SEcube™
Open Security Platform

Introduction

Release: February 2021

Proprietary Notice

The present document offers information subject to the terms and conditions described hereinafter. The authors reserve the possibility to change the content and information described in this document and to update such information at any time, without notice. Despite the attention that has been taken in preparing this document, typographical errors, error or omissions may have occurred.

Authors

Matteo FORNERO *(Researcher, CINI Cybersecurity National Lab)* matteo.fornero@consorzio-cini.it
Nicoló MAUNERO *(PhD candidate, Politecnico di Torino)* nicolo.maunero@polito.it
Paolo PRINETTO *(Director, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(PhD candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Revision History

Release	Date	Changes
008	April 2019	First public release (revisions prior to rel. 008 are not public).
009	October 2019	Updated L0/L1 APIs documentation. Added FPGA documentation, guidelines for **SE*key*™** , **SE*cube*™** data sheet, **SE*cube*™** DevKit schematics, Hello World application example.
010	June 2020	Added documentation about the **SE*key*™** library. Removed outdated documentation about **SE*link*™** library. Updated documentation about **SE*file*™** .
011	February 2021	Added documentation about the new **SE*link*™** library. Updated documentation about L0 and L1 libraries. Updated references to downloadable files in Sections 5 and 9. Updated user guidelines in Section 9.

Contents

1 Introduction

The **SE*cube*™** (Secure Environment cube) Open Security Platform is an open source security-oriented hardware and software platform, designed and constructed with ease of integration and holistic security in mind.
The present section provides a global overview of the **SE*cube*™** *Open Security Platform*. In particular, it presents the evolution of the project behind the platform, its assets (both on the software and the hardware side), its functionalities and the community and the academia program that are growing around the **SE*cube*™** .

1.1 The need for Open Security Platforms

Nowadays, cybersecurity is one of the biggest necessities of the world, because many aspects of our lives resolve around computers and digital infrastructures. People are always connected, posting information about their lives on Facebook, Instagram, Twitter and tens of other social and communication networks. At the same time, most of enterprises are heavily based on a digital infrastructure. All the transactions and important information are stored in personal computers or servers placed and interconnected everywhere around the world. Such a scenario provides endless opportunities to cyber-attackers, whose number dramatically increases every day.
In order to protect the entire information chain, today more than ever, there is a significant need for pervasive security, which is the right security, in the right place at the right time.
Such a security deployment is neither easy nor impossible. However, a layered holistic security approach must be taken, intertwining security technology, physical security, and logical or operational security in the right parts of the information flow.
As a matter of fact, a so pervasive approach may be too challenging for both developers and final users, unless a proper abstraction level is provided. A methodology is thus required to efficiently implement holistic security on hardware and software security systems, which are often underestimated or ignored when a security solution needs to be deployed.
Although software gives easy and flexible protection, hardware is much faster and provides better immunity from contamination, malicious code infections or vulnerability. For instance, hardware-based encryption offers stronger resilience against many common attacks. This is even more effective when heterogeneous technologies are integrated in a unique embedded platform. In this case, in fact, the complexity of possible cyber-attacks drastically increases; therefore, there are several hardware techniques to enforce the security of the system, such as hardware anti-tampering, redundancy, fault-detection, etc.
Nevertheless, such a complex platform also increases the development complexity, requiring to combine and harmonise different technologies in a seamless way.
There are a few security-oriented open platforms available on the market. Some of them are focused on the evaluation of the system robustness against external physical attacks (e.g., Side-Channel attacks, power crypto-analysis, etc.), such as the Sasebo board[1] and the ChipWhisperer[2]. Other platforms based on ARM processors, like Juno ARM Development Platform[3] and the open source USB device provided by InversePath[4], allow creating general purpose software applications, including security-oriented solutions. Nevertheless, they are based on application processors and there are not specific security elements to be fully controlled or customized by the de-

[1] Toppan Ltd, "Side-channel Attack Standard Evaluation Board: SASEBO", http://www.toptdc.com/en/product/sasebo/
[2] C. O'Flynn, and D. C. Zhizhang. "ChipWhisperer: An open-source platform for hardware embedded security research." In: Constructive Side-Channel Analysis and Secure Design. Springer International Publishing, 2014. 243-260
[3] Arm LTD, "Juno ARM Development Platform", http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php
[4] Inverse Path Srl, "USB Armory", https://inversepath.com/usbarmory.html

velopers. Finally, there are single chips realized as a combination of one FPGA and one CPU, like Zynq proposed by Xilinx[5] or Excalibur based on Altera technology[6]. Nevertheless, in both cases the platforms are more suitable at prototyping stage, since they are not cost-efficient, and a specialized security element, like a smart card, is still missing.
In addition, although in the last 25 years many abstraction layers and interfaces have been proposed to integrate hardware security tokens in general purpose systems, the provided functions are typically limited to the low-level encryption operations without giving a real abstraction level independent from the single cryptographic operation.
For example, the PKCS#11 interface[7] available since 1994 and still largely used in the public key security infrastructures, provides over 50 functions to deal with cryptographic algorithms. Nevertheless, the cryptographic algorithms are treated with specific interfaces depending on their nature (e.g., digest, encryption, key generation, etc.). At the same way, Microsoft CSPs (Cryptographic Service Providers) defines several proprietary cryptographic packages with specific interfaces according to the security provider. In any case, there are not abstraction layers providing a service-oriented set of security functionalities implemented on complex and heterogeneous HW/SW security platforms.

