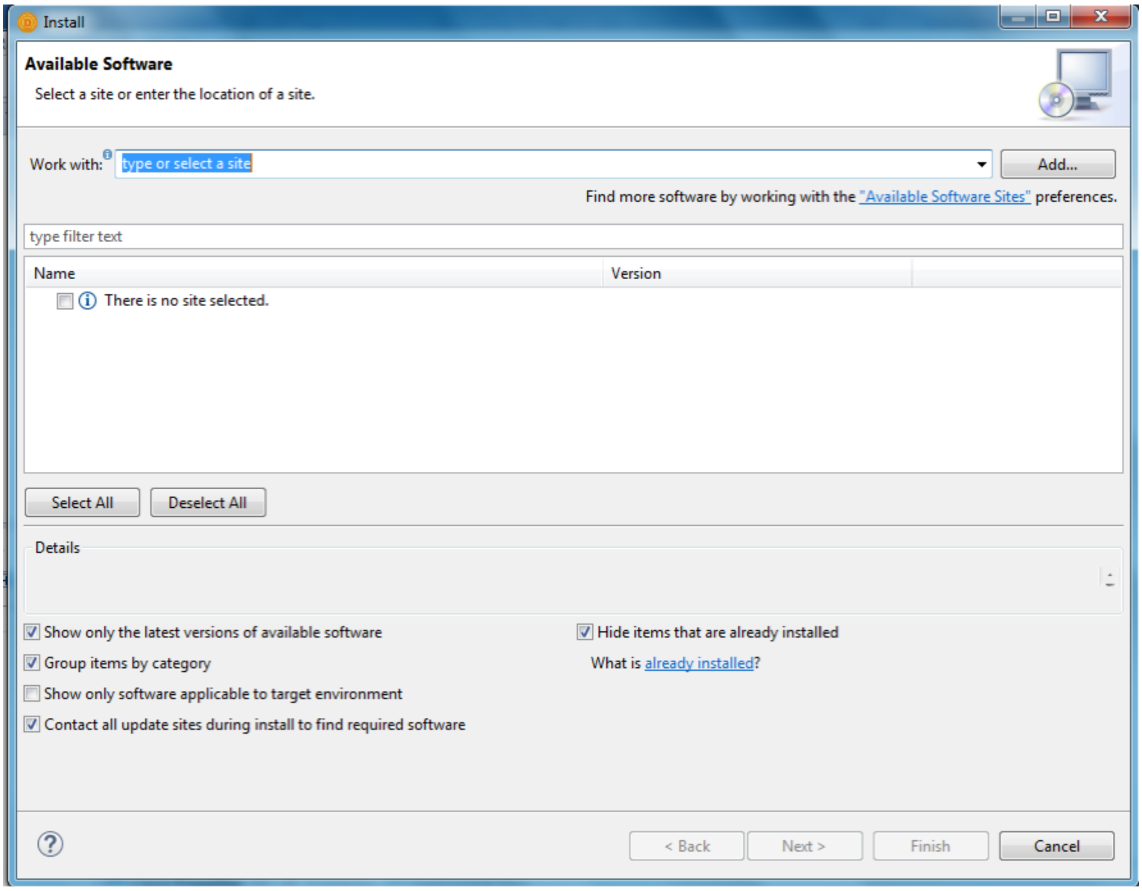
What it's going to be used for
Eclipse is the recommended IDE to develop applications that will run on the STM32F429 processor of the **SE***cube*™ DevKit.

The **AC6 Tool** will install the GNU Embedded Toolchain for ARM, which is a ready-to-use, open source suite of tools for C, C++ and Assembly programming targeting ARM Cortex-M and Cortex-R family of processors. It includes the GNU Compiler (GCC) and is available free of charge directly from ARM for embedded software development on both Windows and Linux operating systems. The reference platform for this document is the System Workbench for STM32 (SW4STM32) Eclipse plugin.
SW4STM32 is an integrated environment that includes:

- Building tools (GCC-based ARM cross compiler, assembler and linker);
- OpenOCD and GDB debugging tools;
- Flash programming tools

Version required
Version 5.0 (or later).
How to get it
To install SW4STM32 as an Eclipse plugin:

1. launch Eclipse IDE
2. on the toolbar, click «Help » Install New Software...»
3. in the Available Software window, click «Add»


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site [Add...]

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
□ ⓘ There is no site selected.	

[Select All] [Deselect All]

Details

☑ Show only the latest versions of available software
☑ Group items by category
□ Show only software applicable to target environment
☑ Contact all update sites during install to find required software
☑ Hide items that are already installed
What is already installed?

[< Back] [Next >] [Finish] [Cancel]

4. in the Add Repository window, set Name and Location fields as follows, and then click «OK»
   - Name: System Workbench for STM32 - Bare Machine edition
   - Location: http://www.ac6-tools.com/Eclipse-updates/org.openstm32.system-workbench.site
5. select OpenSTM32 Tools and click «Next»
6. accept the license agreement and click «Finish» to start the plugin installation, continue the installation also if a warning for incompatible or unsigned components is prompted
7. restart Eclipse

What it's going to be used for
The toolchain will be used to create, build, debug and in general to manage project that will be executed from the STM32 microcontroller comprised within the **SE***cube*™ DevKit.

STM32CubeMX is a graphical software configuration tool that allows generating C initialization code using graphical wizards. It also embeds a comprehensive software platform, delivered per series. This platform includes the STM32Cube HAL (an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio), plus a consistent set of middleware components (RTOS, USB, TCP/IP and graphics). All embedded software utilities come with a full set of examples.
STM32CubeMX is an extension of the existing MicroXplorer tool. It is a graphical tool that allows configuring STM32 microcontrollers very easily and generating the corresponding initialization C code through a step-by-step process.
The reference platform for this document is the STM32CubeMX Eclipse plugin.
Version required
Version 4.0 (or later)
How to get it
The software is downloadable free of charge online[13].
After having registered to the website, it will be possible to download a .zip file containing the STM32CubeMX Eclipse plugin; to install it then follow these steps:

1. launch Eclipse IDE
2. on the toolbar, click «Help » Install New Software...»
3. in the Available Software window, click «Add»
4. in the Add Repository window click on «Archive», select the downloaded ZIP file, and click «OK»
5. check the box corresponding to STM32CubeMX plugin and click «Next»
6. accept the license agreement to install the plugin, continue the installation also if a warning for incompatible or unsigned components is prompted
7. restart Eclipse

[13] http://www.st.com/en/development-tools/stsw-stm32095.html. As an alternative (not recommended), it is possible to install the software as a standalone by downloading and extracting the .zip file downloadable from http://www.st.com/en/development-tools/stm32cubemx.html. If you work under Windows, you can execute directly the .exe executable; if you work under Linux, you have the launch the following command from the command prompt "sudo java -jar filename.exe" (substituting "filename" with the actual file-name of the executable) and to insert your user password if required.

What it's going to be used for
STM32CubeMX eases system development providing:

- C code generation covering initialization code for standard toolchains
- Embedded software libraries and middleware components (e.g., Open-source TCP/IP stack, USB drivers, open-source FAT file system, open source RTOS) with related examples

Lattice Diamond® software is the leading-edge software design environment for cost-sensitive, low-power Lattice FPGA architectures. Lattice Diamond's integrated tool environment provides a modern, comprehensive user interface for controlling the Lattice Semiconductor FPGA implementation process.
Version required
Version 3.5 (or later)
How to get it
The software is downloadable free of charge online[14].
Installation hints
When downloading the software, it is possible to choose the free license.
What it's going to be used for
The software will be used for controlling the implementation process of the Lattice Semiconductor FPGA comprised within the **SE***cube*™ DevKit.

ST-Link v2 drivers provide support for the ST-Link/v2 programmer.
Version required
Version 4.0.0 (or later)
How to get it
The software is downloadable free of charge online[15].
Installation hints
During the installation procedure, it is possible to receive warnings from the Operating System if drivers are not properly signed; however, the installation procedure should not be interrupted.
What it's going to be used for
To allow the usage of the ST-Link/v2 programmer.

The **STM32 ST-LINK Utility** software facilitates fast in-system programming of the STM32 microcontroller families in development environments via the ST-LINK and ST-LINK/V2 tools.
Version required
Version 4.0.0 (or later)
How to get it
The software is downloadable free of charge online for Windows users[16].
Linux user, instead, can resort to the Open On-Chip Debugger (OpenOCD); this software is downloadable free of charge online[17].
What it's going to be used for
To speed up the usage of the ST-Link/v2 programmer.

A **Software Development Kit** (SDK or ”devkit”) is typically a set of software development tools that allows the creation of applications for a given system. To exploit all the functionalities of

[14] http://www.latticesemi.com/Products/DesignSoftwareAndIP/FPGAandLDS/LatticeDiamond
[15] http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link009.html
[16] http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link004.html
[17] https://sourceforge.net/projects/openocd/files/openocd/0.9.0/

your **SE***cube*™ DevKit, we provide a free and open-source SDK which are commented in this document.
Of relevance within this SDK is a configuration file ("SEcubeDevBoard.ioc") which stores the configuration of microprocessor integrated in the **SE***cube*™ DevKit.
How to get it
This file is available as part of the Open Source SDK which can be downloaded from the following link: https://www.secube.eu/resources/open-sources-sdk/. Once downloaded, extract the content into a known location, and keep extracting subarchives until you are able to browse to "SECubeSDK/SDK device side/code/optimized_apis_firmware_nonblockinglogin/secube_sdk/development" to find "SEcubeDevBoard.ioc".
What it's going to be used for
The configuration file is used to generate automatically software driver and or custom configurations tailored for the microprocessor integrated in the **SE***cube*™ DevKit.

5.1.3 Assembling the System

In this Section, we list the instructions you need to follow to properly connect the **SE***cube*™ DevKit to the Host PC and to Programmer/debugger, as shown in Figure 18.
At the end of this Section you will have acquired a clear overview of the procedures to follow to start using the environment.

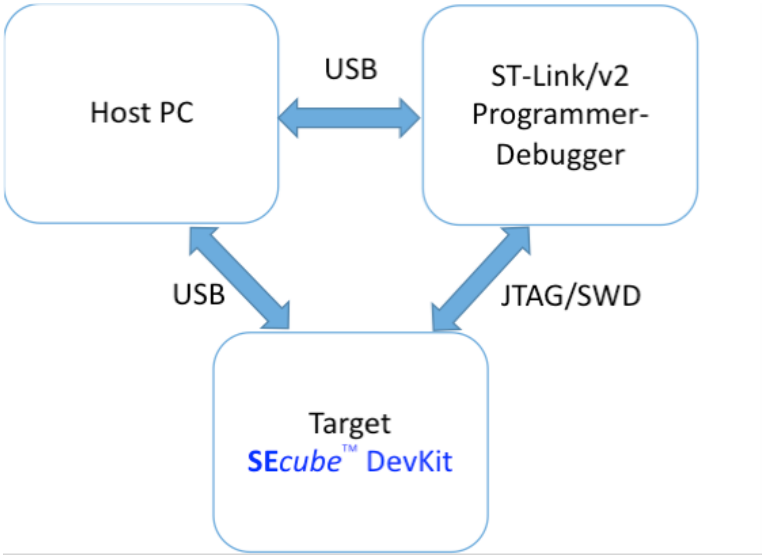


Figure 18: System Architecture

5.1.4 Assembling Steps

If you decide to use the ST-Link/v2 programmer, assembling is composed of the following two steps:

1. Connect the **SE***cube*™ DevKit with the programmer by means of the JTAG/SWD cable: the cable should be inserted on the JTAG docks on both the programmer (in this case the orientation of the plug is forced from the dock) and the DevKit (in this case you must pay attention in inserting the plug on top of both lines of connectors and with its protrusion oriented towards the inner side of the DevKit)

2. Connect the ST-Link/v2 with the PC by means of the USB cable

The system assembled is shown in Figure 19, while a close-up on the JTAG connection is in Figure 20.