1.2 The SE*cube*™ *Open Security Platform* Project

On the basis of the above considerations, in 2015 Blu5 Group[8] involved several Academic institutions in launching the **SE*cube*™** *Open Security Platform*: an open-source security-oriented hardware and software platform, designed and constructed with ease of integration and holistic security in mind.
SE*cube*™ introduces a new approach to provide hardware and software holistic security through abstraction layers which try to hide classical security concepts like cryptographic algorithms and keys focusing, instead, on common operational security concepts like groups and policies.
The Platform is based on the **SE*cube*™** (Secure Environment cube): a security-oriented 3D SiP (System in Package) designed and produced by Blu5 Group. It integrates three key security elements in a single package: a fast floating-point Cortex-M4 CPU, a high-performance FPGA, and an EAL5+ certified SmartCard (Figure 1).

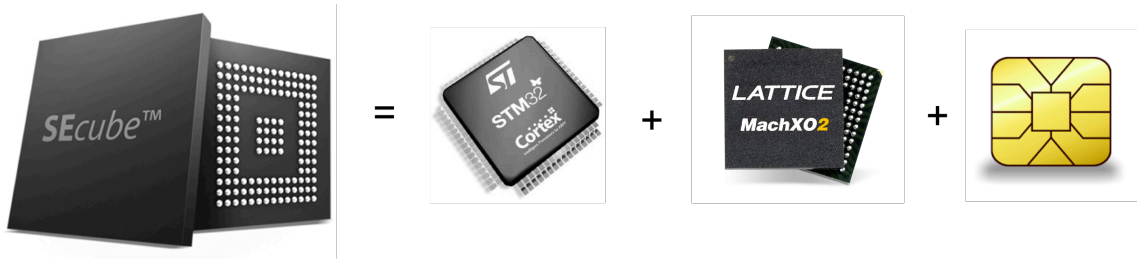


Figure 1: The 3 components of **SE*cube*™** .

[5] L. H. Crockett, R. A. Elliot, M. A, Enderwitz, and R. W. Stewart: The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Ac. Media, 2014
[6] Altera Corportation, "Excalibur Devices", https://www.altera.com/products/general/devices/arm/arm-index.html
[7] Clulow, J. (2003, September). On the security of PKCS# 11. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 411-425). Springer Berlin Heidelberg.
[8] www.blu5group.com

1.2.1 Project evolution

The project currently involves the following institutions:

- Blu5 Labs Ltd, Blu5 Group, Ta Xbiex, Malta –
  Reference: Antonio VARRIALE, av@blu5labs.eu
- CINI Cybersecurity National Lab (Politecnico di Torino Node), Torino, Italy –
  Reference: Paolo PRINETTO, paolo.prinetto@polito.it
- LIRMM, CNRS, Montpellier, France –
  Reference: Giorgio DI NATALE, giorgio.dinatale@lirmm.fr

In 2017, the **SE*cube*™** platform has been selected as the Open Security Platform for the project *FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks*[9], supported by CINI Cybersecurity National Laboratory and funded by CISCO Systems Inc.
Within such a project, the **SE*cube*™** platform is currently used to protect, among the others, Water Supply Systems, Robots and ExoSkeletons of healthcare, IoT applications, and IP protection in industrial machines.

1.3 The Hardware side of the platform

The hardware side of the platform relies on the **SE*cube*™** Hardware device family, which comprise a complete chain of devices, from chip to PCIexpress Board, and namely (Figure 2):

1. The *Chip*, named **SE*cube*™** Chip, or simply **SE*cube*™**
2. The *Development Board*, named **SE*cube*™** DevKit
3. The *Stick*, named U**SE*cube*™** Stick
4. The *Phone*, named **SE*cube*™** Phone
5. The *PCIexpress Board*, named **SE*cube*™** PCIe.

Figure 2: The **SE*cube*™** Hardware device Family.

1.4 The Software Architecture of the platform

The **SE*cube*™** Open Source Software Architecture is structured in several *Abstraction Layers*, as summarized in Figure 3, usually referred to as *L0, L1, L2, L3*, and *Applications*, respectively. At each *Abstraction Layer*, several sets of APIs are provided.
At each level, each component (but the lowest one) represents a “service” for the upper level and relies on “services” provided by lower levels.

[9] www.filierasicura.it

The **SE*cube*™** *-Side* APIs are executed on the embedded processor of the **SE*cube*™** -based hardware device (e.g., the U**SE*cube*™**), whereas at the *External-Side*, the software is tailored for existing devices (e.g., laptops or Desktop PC) that see the **SE*cube*™** hardware as an external peripheral which exposes the **SE*cube*™** *-Side* APIs. In this scenario, **SE*cube*™** acts as a powerful coprocessor providing a secure and fully controlled execution environment.
The *External-Side* Libraries are designed to be scalable and portable on different Operating Systems, thus limiting the usage of and isolating platform-dependent modules. They practically run on top of the host OS, directly relying on the OS system calls. To improve portability and migrations, the libraries are organized in such a way that all the OS-dependent sub-modules (e.g., communication interface, file system, etc.) can be easily identified.
All the software is released in source code under GPLv3 license[10].

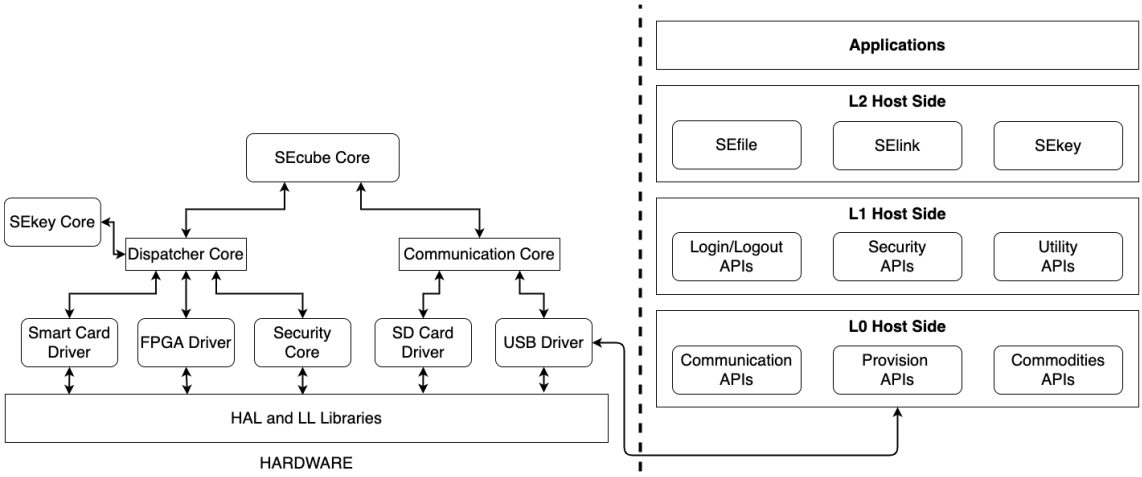


Figure 3: The **SE*cube*™** Software Architecture.

1.5 The SE*cube*™ Assets

In the sequel we shall focused on some of the most relevant **SE*cube*™** assets.

1.5.1 State-of-the-Art Technology in your hands

SE*cube*™ provides the most advanced security hardware technologies in one chip. Being open, both the hardware and the software parts are completely disclosed and documented.

1.5.2 Holistic Security

Within **SE*cube*™** , all the digital and organizational security processes are integrated in a comprehensive, flexible, and seamless way. No need for developers to look at the single building parts. Mathematical and cryptographic elements, like keys and algorithms, are replaced by simpler concepts, like groups and policies (cfr. Section 2).

1.5.3 Multi-Flavor and Multi-Level Libraries

Leveraging on three embedded technologies, the same API can be executed on different cores (STM32, FPGA, Smart Card). The APIs are organized in modular libraries and abstraction layers. Developers are open to create their solution starting from the most suitable entry point according to their expertise. **SE*cube*™** is ready to transform your security ideas to security products. Starting

[10] https://www.gnu.org/licenses/gpl-3.0.en.html

from the Open Source Software Architecture, programmers can easily create, verify and deploy their security solutions on **SE*cube*™** -based professional devices and appliances.

1.5.4 Work Compartmentation for Limited Liabilities

The **SE*cube*™** security architecture allows role separation (Developer, Security Administrator, Users) for work optimization by competence. The same architecture allows protecting all the production and deployment chain players (Factory, OEMs, Final Customers) isolating their respective roles and liabilities.

1.5.5 Pre-Built Functionalities

Basic and standard security modules are ready for use to save up time and resources. In addition, security experts can verify, improve and extend the **SE*cube*™** functionalities working on the open source code.

1.5.6 Full Customization

Starting from the modular and reusable functions, developers are able to reinvent the basic security blocks for new fully customized and controlled security systems.

1.6 The SE*cube*™ Entry Points: How to use SE*cube*™ ?

Leveraging the libraries and applications that are freely available to download from the Internet, you are provided with both low-level (i.e., firmware and middleware to interact at low level with the hardware) and high-level (i.e., fully fledged ready-to-use applications to secure your data and communications) entry points to the **SE*cube*™** Open Source Security Platform.

1.7 The SE*cube*™ Community

Leveraging the platform thought, we intend to create and nurture over time a community for developers at the different levels of security competence and in different application domains. This will ease sharing project, knowledge, and resource and provide the collectivity of members with specialized support tailored to their needs.

1.8 The SE*cube*™ Academia Program

Academic Institutions and Research Centers interested in using the **SE*cube*™** Open Source Security Platform and Devices in:

- Courses and Thesis
- Summer Schools
- Funded Projects
- Consulting Activities
- Hackatons and Competitions

can apply for a dedicated Program offered by Blu5 Group and receive the following benefits:

- Discounts on **SE*cube*™** DevKit and Devices
- Free Basic Technological Trainings for Academics

- Potential involvement in Industrial and Market-Driven projects and activities

The **SE*cube*™** Academia Program is completely FREE.