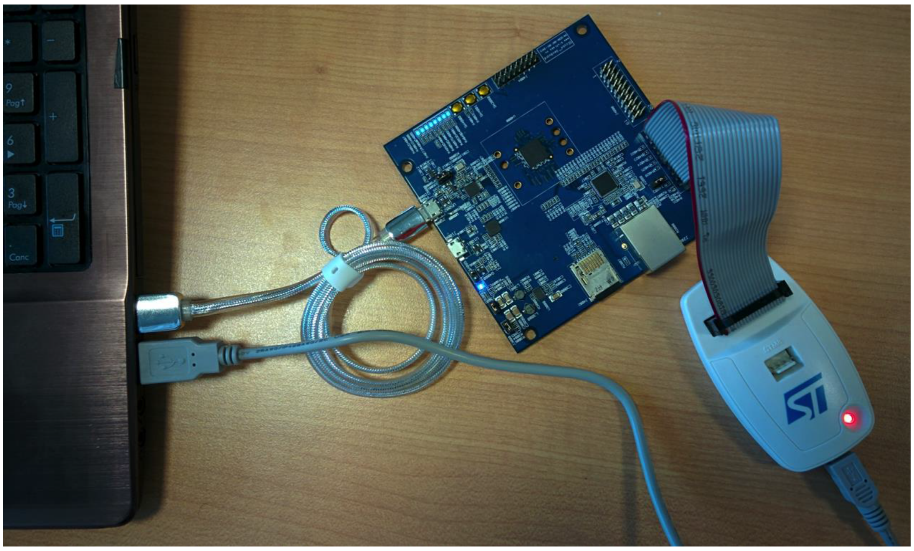


Figure 19: Connection between the STLink/v2 programmer and the **SE***cube*™ DevKit

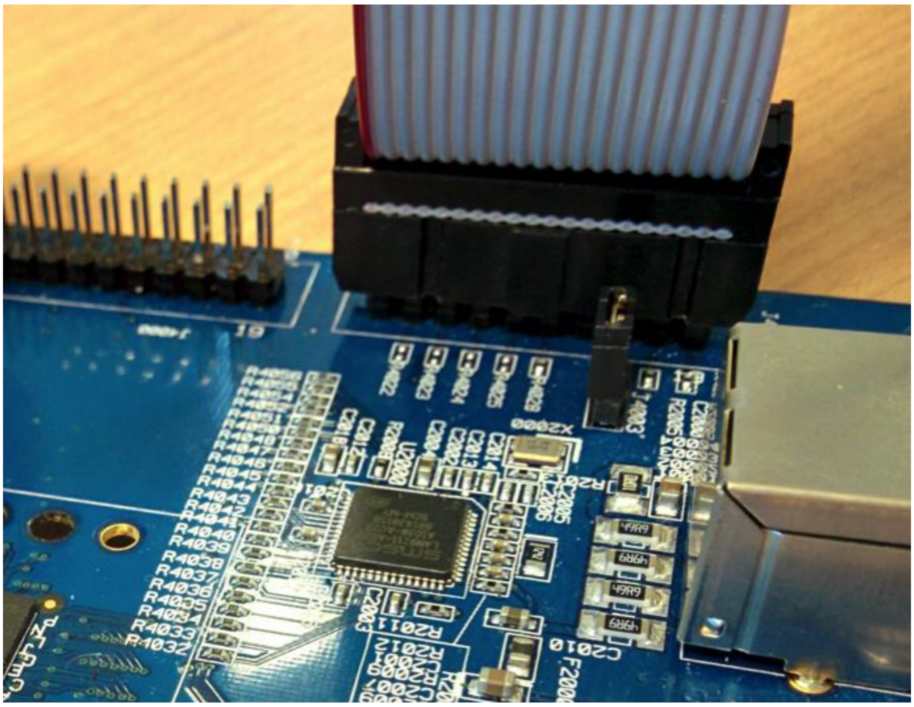


Figure 20: Connection between the STLink/v2 programmer and the **SE***cube*™ DevKit, close-up (highlighted in red) on the JTAG connector orientation

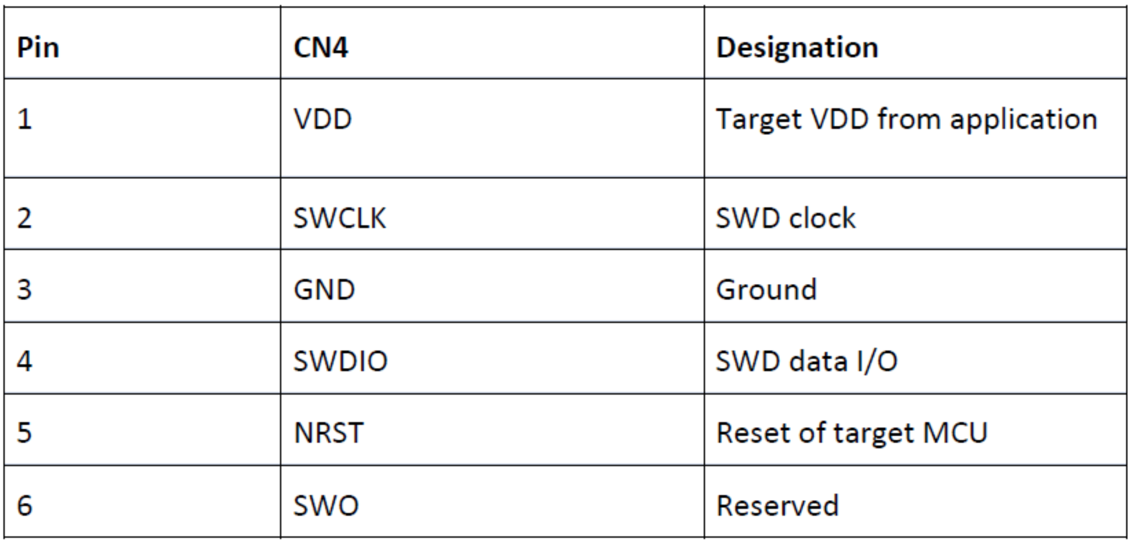
If you decide to use the ST-Link programmer comprised within a Discovery or Nucleo board, assembling requires the following three steps:

1. Isolate the programmer from the rest of the board by moving the jumpers to reach the configuration shown in Figure 22 and described in the STM32 Nucleo-64 boards user manual[18]

[18] Available at the following link, please refer to Section 6.2.4 of the User Manual of the STM32 Nucleo board: https://

2. Connect the **SE***cube*™ DevKit with the programmer by means of the JTAG/SWD cable: the cable should be inserted on the JTAG docks of the DevKit (you must pay attention in inserting the plug with its protrusion oriented towards the inner side of the DevKit) and on the SWD dock of the board, accordingly to the schema in Figure 21

3. Connect the ST-Link/v2 with the PC by means of the USB cable.



Pin	**CN4**	**Designation**
1	VDD	Target VDD from application
2	SWCLK	SWD clock
3	GND	Ground
4	SWDIO	SWD data I/O
5	NRST	Reset of target MCU
6	SWO	Reserved

Figure 21: Programmer SWD pins schema

www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf#page=18&zoom=100,0,652

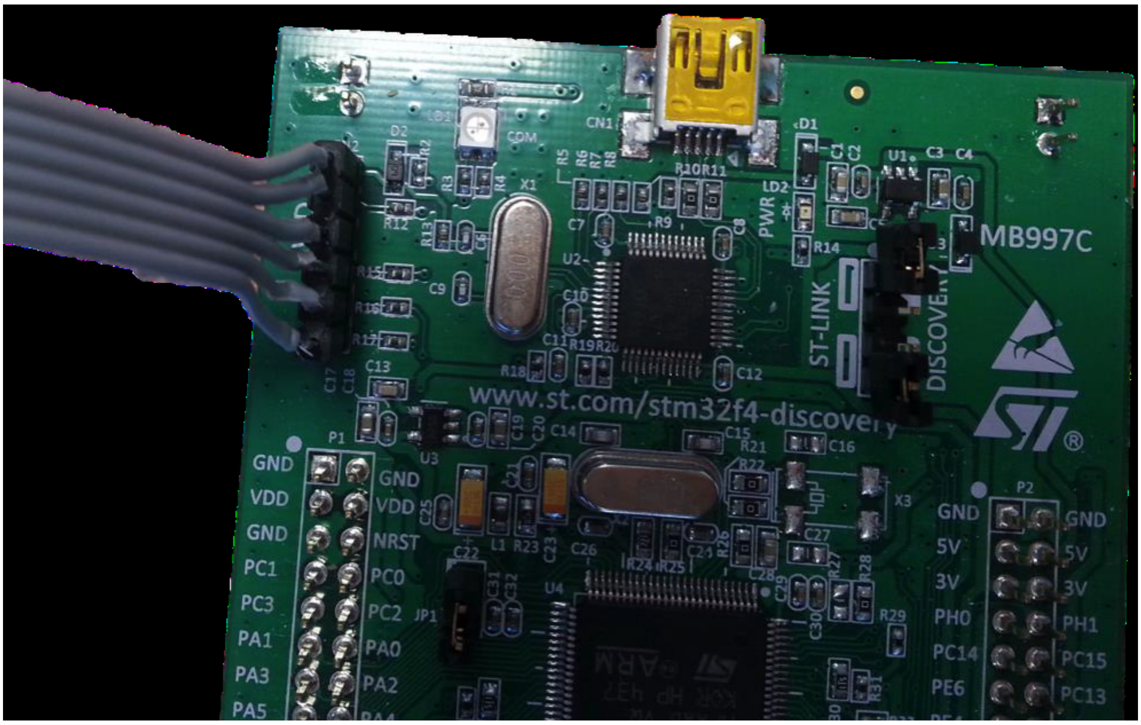


Figure 22: Jumpers configuration to isolate the ST-Link programmer on a Discovery board (highlighted in red, the same applies to Nucleo boards)

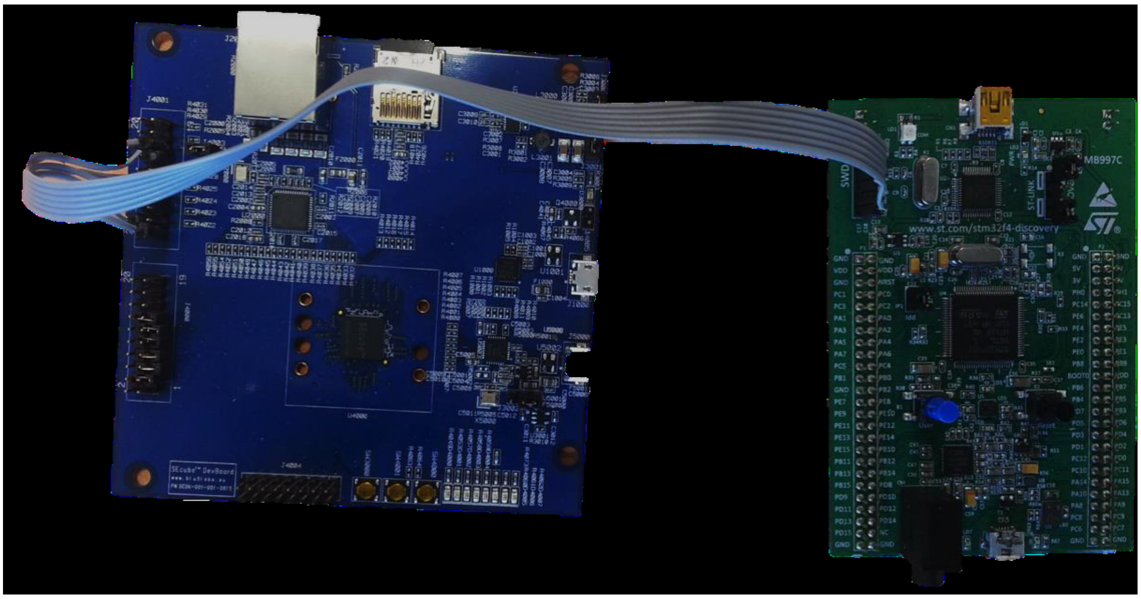


Figure 23: Connection between the Discovery board and the DevKit (the same applies to Nucleo boards)

5.1.5 What it should happen

After having properly connected the programmer through the USB interface its signaling LED should turn on; after having properly connected the DevKit through the USB interface all its LEDs should turn on.

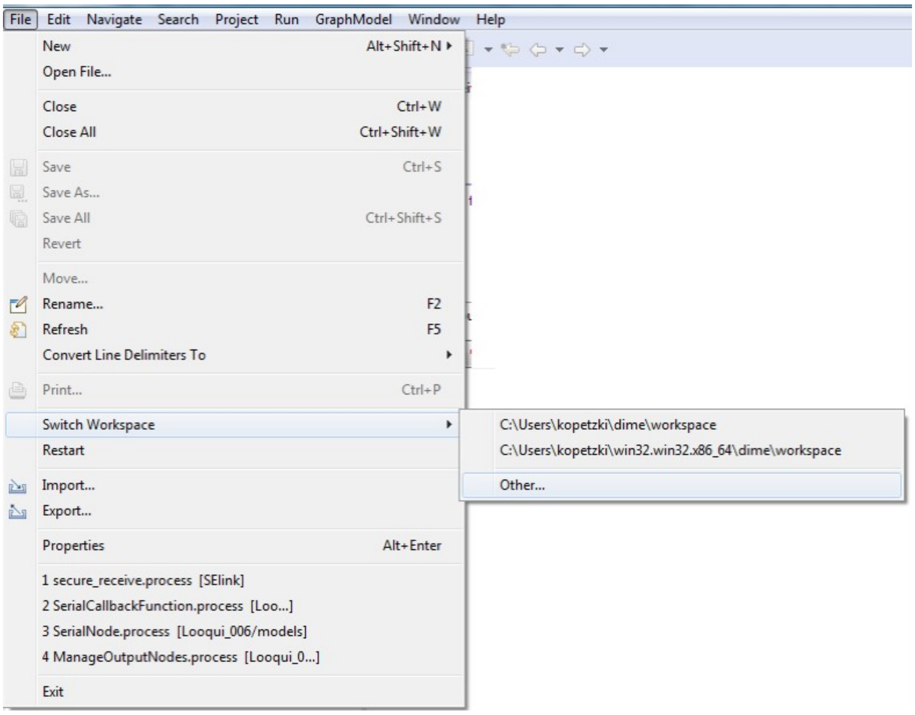
5.2 C/C++ Project

In this Section, we list the instructions you need to follow to import the **SE***cube*™ software libraries within an Eclipse project.
At the end of this Section you will have acquired a clear overview of the procedures to follow to import the libraries and use them to foster the development of your application.

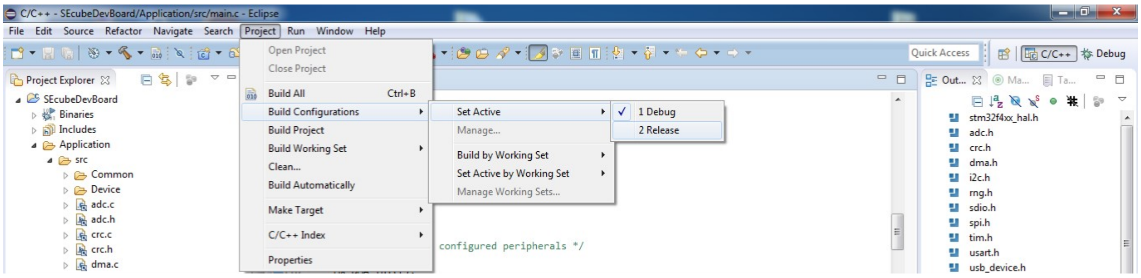
5.2.1 SE*cube*™ Open Source Software Libraries - Device Side

Hereby is listed a step-by-step guide to create the binaries files that will be executed on the **SE***cube*™ DevKit:

1. Download the SDK package containing the project from https://www.secube.eu/resources/open-sources-sdk/
2. Extract the .zip file to a known location
3. Open the folder ”SEcube SDK_v1.4_1” and extract the content of the archive ”SECubeSDK.tar.gz”
4. From your location, go to ”SECubeSDK/SDK device side/code” and extract the content of the archive ”optimized_apis_firmware_nonblockinglogin.zip”
5. Open the folder ”optimized_apis_firmware_nonblockinglogin” and extract the content of the archive ”secube_sdk.zip”
6. From your location, go to ”secube_sdk/development” and extract the content of the archive ”environment.zip”
7. Launch Eclipse
8. Change Eclipse perspective to «C/C++ selecting Window » Perspective » Open Perspective » Other... » C/C++»
9. Switch the Eclipse workspace to «File » Switch Workspace » Other...» and select the “ws” folder contained within the “environment” folder previously extracted from the .zip file



10. Set the Debug configuration from «Project » Build Configuration » Set Active»

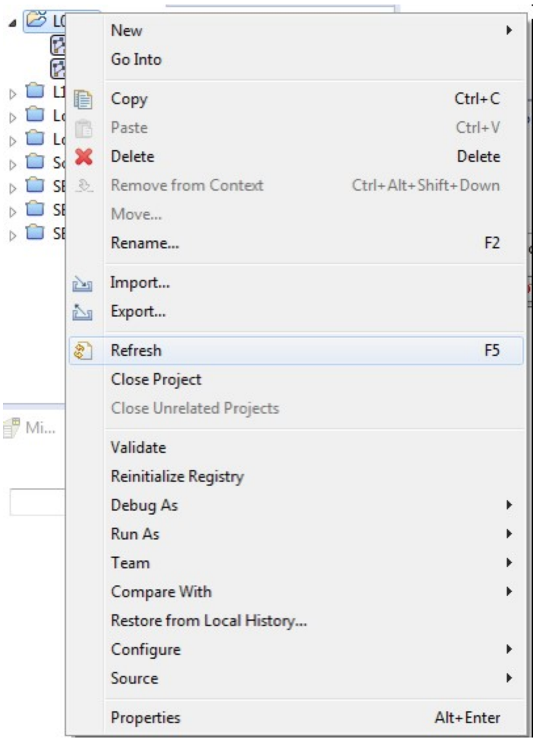


11. Build the project in Debug mode from «Project » Build All»

12. Set the Release configuration from «Project » Build Configuration » Set Active»

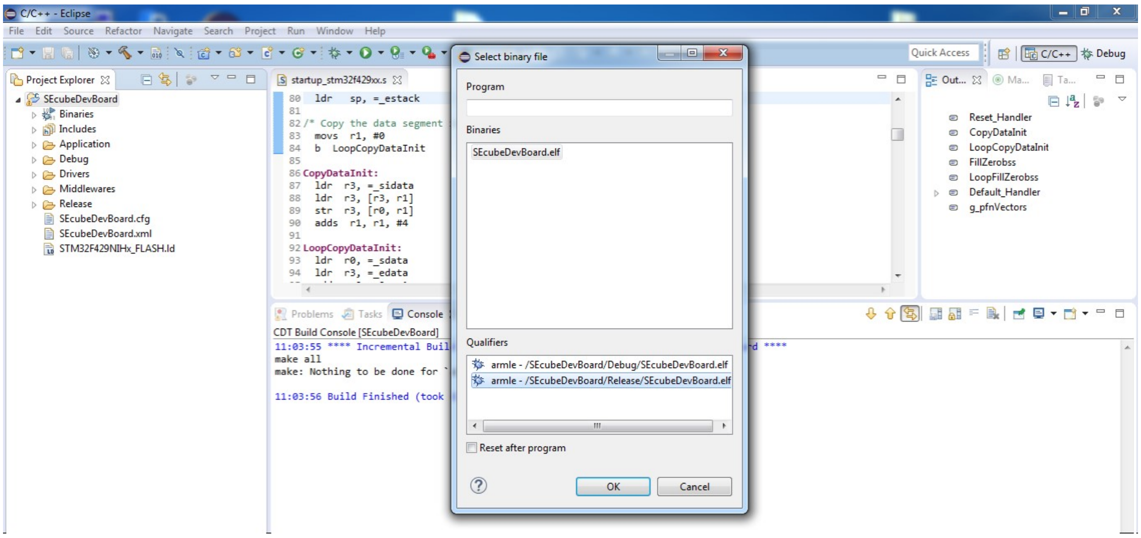
13. Build the project in Release mode from «Project » Build All»

14. Right-click on the project in the Project Explorer and select «Refresh»



15. Connect the DevKit as described in previous section

16. 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")



```
  ldr   sp, =_estack

/* Copy the data segment
  movs  r1, #0
  b  LoopCopyDataInit

CopyDataInit:
  ldr  r3, =_sidata
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4

LoopCopyDataInit:
  ldr  r0, =_sdata
  ldr  r3, =_edata
```

CDT Build Console [SEcubeDevBoard]

```
11:03:55 **** Incremental Buil                    d ****
make all
make: Nothing to be done for `

11:03:56 Build Finished (took
```

Select binary file

Program

Binaries

SEcubeDevBoard.elf

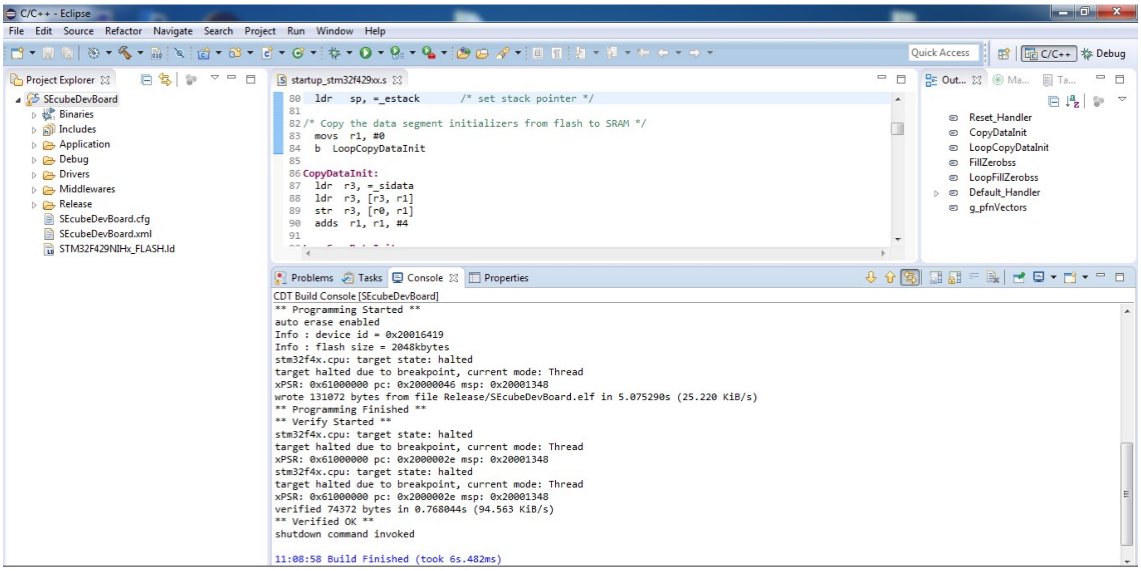
Qualifiers

armle - /SEcubeDevBoard/Debug/SEcubeDevBoard.elf
armle - /SEcubeDevBoard/Release/SEcubeDevBoard.elf

□ Reset after program

OK Cancel

17. Wait until the debugger shows the messages "Programming Finished" and "Verified OK"



```
  ldr   sp, =_estack        /* set stack pointer */

/* Copy the data segment initializers from flash to SRAM */
  movs  r1, #0
  b  LoopCopyDataInit

CopyDataInit:
  ldr  r3, =_sidata
  ldr  r3, [r3, r1]
  str  r3, [r0, r1]
  adds  r1, r1, #4

```

CDT Build Console [SEcubeDevBoard]

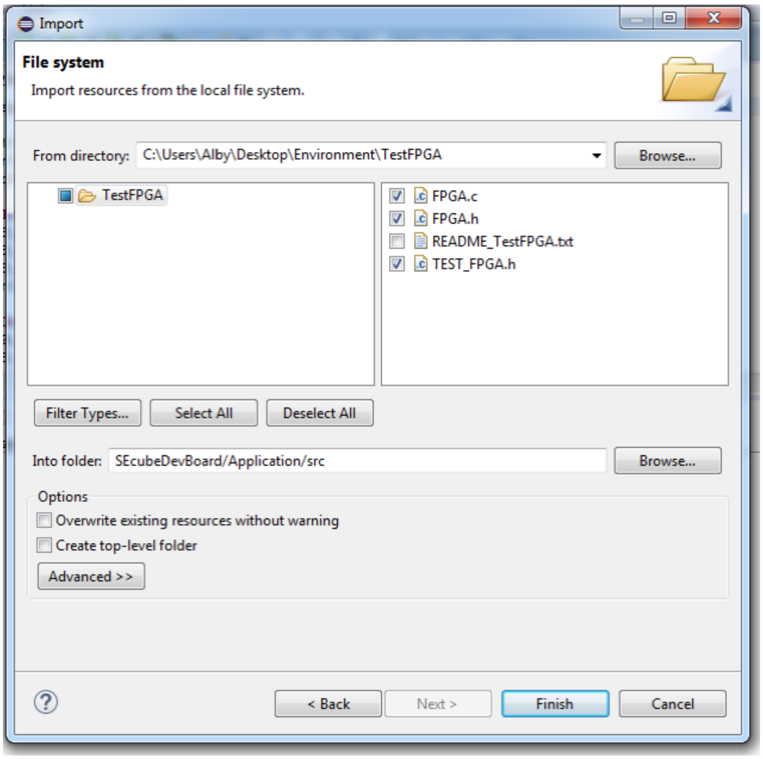
```
** Programming Started **
auto erase enabled
Info : device id = 0x20016419
Info : flash size = 2048kbytes
stm32f4x.cpu: target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000046 msp: 0x20001348
wrote 131072 bytes from file Release/SEcubeDevBoard.elf in 5.075290s (25.220 KiB/s)
** Programming Finished **
** Verify Started **
stm32f4x.cpu: target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000002e msp: 0x20001348
stm32f4x.cpu: target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000002e msp: 0x20001348
verified 74372 bytes in 0.768044s (94.563 KiB/s)
** Verified OK **
shutdown command invoked

11:08:58 Build Finished (took 6s.482ms)
```

Now your DevKit is fully configured and the STM32 microprocessor is ready to be used to develop your security applications.

5.2.2 Running your first program: FPGA_LED (device-side)

The procedure shown in this paragraph guides you to a first example of how to use the Open Source Libraries with the FPGA; it programs the FPGA embedded in the **SE***cube*™ chip to make the LEDs blink.
Hereby we list a step-by-step guide to run this program:

1. Import the project as described in section 5.2.1
2. Import the necessary «File » Import...», select “Filesystem” and press “Next”
3. Browse to the directory where the SDK has been downloaded and then to the path “SEcube_SDK/Libraries/Examples/TestFPGA/”
4. Select the files in that folder (FPGA.c, FPGA.h and TEST_FPGA.h); you might want to set also “Destination Folder” to “SEcubeDevBoard/Application/src” and then press “Finish”


Import

File system

Import resources from the local file system.