2 Holistic Security

A security system becomes appealing when both developers and users are not aware of its complexity. This result can be definitively reached when all the digital and organizational security processes are integrated in a comprehensive, flexible and seamless way.
This approach is called *holistic security*. Users and developers are not required to look at the single building parts. They can rather focus on using the system complexity without taking care of it. On these principles, the security system should be realized keeping in mind that classical security-related concepts like cryptographic algorithms and encryption keys should slowly disappear among the hardware and software abstraction layers provided by the designers.
Although sooner or later information must be secured with cryptographic techniques, the related complexity (e.g., mathematics, statistics, security paradigms, implementations) can be demanded to security experts, which oversee provisioning and maintaining the system, whilst users and application developers can be focus on the security service at a higher level.
Information can typically be in two major status: *at rest* and *in motion*. In both cases the most important and intuitive question to be considered when protecting data is the following: who is the recipient of the information that needs to be protected? Usually there are three possible answers:

- Information protected for yourself (*Personal Security*)
- Information protected for a group of people (*Group Security*)
- Information protected for anybody sharing the same security platform (*Family Security*).

On this assumption, the security can be applied to the information independently from the kind of secure service we are targeting. For example, secure services like secure databases, secure file repositories, secure galleries, secure wallets, etc. can be protected by means of the **SE*file*™** holistic library (cfr. Section 6), since this is the case of data at rest protection. For services like secure voice calls, secure messaging, secure client-server web applications, etc. the **SE*link*™** holistic library (cfr. Section 7) can be used, since we are in the case of data in motion protection.
Security services based on the holistic security approach, can be also developed without taking care of cryptography, hardware/software implementations or any other architectural complexity. Concepts like *closed communication / protection groups* and *security policies* can easily replace keys, algorithms, and cryptographic parameters.

2.1 Secure Communication/Protection Groups

A *Secure Communication/Protection Group* is a pool of one or more users. Each group is featured by a set group keys and a security policy, both shared by ALL the members of the group.
The keys of the group are used to protect both *static data* (data at rest) and *data transfers* (data in motion) among the members of the group. In particular, the keys are usually used, in the former case, to derive “secrets” (or “seeds”) used by cryptographic algorithms, and, in the latter one, to set up secure communication channels.
The security policies allow specifying, among the others, the cryptographic algorithm used to protect the information related to that group and the mechanism to be adopted for generating the session keys.

2.2 Centralized vs. Distributed security infrastructures

In *managed security infrastructures*, also called *centralized security infrastructures*, users are grouped in secure groups by a security administrator that operates on a Key Management System

(KMS).
The KMS is an extremely important part of the security infrastructure, since it is entitled to:

- provision/initialize all the security elements (e.g., USB tokens, etc.);
- create and populate communication/protection groups;
- refresh and distribute the communication/protection keys and policies.

In *distributed security infrastructures*, i.e., in security infrastructures not managed by a centralized system, any user can be administrator of one or more communication / protection groups. It is strongly recommended not to have more than one administrator per group, to respect the responsibility paradigm: in case of disputes, the responsible should be univocally identified.
In any case, for both centralized and distributed security infrastructures, the concepts of groups and policies can be applied instead of keys, cryptographic algorithms, etc.
On the functional side, both the developers and the users just see groups. They are not required to know which algorithms or keys are behind each group. The *security administrator* will associate keys and algorithms to any group and populate them with users according to the organization security policies.
It is easy to understand that two or more users can access the information only if they share at least one group.
In this way, at the application level it is not required to specify any kind of encryption method or cryptographic keys. As described in Sections 6 and 7, by means of data at rest and data in motion protection libraries, it is possible to reach a very high security abstraction level through a small set of APIs that are easy to be integrated in the final service.
As a matter of fact, this approach isolates the cryptographic functionality from the security service. Nevertheless, another abstraction layer is required to isolate the cryptographic functionalities from their implementations, which may rely on different technologies (e.g., CPU, FPGA, and SmartCard).
The combination of a proper software architecture, which provides intuitive security at application level, and a seamless cryptographic framework, which harmonizes different implementations in a unique object-oriented interface, allows to design a real holistic security system on top of a complex HW/SW platform like **SE*cube*™** .

3 The Hardware side of the Platform

3.1 The SE*cube*™ Chip

The core of the **SE*cube*™** Hardware device family is a 3D SiP (System in Package) (a detailed view is shown in Figure 4), integrated in a 9mm x 9mm BGA package (Figure 5). The full **SE*cube*™** Data Sheet is available in Appendix A[11].