From directory: C:\Users\Alby\Desktop\Environment\TestFPGA

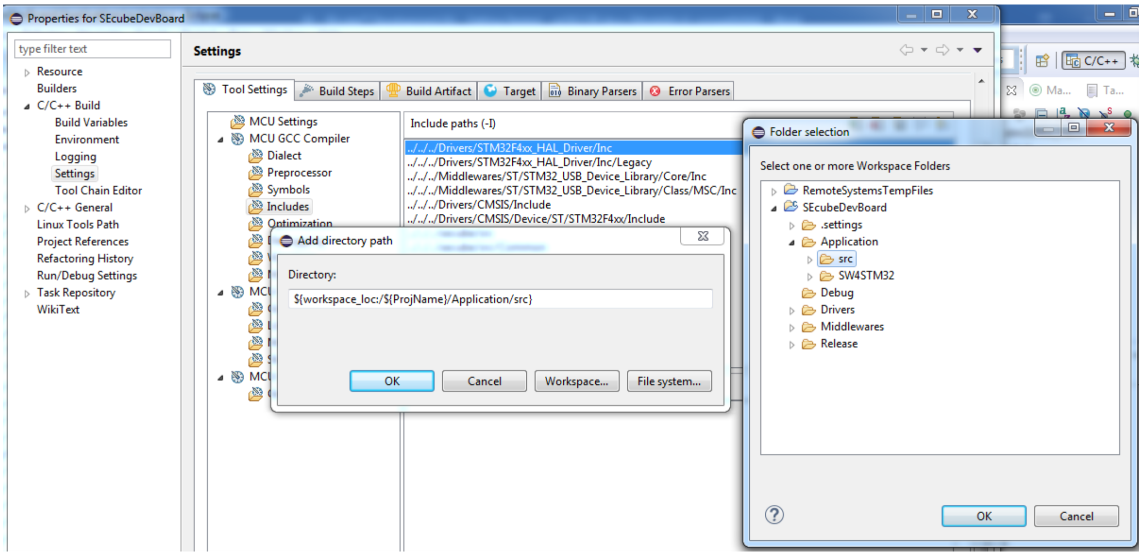
TestFPGA

☑ FPGA.c
☑ FPGA.h
☐ README_TestFPGA.txt
☑ TEST_FPGA.h

Into folder: SEcubeDevBoard/Application/src

Options
☐ Overwrite existing resources without warning
☐ Create top-level folder

5. Configure both “Debug” and “Release” configurations from «Project » Properties » C/C++ Build » MCU GCC Compiler » Includes» and add the “Destination folder”


Properties for SEcubeDevBoard

Include paths (-I)

../../../Drivers/STM32F4xx_HAL_Driver/Inc
../../../Drivers/STM32F4xx_HAL_Driver/Inc/Legacy
../../../Middlewares/ST/STM32_USB_Device_Library/Core/Inc
../../../Middlewares/ST/STM32_USB_Device_Library/Class/MSC/Inc
../../../Drivers/CMSIS/Include
../../../Drivers/CMSIS/Device/ST/STM32F4xx/Include

Add directory path

Directory: ${workspace_loc:/${ProjName}/Application/src}

Folder selection

Select one or more Workspace Folders

6. Now edit the code in “main.c” file including the header file “FPGA.h”

7. Also in “main.c” add a call to `B5_FPGA_Programming()` function

8. Open the file named “gpio.c” and add the following lines to the function `MX_GPIO_Init ()`, needed for configuring the JTAG port used for programming the FPGA:

```
/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = GPIO_PIN_2;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);
/*Configure GPIO pins : PE3 PE4 PE5 PE6 */
GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5|
   GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);
```

9. Save the changes to all files and build the project

10. Connect the DevKit as described in previous section

11. 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")

Notice that the after turned on, programming of the FPGA might require some time (1-2 minutes) to be completed. After that, you should see that the LEDs on the board start blinking.

5.2.3 How to import your own project

At this point, you should have understood how a CPU-FPGA communication project should be configured. You need the programming library for the FPGA composed by the three files included in the example before, you need to set the GPIOs of the JTAG port of the FPGA as indicated by the piece of code in the previous section, and you certainly need a call to the FPGA programming function in the `main()`. What is to be substituted are the two huge byte arrays present in the file "TEST_FPGA.h" with the bitstream containing the information about the pin interface and for programming internally the FPGA through the JTAG. Such file is generated automatically from your own HDL description of the FPGA by the Lattice Diamond® Deployment Tool after the synthesis steps. What you should do is nothing else than replace within the file these two arrays with the ones generated by this tool, but this will be explained in detail the next section.
Another important setup that you must do in order to interface the FPGA design described by this project is the import of our API library with the general managing class we created.

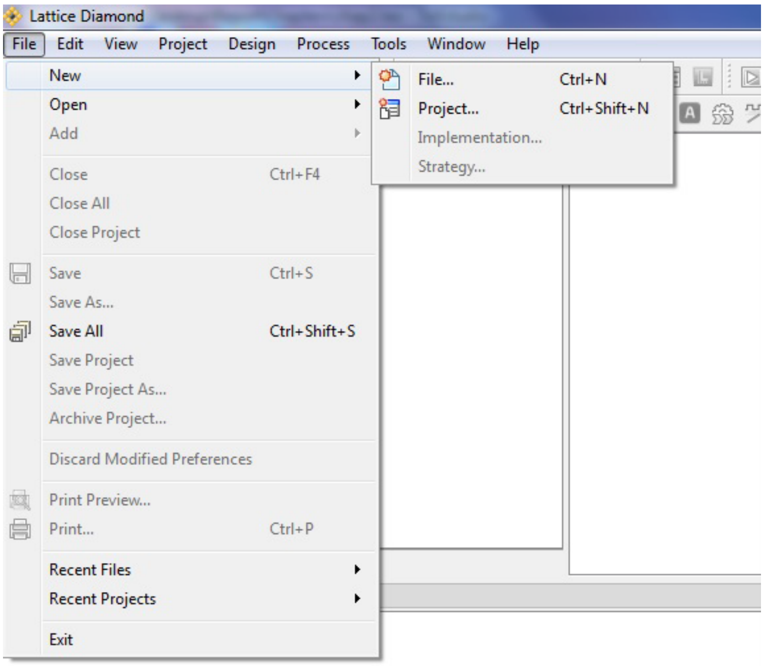
5.3 HDL project

In this Section, we list the instructions you need to follow to setup your own set of IP cores and to import them within the project.
First, you need to download the HDL source files for the data buffer, the IP Manager, the package containing the constants and the top-entity structural description of the FPGA available on the download section of the **SE***cube*™ website. For a project that works as specified in this document, these sources must be altered only in minimal part as it will be indicated, while all the rest of the code must be maintained untouched. The group of available files also comprehends an example core performing SHA256 algorithm and a relative generic testbench usable for simulation purposes, but this will be explained in the last Section.

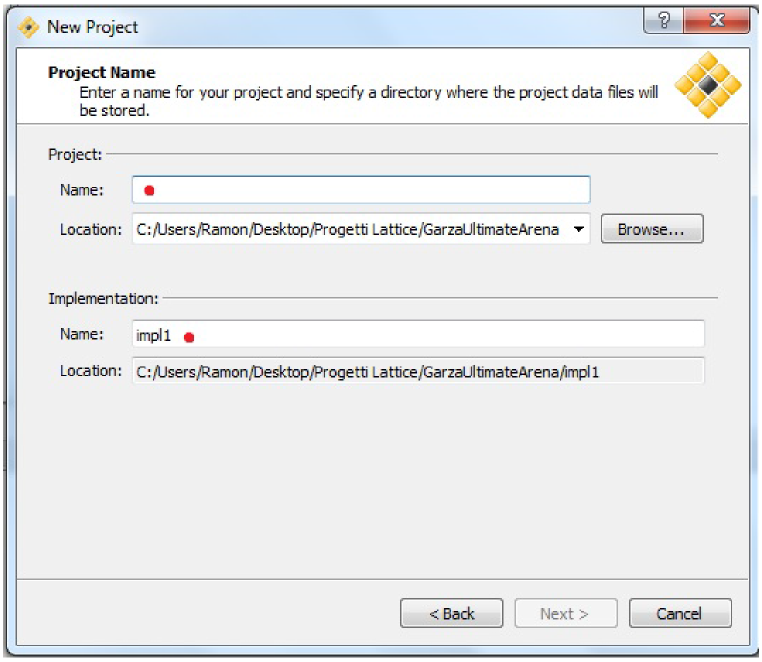
5.3.1 How to create a Lattice Project

Once your IP cores have been developed, tested and validated as working with whatever simulation tool you prefer, the synthesis process must begin. Open Lattice Diamond® and follow these steps.

1. Create a new project



2. Browse to the location of your HDL project folder and insert an implementation name



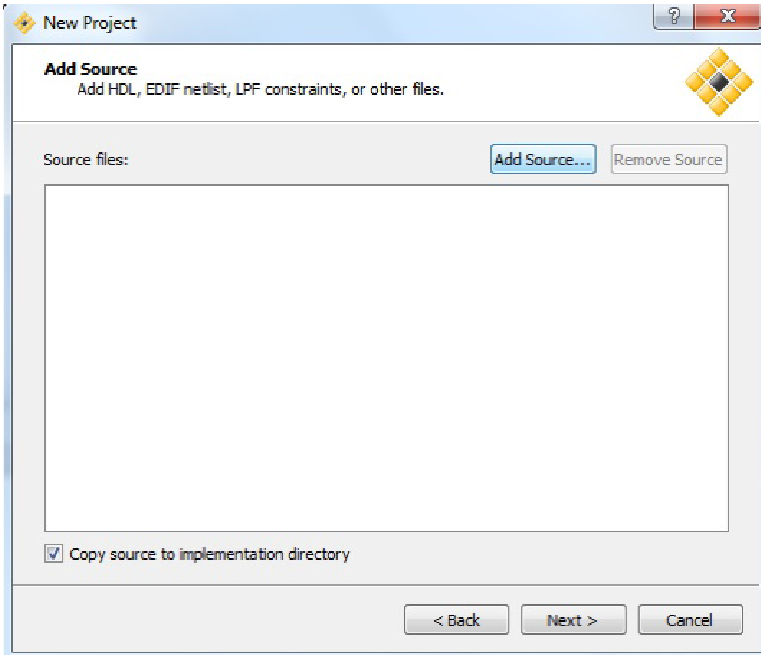
New Project

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project:
Name: ●
Location: C:/Users/Ramon/Desktop/Progetti Lattice/GarzaUltimateArena

Implementation:
Name: impl1 ●
Location: C:/Users/Ramon/Desktop/Progetti Lattice/GarzaUltimateArena/impl1

3. Add VHDL source files in this step or inside the project by right clicking on the input files folder. Remember that files named “CONSTANTS.vhd”, “DATA_BUFFER.vhd”, “IP_MANAGER.vhd”, “TOP_ENTITY.vhd” must be included, plus your own custom IP cores source files


New Project

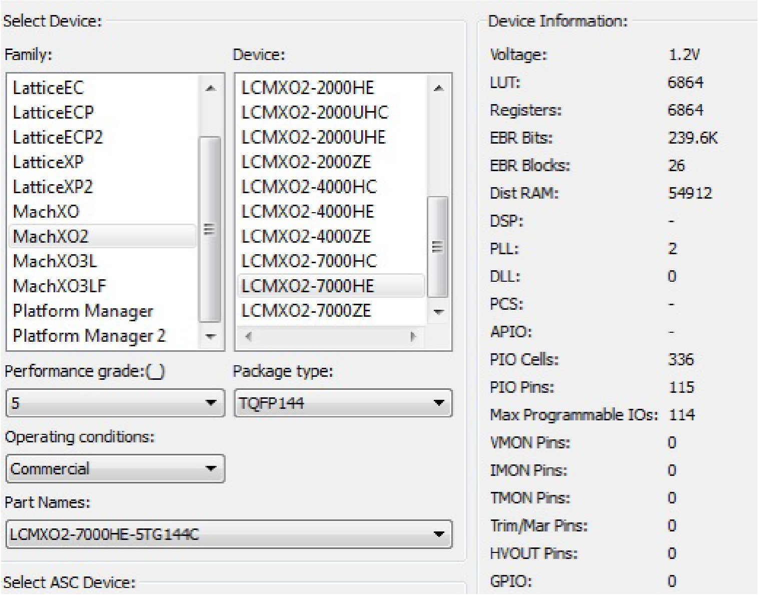
Add Source

Add HDL, EDIF netlist, LPF constraints, or other files.

Source files:

☑ Copy source to implementation directory

4. Select the correct FPGA model. In this case, the **SE*cube***™ FPGA correct version is MachXO2-LCMXO2-7000HE-5TG144C


Select Device:

Family: LatticeEC, LatticeECP, LatticeECP2, LatticeXP, LatticeXP2, MachXO, MachXO2, MachXO3L, MachXO3LF, Platform Manager, Platform Manager 2

Device: LCMXO2-2000HE, LCMXO2-2000UHC, LCMXO2-2000UHE, LCMXO2-2000ZE, LCMXO2-4000HC, LCMXO2-4000HE, LCMXO2-4000ZE, LCMXO2-7000HC, LCMXO2-7000HE, LCMXO2-7000ZE

Performance grade:(_): 5

Package type: TQFP144

Operating conditions: Commercial

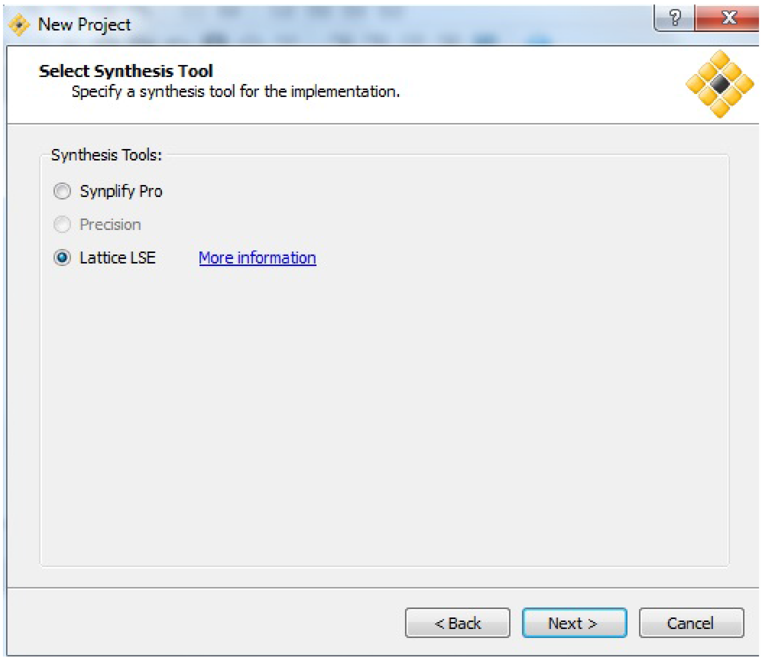
Part Names: LCMXO2-7000HE-5TG144C

Select ASC Device:

Device Information:

Voltage:	1.2V
LUT:	6864
Registers:	6864
EBR Bits:	239.6K
EBR Blocks:	26
Dist RAM:	54912
DSP:	-
PLL:	2
DLL:	0
PCS:	-
APIO:	-
PIO Cells:	336
PIO Pins:	115
Max Programmable IOs:	114
VMON Pins:	0
IMON Pins:	0
TMON Pins:	0
Trim/Mar Pins:	0
HVOUT Pins:	0
GPIO:	0

5. In the following window you have to select the Synthesis tool used. Select LATTICE LSE and click “Next”



Select Synthesis Tool

Specify a synthesis tool for the implementation.