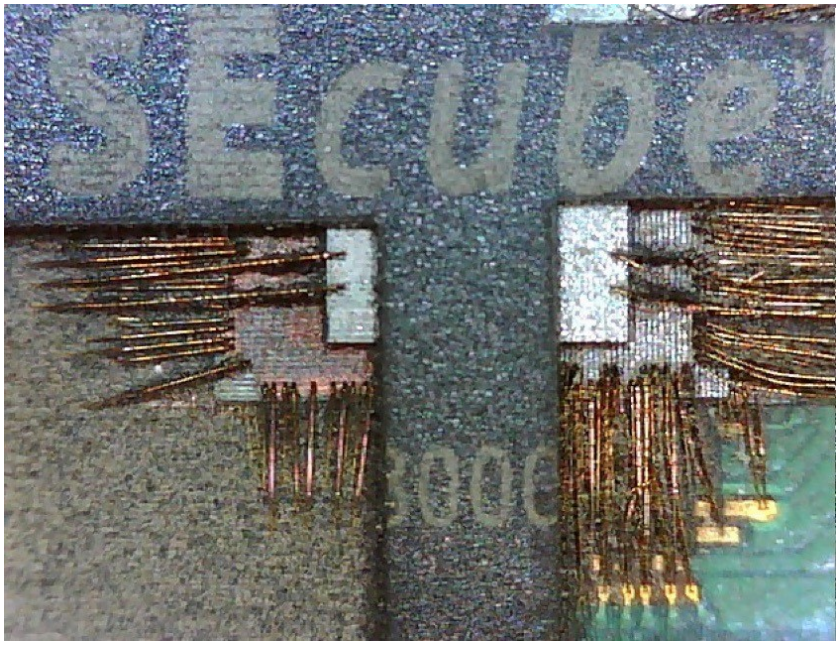


Figure 4: Detail of the die of the **SE*cube*™** Chip

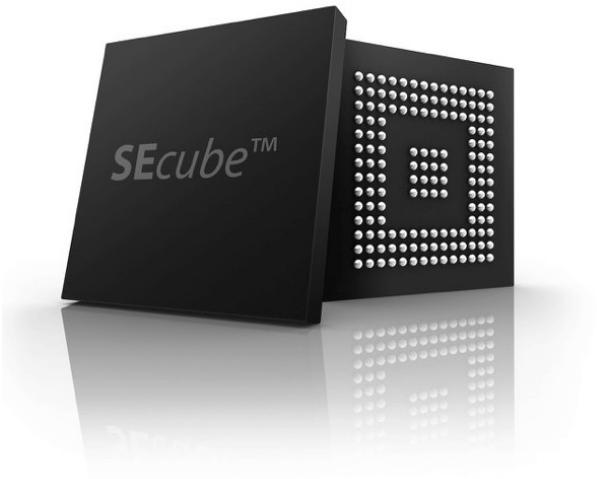


Figure 5: The **SE*cube*™** Chip

The single chip embeds three hardware components: a powerful processor, a flexible FPGA, and an EAL5+ certified smart card.

[11] cfr. https://www.secube.eu/site/assets/files/1145/secube_datasheet_-_r7.pdf

3.1.1 The Processor

The processor adopted within the **SE*cube*™** is the STM32F429, produced by ST Microelectronics™ [12], which includes a high-performance ARM Cortex M4 RISC core and provides the following features:

- 2 MB[13] of Flash memory
- 256 KB 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 implemented 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.

3.1.2 The FPGA

The FPGA element, a Lattice MachXO2-7000 device[14], is based on a fast, non-volatile logic array providing the following main features:

- 7,000 LUTs
- 240 Kbits of embedded block RAM
- 256 Kbits of user Flash memory
- Ultra low-power device

[12] http://www.st.com/en/microcontrollers/stm32f4-series.html?querycriteria=productId=SS1577
[13] For this document:
  - 1 KB = 2^{10} Bytes
  - 1 MB = 2^{20} Bytes
  - 1 GB = 2^{30} Bytes

[14] http://www.latticesemi.com/view_document?document_id=38834

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 Gb/s.
As outlined in Figure 6, 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 Gb/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.

3.1.3 The SmartCard

The third component of the **SE*cube*™ Chip** is a SLJ52G EAL5+ certified security controller by Infineon[15], hereafter named smartcard, that provides the following features:

- ISO7816 interface
- JavaCard Platform, Global Platform 2.2
- 128 KB Flash
- EC, ECDH up to 521 bit (HW accelerator)
- RSA up to 2 Kb (HW accelerator)
- AES128/192/256 (HW accelerator)

As outlined in Figure 6, within the **SE*cube*™ Chip**, the SmartCard is connected to the CPU via a standard ISO7816 interface.
The smartcard does not expose any interface outside the chip. This architectural solution provides high-grade and certified security functionalities behind a simpler and easy-to-use application interface.

[15] https://www.infineon.com/cms/en/product/security-smart-card-solutions/secora-security-solutions/secora-id-security-solutions/slj52gdaxxxcx/

3.1.4 On-chip connections

The Figure 6 summarizes the interconnections between the three components of **SE*cube*™** and their interfaces with the external.