Synthesis Tools:

○ Synplify Pro

○ Precision

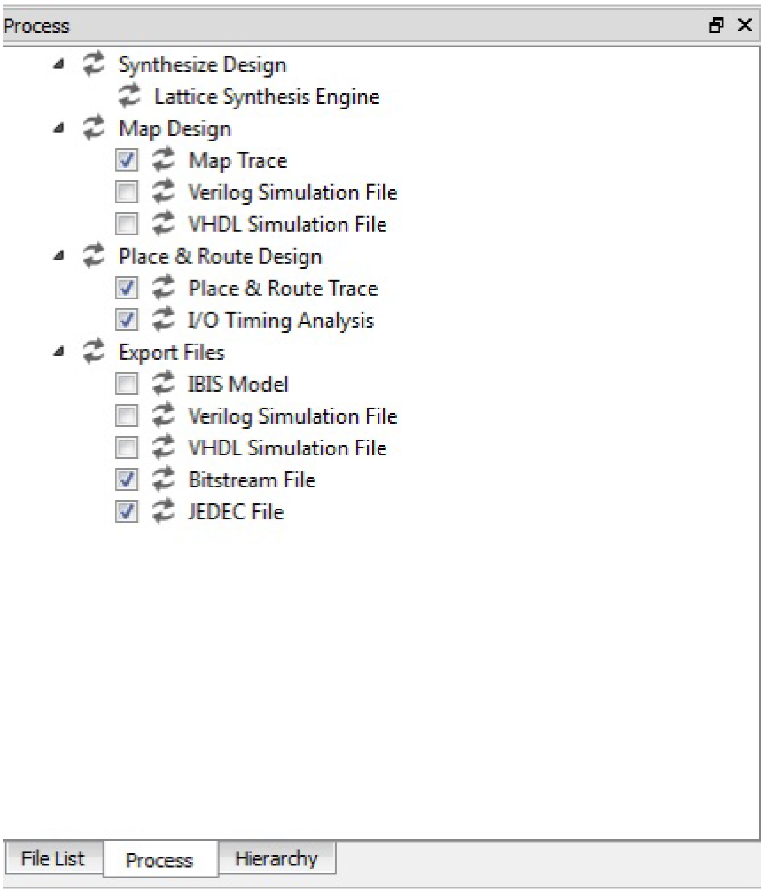
● Lattice LSE More information

6. Open and edit the LPF source file in the section “LPF Constraint File”. This file is used for I/O mapping and clock configurations. Reference to our predefined file “FPGA_IPM.lpf” that you find along with the other VHDL source files for the correspondence between the pinout of the FPGA and the top entity ports. You can anyway customize your own LPF file. Make sure that the frequency constraint is lower or equal than 60 MHz, not to collide with the software driver settings

7. Save all current changes

5.3.2 Synthesis Procedure

1. Go to «Process» tab of the Process Window and select the check marks as in the following.

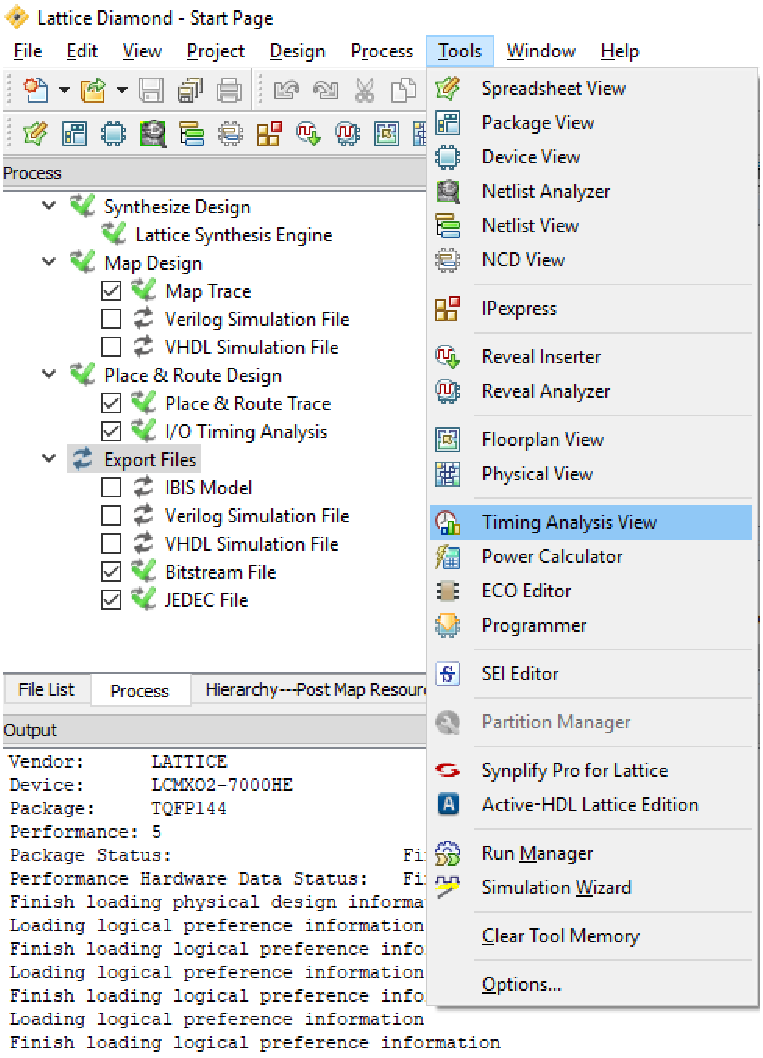


Process

- Synthesize Design
  - Lattice Synthesis Engine
- Map Design
  - ☑ Map Trace
  - □ Verilog Simulation File
  - □ VHDL Simulation File
- Place & Route Design
  - ☑ Place & Route Trace
  - ☑ I/O Timing Analysis
- Export Files
  - □ IBIS Model
  - □ Verilog Simulation File
  - □ VHDL Simulation File
  - ☑ Bitstream File
  - ☑ JEDEC File

2. Right-click on “Run” for all the main voices and check if error messages are present in console

3. At the end of all synthesis steps, go to «Tools » Timing Analysis View» and check if there are no violation for setup and hold times

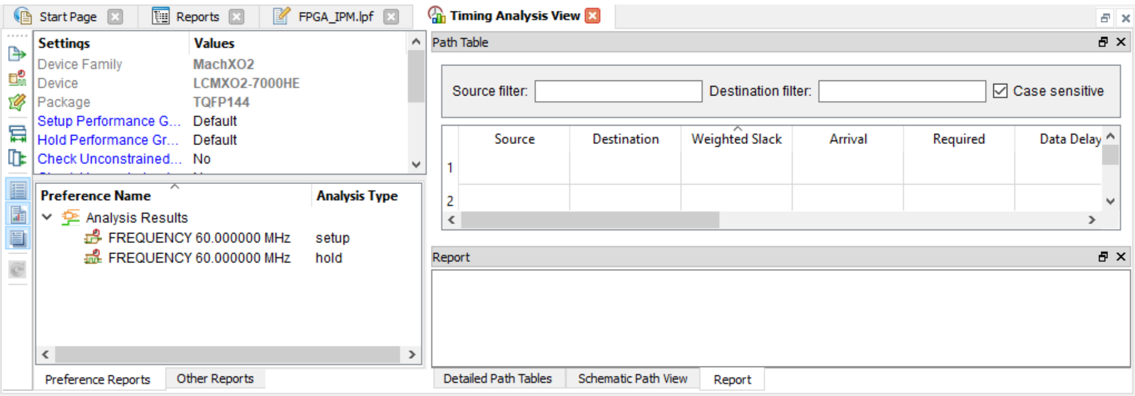


Process

- Synthesize Design
  - Lattice Synthesis Engine
- Map Design
  - ☑ Map Trace
  - ☐ Verilog Simulation File
  - ☐ VHDL Simulation File
- Place & Route Design
  - ☑ Place & Route Trace
  - ☑ I/O Timing Analysis
- Export Files
  - ☐ IBIS Model
  - ☐ Verilog Simulation File
  - ☐ VHDL Simulation File
  - ☑ Bitstream File
  - ☑ JEDEC File

Output

```
Vendor:       LATTICE
Device:       LCMXO2-7000HE
Package:      TQFP144
Performance:  5
Package Status:                           Fi
Performance Hardware Data Status:         Fi
Finish loading physical design informa
Loading logical preference information
Finish loading logical preference info
Loading logical preference information
Finish loading logical preference info
Loading logical preference information
Finish loading logical preference information
```



Settings	Values
Device Family	MachXO2
Device	LCMXO2-7000HE
Package	TQFP144
Setup Performance G...	Default
Hold Performance Gr...	Default
Check Unconstrained...	No

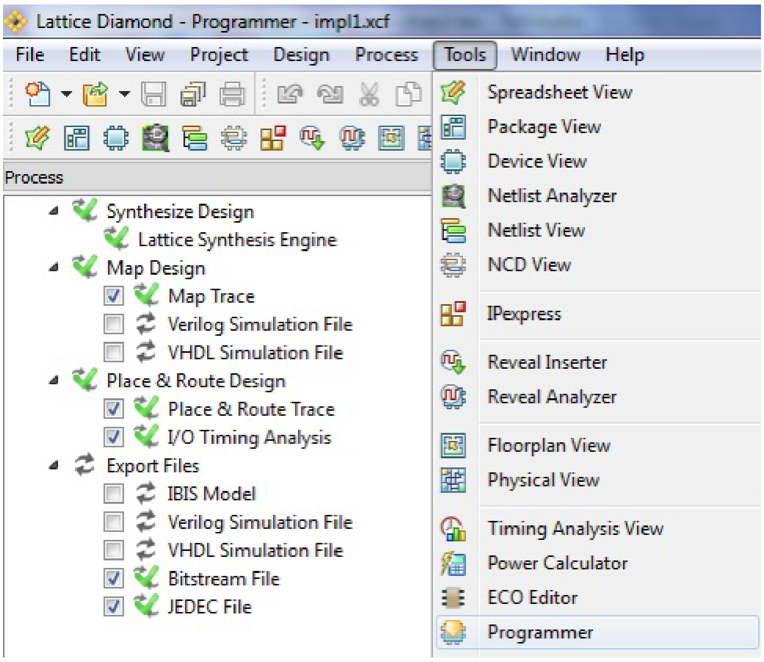
Preference Name	Analysis Type
Analysis Results	
FREQUENCY 60.000000 MHz	setup
FREQUENCY 60.000000 MHz	hold

Source filter: Destination filter: ☑ Case sensitive

	Source	Destination	Weighted Slack	Arrival	Required	Data Delay
1						
2						

If times are respected, constraints are written in black. Otherwise, they are written in red. By clicking on the constraint, on the right it is possible to see details about the violating path

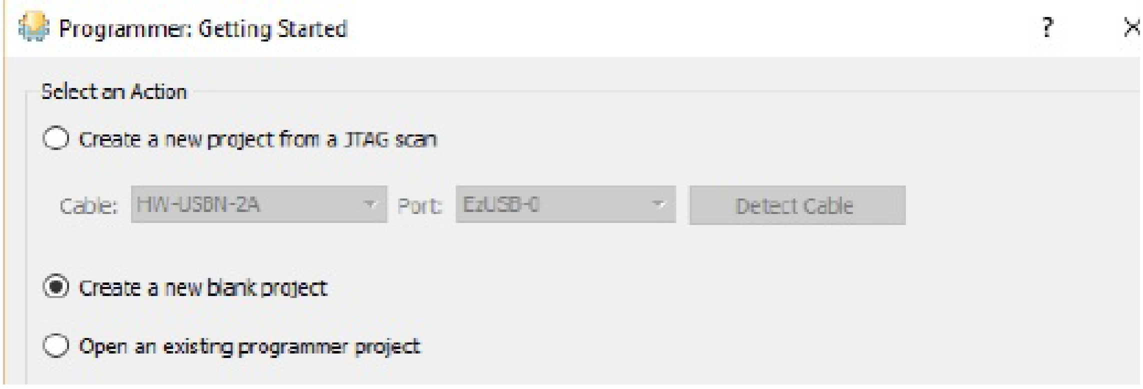
4. If there are no problems, go to «Tools » Programmer»



Process

- ✓ Synthesize Design
  - ✓ Lattice Synthesis Engine
- ✓ Map Design
  - ☑ ✓ Map Trace
  - □ Verilog Simulation File
  - □ VHDL Simulation File
- ✓ Place & Route Design
  - ☑ ✓ Place & Route Trace
  - ☑ ✓ I/O Timing Analysis
- Export Files
  - □ IBIS Model
  - □ Verilog Simulation File
  - □ VHDL Simulation File
  - ☑ ✓ Bitstream File
  - ☑ ✓ JEDEC File