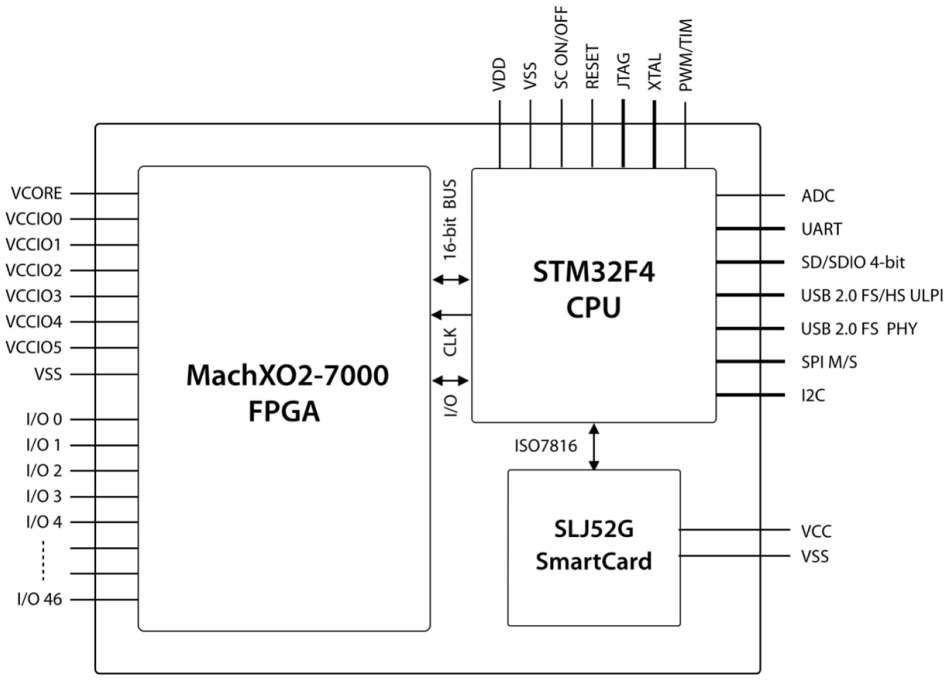


Figure 6: The **SE*cube*™** Hardware Architecture.

3.2 The SE*cube*™ Devkit

The **SE*cube*™** DevKit is an open development board (Figure 7) designed to support developers to integrate the **SE*cube*™** Chip in their hardware and software projects.

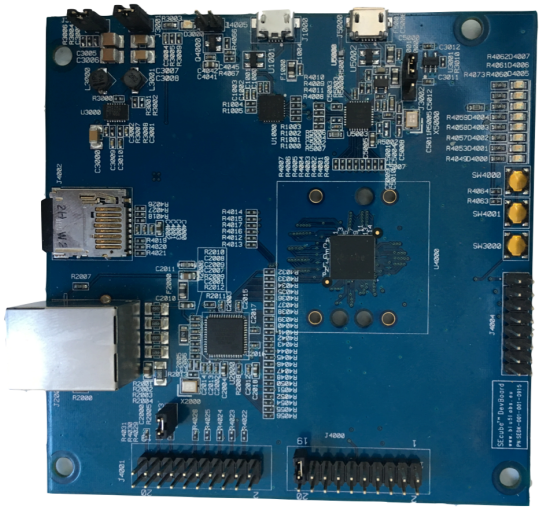


Figure 7: The **SE*cube*™** DevKit.

A floor planning view of the board is in Figure 8, whereas schematic can be found in Appendix B. The board is equipped with several interfaces and peripherals, including:

- USB 2.0 High-Speed (J5000)
- USB 2.0 to UART (J1000)
- microSD card (J4002)
- Ethernet 10/100 socket (J2000)
- Switches and LEDs (SW4000, SW4001, SW3000, LED0, ..., LED7)
- **SE*cube*™** embedded FPGA and CPU GPIOs (J4004, J4000)
- **SE*cube*™** embedded CPU JTAG (J4001).

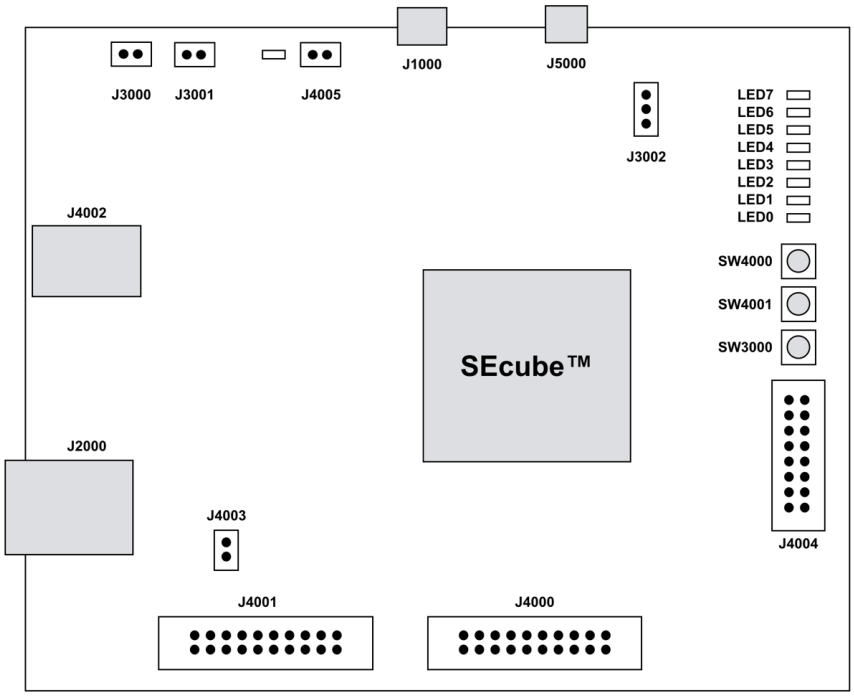


Figure 8: Floor planning view of the **SE*cube*™** DevKit.