5. Select "Create a new blank project" on the appearing window



Select an Action

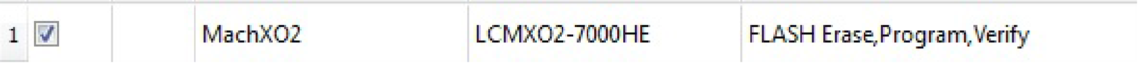
○ Create a new project from a JTAG scan

Cable: HW-USBN-2A Port: EzUSB-0

● Create a new blank project

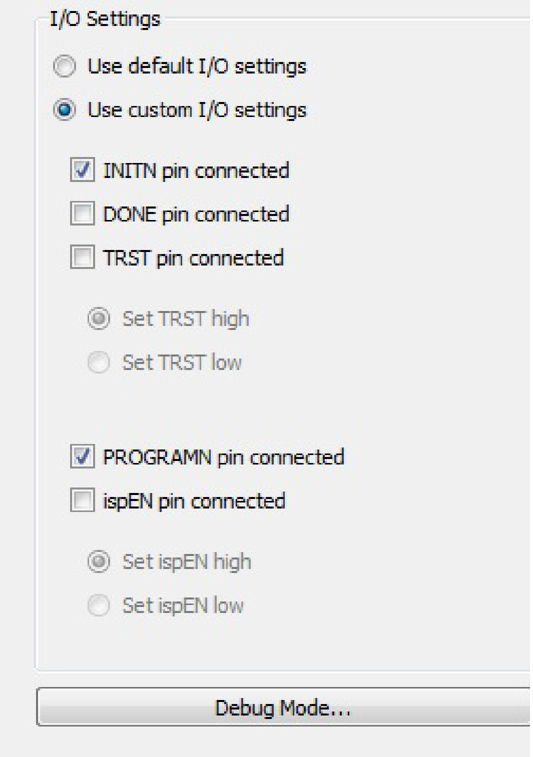
○ Open an existing programmer project

6. Verify the device family and the XCF file name



1	☑		MachXO2	LCMXO2-7000HE	FLASH Erase,Program,Verify

7. Check "I/O Settings" as in figure



I/O Settings

○ Use default I/O settings

● Use custom I/O settings

☑ INITN pin connected

□ DONE pin connected

□ TRST pin connected

● Set TRST high

○ Set TRST low

☑ PROGRAMN pin connected

□ ispEN pin connected

● Set ispEN high

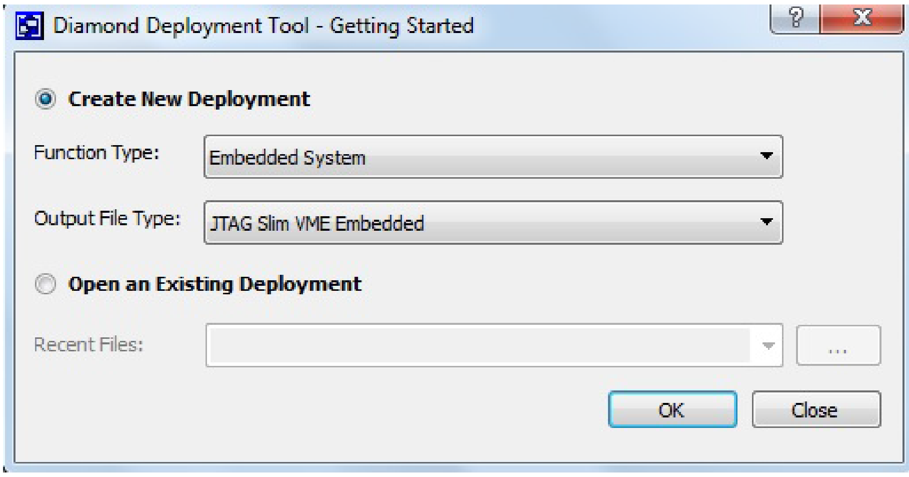
○ Set ispEN low

8. Save the changes applied. This file is necessary for Deployment Tool

5.3.3 Deployment Tool usage

The Deployment Tool has the aim of converting the XCF file into the array format needed to program the FPGA through the microcontroller.

1. First, open the Deployment Tool and create a new project like in the following



Diamond Deployment Tool - Getting Started

● **Create New Deployment**

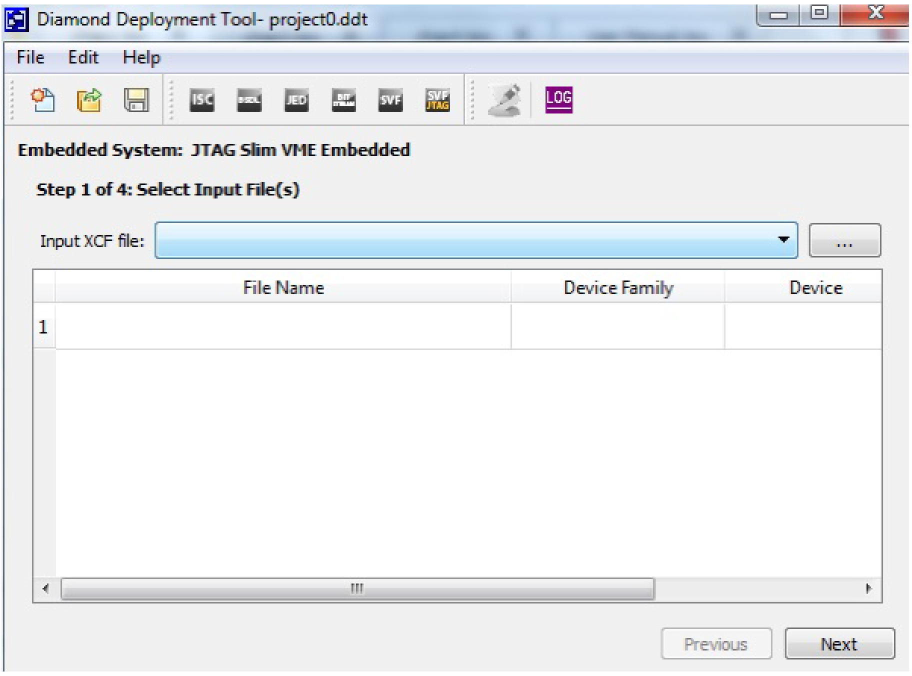
Function Type: Embedded System

Output File Type: JTAG Slim VME Embedded

○ **Open an Existing Deployment**

Recent Files:

2. Locate the XCF file and click on “Next”



Diamond Deployment Tool- project0.ddt

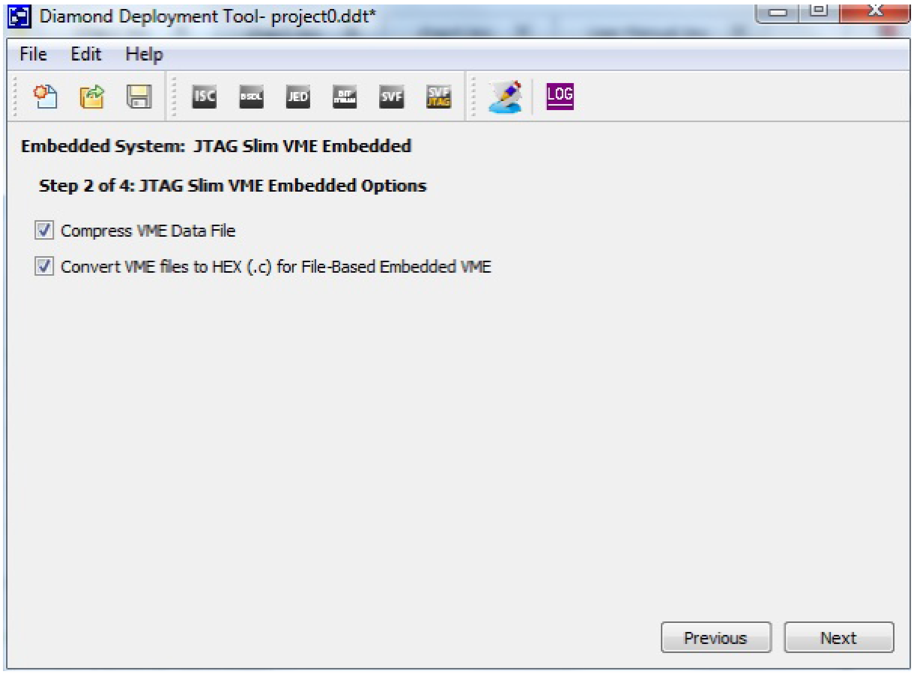
Embedded System: JTAG Slim VME Embedded

Step 1 of 4: Select Input File(s)

Input XCF file:

	File Name	Device Family	Device
1			

3. Be sure to check both the marks “Compress VME Data File” and “Convert VME to HEX (.c) for File-Based Embedded VME” as in figure



Diamond Deployment Tool- project0.ddt*

Embedded System: JTAG Slim VME Embedded

Step 2 of 4: JTAG Slim VME Embedded Options

☑ Compress VME Data File

☑ Convert VME files to HEX (.c) for File-Based Embedded VME

4. Click Next and generate the C files containing the two arrays that will be used to program the FPGA

5.4 Putting all together

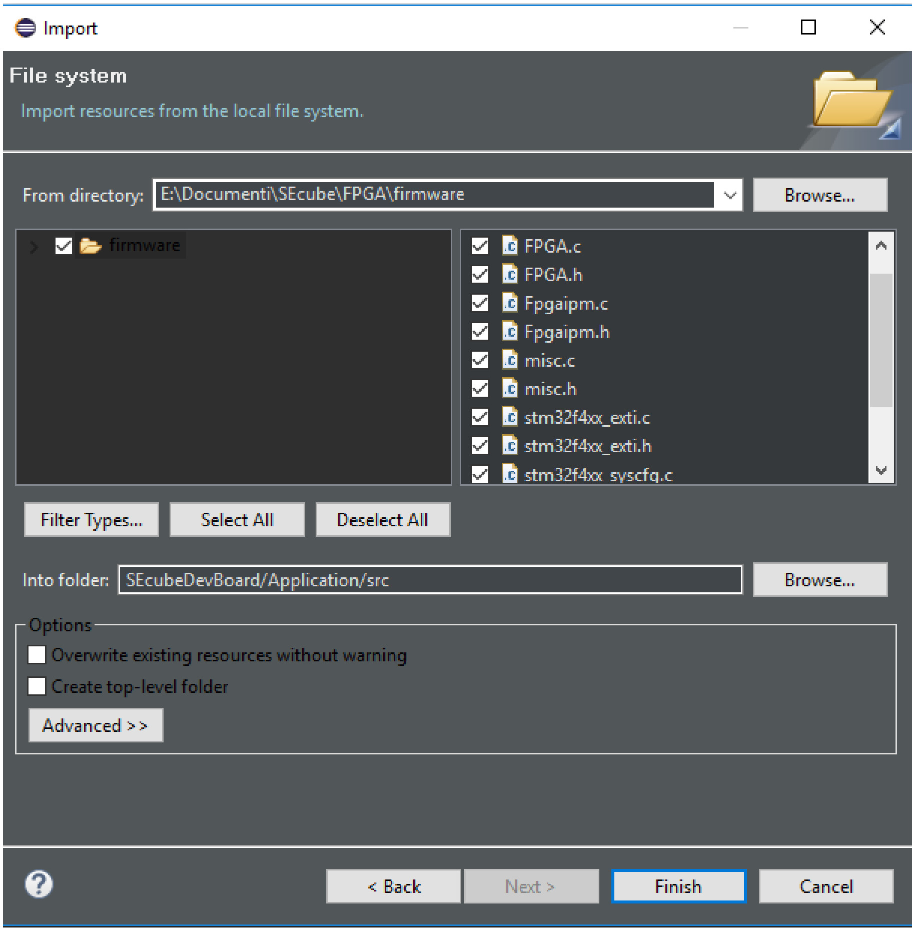
Now that your HDL project is complete, you have to include it in a custom device-side project set up for working with the FPGA. In order to do that,

1. Import the project as described in section 5.2.1

2. Import the necessary «File » Import…», select “Filesystem” and press “Next”

3. Browse to the directory where the API libraries for the CPU-FPGA communication and the additional ST libraries for the CPU are located

4. Select the files in that folder (“FPGA.c”, “FPGA.h”, “TEST_FPGA.h”, “Fpgaipm.c”, “Fpgaipm.h”, “misc.c”, “misc.h”, “stm32f4xx_exti.c”, “stm32f4xx_exti.h”, “stm32f4xx_syscfg.c”, “stm32f4xx_syscfg.h”); you might want to set also “Destination Folder” to “SEcubeDevBoard/Application/src” and then press “Finish”


Import

File system

Import resources from the local file system.

From directory: E:\Documenti\SEcube\FPGA\firmware

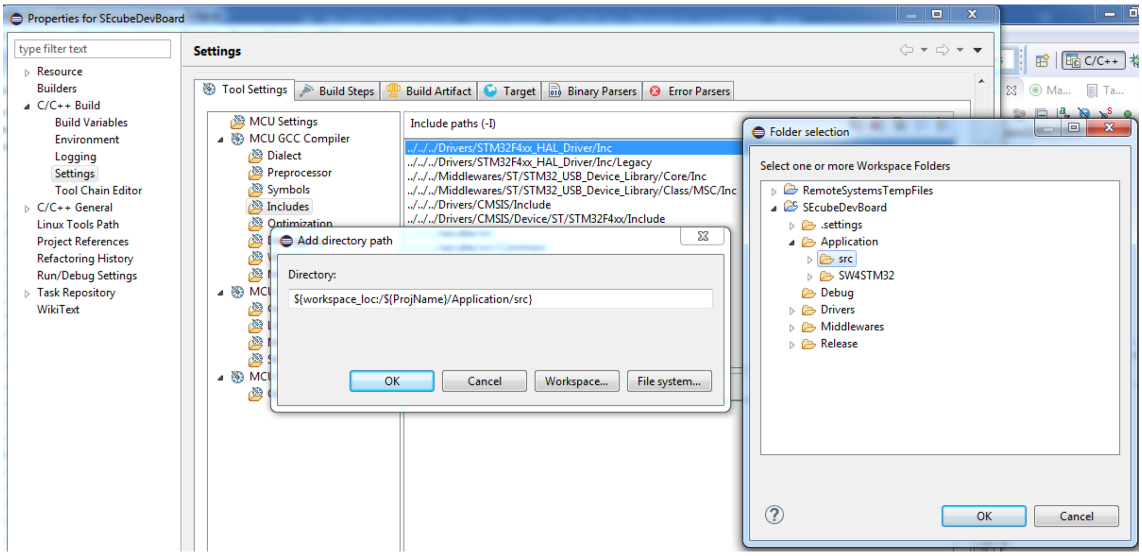
☑ firmware

☑ FPGA.c
☑ FPGA.h
☑ Fpgaipm.c
☑ Fpgaipm.h
☑ misc.c
☑ misc.h
☑ stm32f4xx_exti.c
☑ stm32f4xx_exti.h
☑ stm32f4xx_syscfg.c

Into folder: SEcubeDevBoard/Application/src

Options
☐ Overwrite existing resources without warning
☐ Create top-level folder

5. Configure both “Debug” and “Release” configurations from «Project » Properties » C/C++ Build » MCU GCC Compiler » Includes» and add the “Destination folder”



Include paths (-I)

```
../../../Drivers/STM32F4xx_HAL_Driver/Inc
../../../Drivers/STM32F4xx_HAL_Driver/Inc/Legacy
../../../Middlewares/ST/STM32_USB_Device_Library/Core/Inc
../../../Middlewares/ST/STM32_USB_Device_Library/Class/MSC/Inc
../../../Drivers/CMSIS/Include
../../../Drivers/CMSIS/Device/ST/STM32F4xx/Include
```

Directory: ${workspace_loc:/${ProjName}/Application/src}

Select one or more Workspace Folders

6. Browse on synthetized HDL project folder, check the presence of the .c files with name ending with "_algo.c" and "_data.c". These files are containing the two arrays, gpucAlgoArray[] and gpucDataArray[] that must be substituted in the file "TEST_FPGA.h" already included in the project, thus substituting the two already present arrays

7. Copy the content of these two arrays in the corresponding arrays __fpga_alg and __fpga_data of "TEST_FPGA.h". In file "FPGA.c", values of giAlgoSize and giDataSize must be substituted with the one written respectively at the top of the two files generated by the HDL synthesizer

8. Open the file named "gpio.c" and add the following lines to the function MX_GPIO_Init(), needed for configuring the JTAG port used for programming the FPGA:

```
/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = GPIO_PIN_2;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);
/*Configure GPIO pins : PE3 PE4 PE5 PE6 */
GPIO_InitStruct.Pin = GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5|
   GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);
```

9. Now edit the code in "main.c" file including the header files "FPGA.h" and "Fpgaipm.h"

10. Also in "main.c" add a call to FPGA_IPM_init() along with the MX_ initialization functions for the microcontroller modules. Make sure that MX_GPIO_Init() is called before it, and delete (if present) the call to MX_FMC_Init(), as it collides with the FMC initialization settings already contained in FPGA_IPM_init()

11. Also in "main.c" add a call to B5_FPGA_Programming() function, and before starting to use the APIs to interface with the FPGA, be sure that it is brought to a reset state through adding the two following lines:

```
HAL_GPIO_WritePin(GPIOG, FPGA_RST_Pin, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOG, FPGA_RST_Pin, GPIO_PIN_RESET);
```

which have the aim of resetting the whole FPGA

12. Open the "stm32fxx_hal_conf.h" and uncomment the two defines `HAL_SRAM_MODULE_ENABLED` and `HAL_RTC_MODULE_ENABLED`

13. Go to «File » Import…», select "Filesystem" and press "Next"

14. Browse to the directory where the SDK has been downloaded and then to the path "/secube_sdk/development/environment/drivers/stm32f4xx_hal_driver/Src"

15. Select the file "stm32f4xx_hal_sram.c" in that folder; you might want to set also "Destination Folder" to "SEcubeDevBoard/Drivers/STM32F4xx_HAL_Driver" and then press "Finish"

16. Save the changes to all files and build the project

17. Connect the DevKit as described in previous section

18. 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")

Remember that at startup, all the LEDs of the **SE***cube*™ DevKit are in a weak pullup state which indicates that the programming is advancing. After the programming (that may last up to 2 minutes) the LEDs are set on or off or left in the same state depending on what is stated by the HDL code and the connection done through the LPF file: all the LEDs are in control of the FPGA, as it will be explained in the next Chapter.

6 Technical Guidelines

This section wants to give some guidelines for future hardware/software developers within this project, with reference to technical details of our work.

6.1 Hardware design guidelines

As already mentioned, the IP cores can be designed in whatever way internally but must follow some fundamental rules.

First of all, since the communication protocol reckons on enable, acknowledgment and strobe signals for notifying the end of some activity by the CPU, and their activation is distributed over time, the block must be sequential. A combinational IP core is not feasible for this project.

The external interface of the cores as presented in Section 2.3.2 must not be modified in any way to guarantee the correct behaviour during the communication. The IP Manager is thought to control this and only this interface with the generic core, so if for example the opcode field or the interrupt/polling bit are deemed unnecessary, or the ack and the interrupt lines are not used since the core is thought to work only in polling mode, the ports must be only ignored or kept stuck at given values, but never dropped.

Modifications that do not involve any change in the communication interface with the CPU but, for example, are intended to add ports to drive other I/O pins of the FPGA, the LED (as explained soon) or to read the buttons are instead allowed. The information about the physical connection of the I/O pins of the FPGA are reported at the bottom of the general documentation of **SE***cube*™ [19].

All the cores are fed with the global FPGA clock and reset signals coming from the CPU. When the reset arrives, all the blocks of the design are reset, so the FPGA as a whole is reset, as well as when the clock is suspended by the CPU, the entire FPGA stops. There is no way to gate or control these two signals for the cores through the manager using some special command for the Manager or something. The signals can only be managed internally by the custom description of the core, but this is an unsafe and not recommended solution.

The typical IP core should be structured internally as an FSM with datapath (FSM-D) that performs a given algorithm. It usually remains in an IDLE state until the enable signal arrives. This signal unlocks the machine. The core acquires the information about the type of transaction (interrupt or polling) and the operative code transmitted by the CPU, and usually moves to a reading state, in which it waits for the write completed strobe signal before addressing the buffer and read the inputs. Once read the last input, the core starts doing its job.

The computation may take an undefined number of clock cycles. The CPU can wait a precise number of clock cycles knowing the clock frequency the FPGA is fed with, but this choice is always not recommended for the possible indeterministic delay that can affect the computation. Usually, either the CPU polls a register waiting for the end of the work or it closes the transaction and waits for an interrupt request continuing its operations.

During the computation phase in interrupt mode, the core is not enabled to read the buffer. It is instead enabled to do it in a polling transaction, but the idea is that the buffer should not be used as a RAM by the core, which implements its own internal block of storage if required. The buffer is only left for exchange of inputs and outputs.

Once the algorithm is executed, either the core is still enabled, and so it writes its input on the buffer before unlocking the CPU stuck on the polled register, or it signals the Manager with rising its interrupt line. The request is forwarded to the CPU when possible, as we said, and after a certain time it is acknowledged.

[19] https://www.secube.eu/site/assets/files/1152/secube_sdk_wiki_-_rel_008_190403.pdf, **Appendix B - SE*cube*™ DevKit Schematics**

The IP core thus receives an enable signal along with the ACK, and its machine is unlocked. The access to the buffer is granted and now the core can start writing its outputs. Here a debate can be born on the CPU synchronization: after having started the ACK transaction, when is the CPU enabled to read the buffer? Some time is surely required to fill the buffer with the results, and there is no inverse strobe from FPGA to CPU to say that some write operation has been done.
In the normal case, a further polling solution is considered as philosophically incorrect, but the core can be designed to clear a row in the buffer and to oblige the CPU to poll that area until the writing is not ended. Considering the different temporization between the CPU and the FPGA this is a not recommended solution, which is preferably substituted by a sleep time after the ISR call, as the next section will explain.

For the limited dimensions of the FPGA mounted on the **SE***cube*™ DevKit and consequently of the data buffer implemented in the project, a core should be not designed for reading instructions as input, even if this solution is exploited in many coprocessor examples on FPGA. The field for the operative code left in the command word can somehow try to overcome this problem, allowing the possibility of designing a core in a flexible way, with the ability of executing more than one program on the inputs stored in the buffer. Although, it is to be remarked that the **SE***cube*™ FPGA is not a large device, 7000 LUTs are barely sufficient to host the buffer, the Manager and two or three simple cores, so mounting up a complex and flexible hardware may lead to have an extra-occupancy.
Alternatively, a little microcode memory can be inserted in the RTL of the core and changed with redesign process or even at runtime exploiting the limited resources available. However again, there may be no enough space for writing a program for a core using the LUTs or the distributed RAM as code memory.

The IP cores inserted in the design may be of whatever type, not only accelerators for common algorithms which require parallelism enhancements. Especially in the embedded domain, the FPGA can be exploited to implement memory controllers for external memories or even peripheral controllers. The 47 general-purpose pins present in the external interface of the FPGA have been made available right for this scope. In such scenarios, transactions are aimed at sending or receiving data from external devices. The CPU therefore waits for the end of a data transfer rather than for the results of an algorithm, and it is possible that transactions with different modes are wondered to control the upstream and the downstream. Taking the example of a controller for a local network interface, the CPU could write outgoing packets on the data buffer and could wait for the end of the upstream polling a control word, while it could be informed of packet arrival by the interrupt signal.
In such IP cores, more aimed at communication than at calculation, a certain number of preliminary settings preceding the the actual transfer of data is likely required (settings of transfer rate, preferred data width, window size, packet header and so on), so two different types of transactions could be foreseen by the designer, one for configuration and one for data.
Whatever the core implemented, the developer must take into account that the FPGA is the block that control the LEDs, numbered from 0 to 7, of the **SE***cube*™ DevKit. Referring to the pinout of the FPGA shown in the Getting Started manual and also in Figure 2 of this document, LEDs are controlled by the following pins:

LED	PIN
LED0	PR16C
LED1	PR16D
LED2	PR17C
LED3	PR17D
LED4	PR18C
LED5	PR18D
LED6	PR19A
LED7	PR19B

If not mapped in the LPF file, these LEDs are left in a weak pullup state. To be controlled by a core, they must be mapped in the LPF file following the table above. Remember that, being in pullup, they are lightened by setting at logic 0 their control pin and turned off vice versa. The usage of LEDs must be very useful for core's debugging purposes.

Similarly, the FPGA also controls the two push buttons (PB) placed under the LEDs on the DevKit. The switches control in pulldown (active low) the following pins:

PB	PIN
PB0 (SW4001)	PR19C
PB1 (SW4000)	PR19D

As for the LEDs, also the PBs must be mapped in the LPF file following the table above.

Due to the timing paths present in the data buffer and in the IP Manager entity, the FPGA must run with a clock 3 times slower than the CPU (which runs at 180 MHz), as only integer prescaler can be set and a half rate (90 MHz) is too fast for the two main blocks of the architecture. A 60 MHz (or lower) constraint must be thus set in the LPF file, so that if setup and hold times are violated (see 5.3.2), the designer is advised and is free to choose either to further slow down the clock frequency arriving from the CPU or to edit the design in order to meet the constraint. The chosen running frequency is related to the two generic parameters of the top entity VHDL file, ADDSET and DATAST, which must respect the software configuration stated by the low-level driver, as it will be presented in the following paragraph.

6.2 Software design guidelines

For correctly interacting with the FPGA, a dedicated low-level driver was developed. Functions for initializing the environment, opening and closing a transaction, writing and reading to or from a specified location in the buffer and a global FPGA interrupt service routine are present in this set. A semaphore then is encharged of blocking possible new transactions when one of them is still opened with a core. The APIs use recasts and wrappers for native types and relative addresses not to force the user to know lower details. Therefore, the first suggestion is to always and only use this API to control the CPU-FPGA communication.
In any case, it is strongly recommended to write a piece of software which is aware of the rules of the communication explained in Section 3 (e.g. when in polling mode, first open the transaction, then write inputs, then listen to a register and then read outputs before closing, when in interrupt mode do not poll the core if the interrupt is not raised yet, etc.)
When an interrupt request arrives, the global ISR of the FPGA (connected to pin PA9 of the CPU) is called (`EXTI9_5_IRQHandler()`). The body of this function must be actually customized by the user. Anyway, a trace is already present in the released file of the driver. After having checked that the logic level of the line is high, the CPU should read the address 0x00 of the buffer first, to identify the interrupting core. At that point, the private variable `row0` is updated and can be

switch-cased. For each allowed value, i.e., for each present core, the dedicate interrupt service routine has to be written there. Before returning from the global routine, the interrupt flag is cleared so that the CPU is ready to be interrupted another time.

The generic ISR of a core should open an acknowledgment transaction rather than a normal one, so that both the core and the manager are advised that the request has been accomplished. When received the ACK along with the enable, the core knows that the CPU wants to receive data from it, so the core can immediately start to fill the buffer. If the CPU starts reading the buffer immediately after the transaction opening, it may incur in timing errors, as it does not know what is happening and what are the words already written and the ones to be written yet. It is thus advisable for the CPU to wait a little time, depending on an estimation on what is the time required by the core to write its outputs.