The board is directly powered by one of the 2 micro USB connectors. The jumper 3002 selects the connector to be used to power the board (pins 1-2 select J5000, pins 2-3 select J1000). The **SE*cube*™** DevKit allows connecting two power supply lines and measuring the related power consumption, through the following jumpers:

- J3000: 1V2 power supply line
- J3001: 3V3 power supply line.

The power supply of the SmartCard embedded into **SE*cube*™** is controlled by a dedicated pin. Nevertheless, the jumper J4005 allows to bypass this control and power the embedded smartcard permanently.
The jumper J4003 allows a direct control of the **SE*cube*™** reset pin via the JTAG interface.

3.2.1 How to get it

The **SE*cube*™** DevKit can be ordered online[16].
Your purchase should comprise the following items (Figure 26):

- The **SE*cube*™** DevKit
- USB 2.0 A to Mini-B cable.

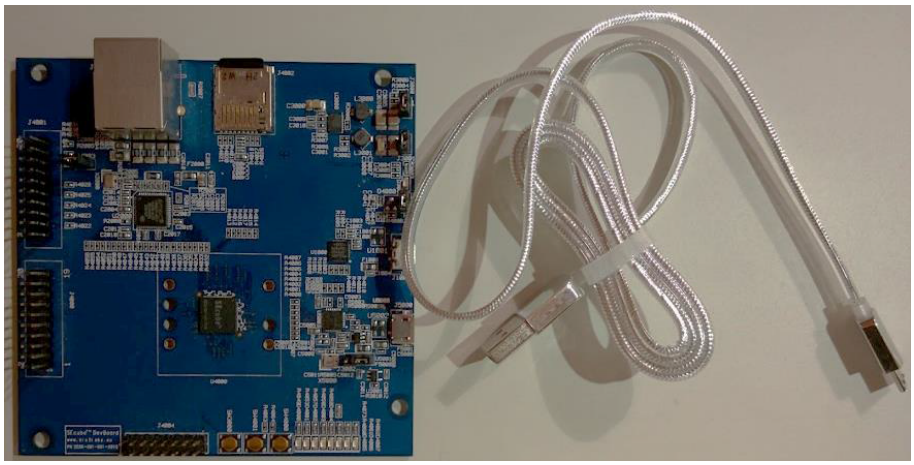


Figure 9: Components purchased with the **SE*cube*™** DevKit.

3.3 The USE*cube*™ Stick

As shown in Figure 10, the **USE*cube*™** Stick is a device characterized by a traditional USB form factor, a very effective way to deploy the **SE*cube*™** functionalities through a USB 2.0 High-Speed interface. As always, in fact, this device is based on the **SE*cube*™** Chip.

Figure 10: The **USE*cube*™** Stick.

From the hardware point of view, the **USE*cube*™** Stick is designed as part of the **SE*cube*™** DevKit. This allows the developers to migrate in a very fast way from the development board to the Stick and be ready for a market deployment.
The **USE*cube*™** Stick is compatible with any Operating System and the **SE*cube*™** functionalities are easily exposed to applications and services without installing any driver.
Since the **USE*cube*™** Stick storage capability is based on a microSD card, both the size and the speed can be tuned per the user requirement and can be changed at any time, just replacing the

[16] www.secube.eu

microSD, without buying a new **USE*cube*™ Stick**.
The microSD card socket is embedded in the USB connector. As shown in Figure 11, this solution allows to save space making the **USE*cube*™ Stick** very compact and, at the same time, dust and water-resistant.
Since the **USE*cube*™ Stick** is not provided with the JTAG interface, to inject the firmware previously developed and tested on the **USE*cube*™ DevKit**, all the devices come with an embedded secure boot loader.

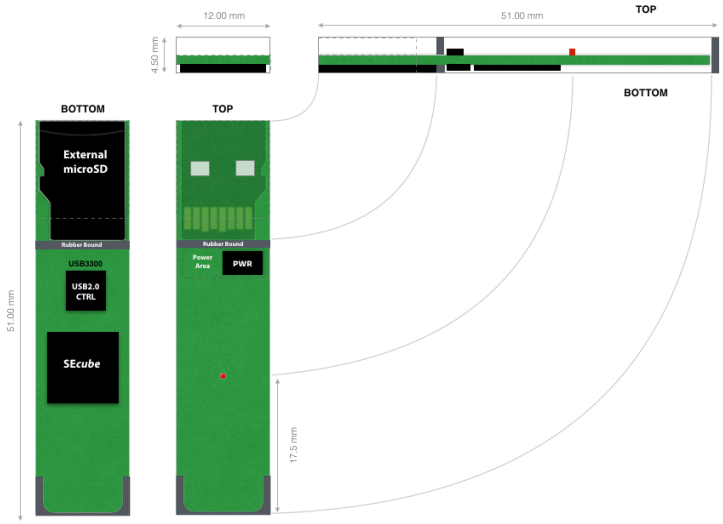


Figure 11: **USE*cube*™ Stick** internal structural details.