The other functions of the driver can be trusted, left untouched and just used. The `FPGA_IPM_init()` function, however, may be customized for the setup/hold times of the FMC and for the frequency of the clock passed to the FPGA. If the design does not meet the 60 MHz constraint, the second parameter of the macro `__HAL_RCC_MC01_CONFIG()` present in that function can be changed to set another prescaler. The parameter is `RCC_MCODIV_X` and `X` can assume values from 1 to 5, i.e., the FPGA can run at frequencies from 180 MHz to 36 MHz. Once decided the operating frequency, it is also possible to change `AddressSetupTime` and `DataSetupTime` fields of the timing structures relative to read and write operations of the FMC, to stretch them as preferred. Values are expressed in terms of CPU clock cycles. Changes are free, but the overall settings must be compliant each other. In fact, given `X` the value of the prescaler, software parameters `AddressSetupTime` and `DataSetupTime` and hardware parameters `ADDSET` and `DATAST` (VHDL generics of the top entity), the following formulas must be absolutely respected to make the system work:

```
ADDSET = AddressSetupTime / X
DATAST = DataSetupTime / X
```

Since the two hardware parameters are read by a counter internal to the FPGA, they must also be natural non-zero values.

6.3 Single-core applications

The FPGA architecture developed in this project is thought for a multicore environment, where multiple IPs coexist to perform different tasks, with the advantage of being able to address them individually without having to reconfigure the entire FPGA whenever you want to use a different service. However, in principle nothing forbids having an environment where there is a need to exploit the advantages of a single IP core. In this scenario, the IP Manager has no effective task to perform and becomes a useless intermediary between the buffer and the only IP present. For single-IP applications that require optimized timing and area constraints, the Manager could be dropped without the need for a radical redesign of the environment: the IP can maintain the standard interface defined by our work.

In fact, since the IP Manager works primarily as a dynamic connector of the core interface with the buffer interface, in case there is a single core this connection can be operated statically in the top entity of the VHDL design, without changing anything neither in the default interfaces of the buffer and of the core nor in their internal behavior.

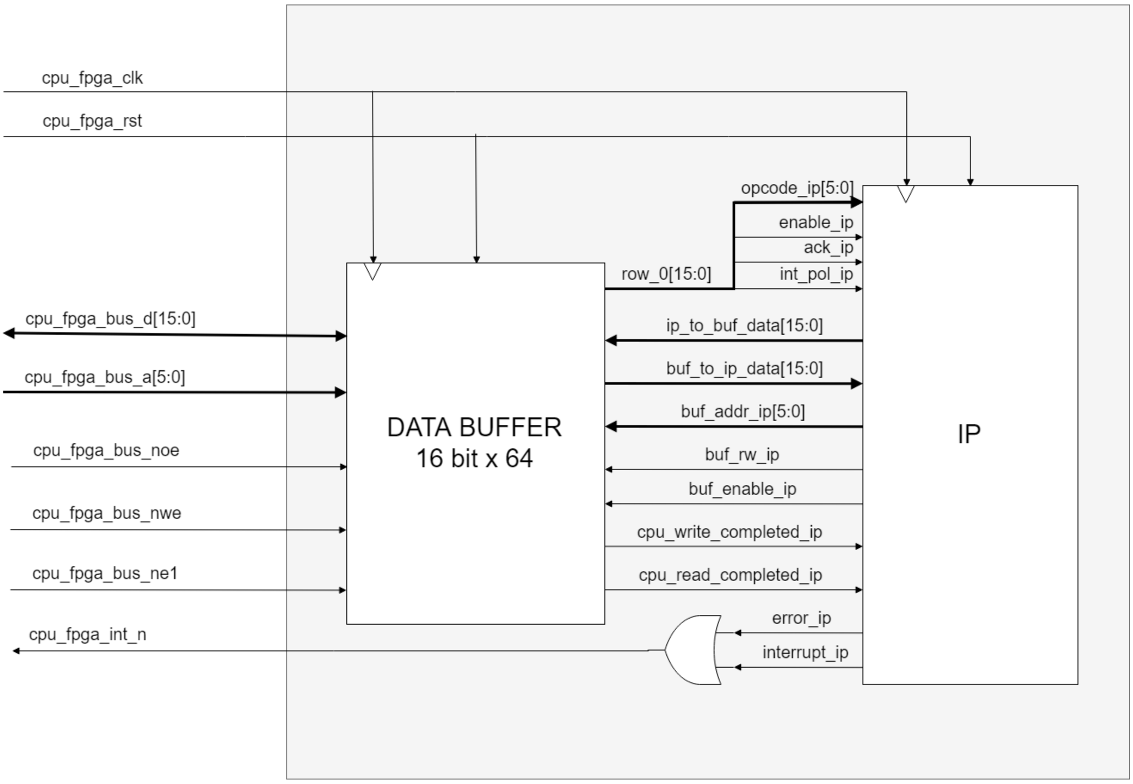


Figure 24: Example of the FPGA internal connections with a single IP core

The Figure 24 shows the internal connections of an FPGA package with a single core and without the IP Manager. As it is possible to see, there is no port modification in the entities. The first address of the buffer, containing the control word, could be simply split in its components and immediately reflected to the IP which then is informed of the start/end of the transaction and of the transaction parameters. Data, address, control and strobe signals are not multiplexed but statically assigned to the only IP present, which uses them in the exact same way than in the IP Manager scenario. The interrupt and error signals are not to be handled by any controller, since the request can only arrive from one IP, so both of them can be attached to the FPGA global interrupt signal. The distinction between these two cases could be overcome.
Obviously, the CPU must be aware of this situation, and the driver should be minimally changed to avoid useless controls. For example, it is no longer necessary to check at software level the identity of the active core, but only if there is already an open transaction or not, at most. Alike, setting the **IPADDR** field of the control word becomes pointless (that part of row 0 is neglected at buffer interface).
The new ISR for such an FPGA could skip the reading of row 0 and immediately start an acknowledgment transaction to get the results or to understand the problem, in case the interrupt was triggered in response of an error line assertion.

7 An example IP core: SHA256

After creating a reliable environment, we decided to include an example IP core in the project, a core that executes an algorithm of effective and wide use in the cryptographic domain. We are talking about the SHA256. This section is intended to present the SHA256 IP core, first introducing it with a brief overview of the algorithm and the architecture, and then continuing with the presentation of its use modes.

7.1 Overview

Secure Hash Algorithm (SHA) is a label given to a family of cryptographic algorithms which have been developed and published since '90s by the U.S. Government. Given a message or a file or any sequence of information, the SHA produces a *digest* for it, a compact string (of 256 bits in the SHA256 case) which is unique and not reversible. These features make it a safe way to store sensitive information, preventing it from tracing back to the original message (think of the case of passwords stored on servers) or from modifying (think of a file whose hash represents its kind of digital certificate, since just a few bytes of difference completely change the digest).
The hashed message can be of whatever dimension but must be divided into *blocks* of 512 bits that one after the other contribute to transform the starting hash (always the same, defined by the standard) into the digest with a series of plain operations, mainly rotations and additions. At every round of the algorithm, one 32-bit word from the block (extended from 512 to 2048 bits) enters the main sequence along with the hash (called also state), and as a result a new state is produced. There are thus 64 rounds like this, then at the end the state is summed with the starting hash to obtain the digest. The algorithm is therefore mostly sequential and, more important, one block must wait the computation of the following[20].
The core we developed for the execution of the SHA works with one block and leaves the division of the message into blocks and the padding of the last one to the driver. In other words, every time a block is to be computed, the intermediate state and the block itself must be written on the data buffer and the production of a new state must be awaited. This was an almost obligatory choice for the reduced dimensions of the data buffer, which offers a memory space which is barely enough to contain the state (16 16-bit words) and a single block (32 16-bits words).

[20] For additional details on SHA family of algorithms, please refer to https://web.archive.org/web/20130526224224/http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

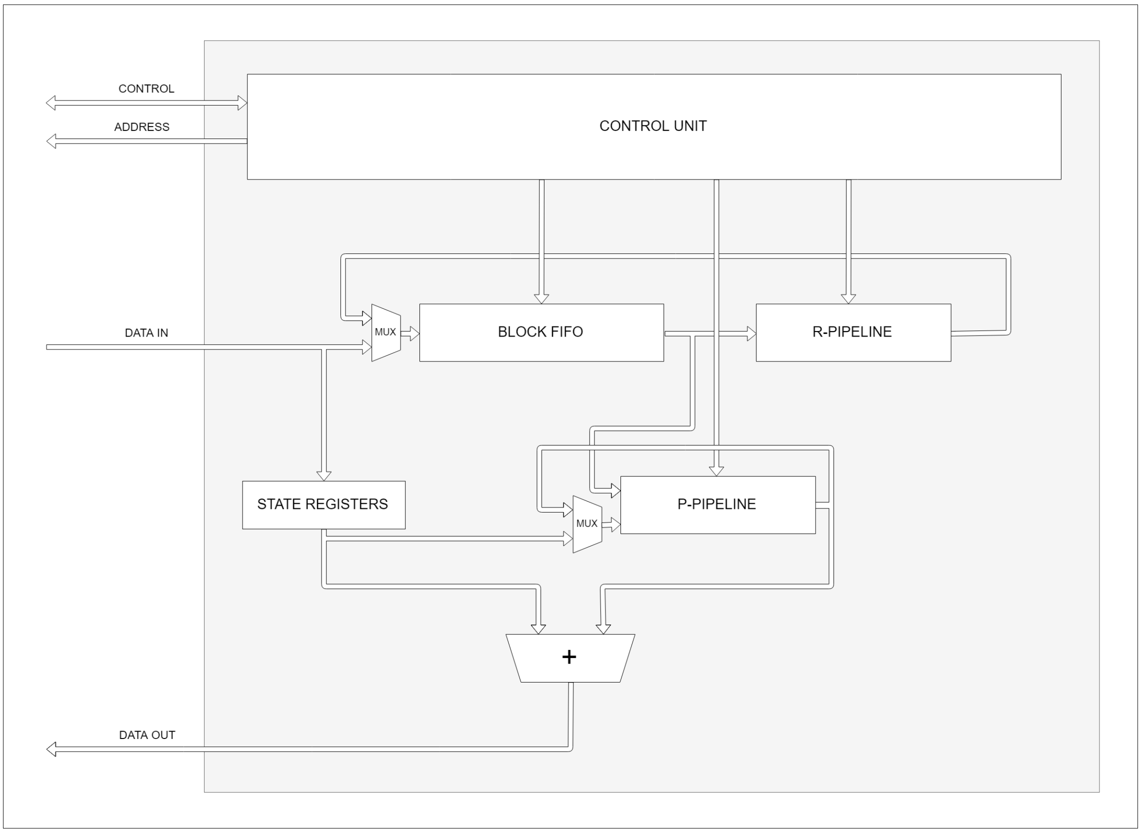


Figure 25: SHA256 IP core architecture overview

The Figure 25 shows the internal architecture of the core. The control unit is the main central entity of the core, and contains the microcode for controlling the datapath, but also is responsible of control signal interpretation and driving and address forwarding towards the buffer. Whenever the enable signal is sensed, the machine inside this unit awakens and starts reading the state putting it into the dedicate registers. After that, the reading of the block starts: the acquired words are inserted in a queue of registers called *block FIFO*. The transaction is then closed and the chain, now filled, releases its content with a first-in-first-out policy towards a first computational block called *R-pipeline*. This pipeline contains 3 stages and is used to extend the block from 512 to 2048 bits creating 48 additional 32-bit words needed for the computation. The work of the R-pipeline actually goes in parallel with the main sequence of the algorithm, represented by the *P-pipeline*. This is a 4-stage pipeline which makes the actual encryption of the block. The first word of the block is encrypted using the initial state stored in the dedicate registers, while the following are encrypted using the state coming from the previous cycle of the P-pipeline. Once every of the 64 32-bit words have passed through the 4-stage pipeline one after the other, the final sum between the obtained state and the initial one can be computed, and the output can be stored, with modes and times dictated by the transaction mode adopted (interrupt/polling).

7.2 Testing the core via HDL simulation

In the downloadable material relative to this project, the folder containing the HDL code for the FPGA also contains a testbench (“FPGA_testbench.vhd”) which can be simulated with whatever HDL simulation tool available to you. We anyway recommend the use of Altera™ ModelSim, for which a TCL script file is already provided in the same folder.

The testbench contains procedures which emulate the FMC behaviour during read and write operations and 2 benchmarks that show a polling and an interrupt transaction respectively, with the core computing the same block with the same starting hash.

7.3 Testing the synthesized core via high-level driver example

In the other folder of the downloadable material, the C source files are stored. These files are to be added to the Eclipse project as explained in 5.4. The file "TEST_FPGA.h" present here already contains the FPGA configuration with the SHA256 core, as well as the file "FPGA.c" already contains the correct value for the variable `g_iDataSize`. Therefore, the process of synthesis explained in Sections 5.3.2 and 5.3.3 can be skipped.

Besides the files named in Section 5.4, two additional files ("sha256_fpga.c" and "sha256_fpga.h") must be added with the same procedure in order to interact with the SHA256 IP core. Such files contain an example on how a high-level driver for controlling a core can be written. It offers just one public function,

```
SHA256_FPGA_digest_message(const uint8_t *message, uint64_t
   dataLen, uint8_t *digest)
```

which receives the pointer to the message, the pointer to the resulting digest and the number of bytes to be processed. Internally, it divides the message into blocks and pads the last block, and every time a block is ready to be computed, it is sent to a lower-level private function which handles reads and writes to establish a polling transaction with the core for performing the required computation.

The header "sha256_fpga.h" must be included in "main.c" for correct working, besides the includes already suggested in 5.4.