Figure 12: The **USE*cube*™ Stick** plugged into a laptop.

3.3.1 How to get it

The **USE*cube*™ Stick** can be ordered online[17].
Your purchase should comprise various items, depending on which purchase option you choose.

[17] www.secube.eu

4 The Software Architecture of the Platform

The **SE*cube*™** Open Source Software Architecture is structured in several *Abstraction Layers*, as summarized in Figure: 13, usually referred to as *L0*, *L1*, *L2*, *L3*, and *Applications*, respectively. At each *Abstraction Layer*, several sets of APIs are provided. At each level, each component (but the lowest one) represents a "service" for the upper level and relies on "services" provided by lower levels. The **SE*cube*™** -Side APIs are executed on the embedded processor of the **SE*cube*™** -based hardware device (e.g., the **USE*cube*™**), whereas at the External-Side the software is tailored for existing devices (e.g., laptops or Desktop PC) that see the **SE*cube*™** hardware as an external peripheral which exposes the **SE*cube*™** -Side APIs. In this scenario, **SE*cube*™** acts as a powerful coprocessor providing a secure and fully controlled execution environment. The *External-Side* Libraries are designed to be scalable, i.e., for dealing with multiple devices, and portable on different Operating Systems, thus limiting the usage of and isolating platform-dependent modules. They practically run on top of the host OS, directly relying on the OS System calls. To improve portability and migrations, the libraries are organized in such a way that all the OS-dependent submodules (e.g., communication interface, file system, etc.) are easily identifiable. All the source code is released under GPLv3 license[18].

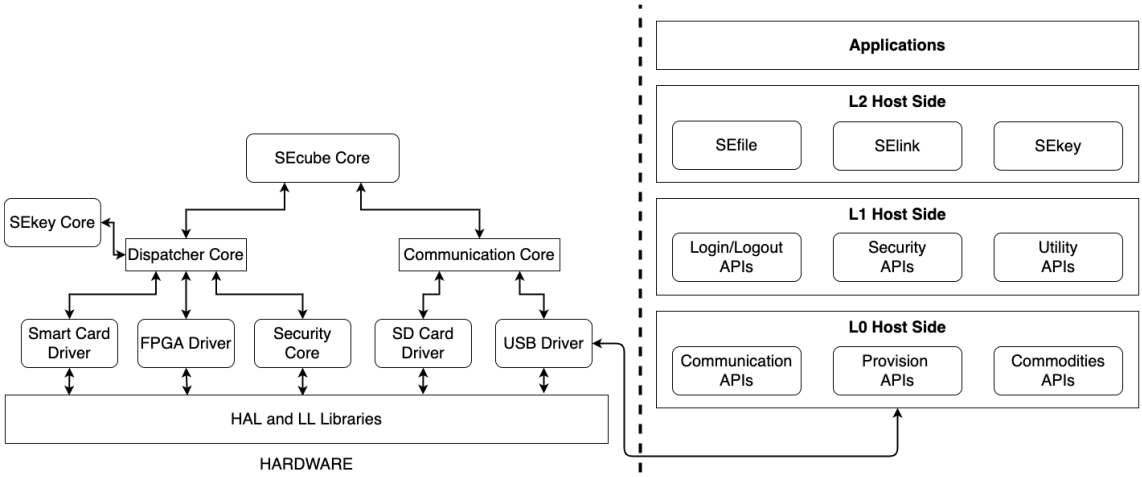


Figure 13: Software Architecture of **SE*cube*™** .

[18] https://www.gnu.org/licenses/gpl-3.0.en.html

4.1 Device-Side Libraries

In this section are presented some of the most significant Device-Side APIs, a more detailed documentation and the full list of the APIs can be found in the SDK package available online[19].

4.1.1 SE*cube*™ Core

```
uint16_t echo(uint16_t req_size, const uint8_t* req, uint16_t*
   resp_size, uint8_t* resp)
```

With this function, the **SE*cube*™** echoes back any data it receives from the host PC.

```
uint16_t factory_init(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

This is the function that initializes the serial number of the **SE*cube*™** .

4.1.2 Dispatcher Core

```
uint16_t key_edit(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Insert or delete a cryptographic key. This works only for keys that are not used for special purposes (i.e. keys used to encrypt data internally used by **SE*cube*™** libraries at L2 and L3 levels) and that are outside of the scope of **SE*key*™** , the KMS for the **SE*cube*™** device. The keys are always stored in the internal flash memory of the **SE*cube*™** ; similarly, they are always deleted from the same internal flash memory.

```
uint16_t key_find(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Search for a key with a given ID value into the flash memory of the **SE*cube*™** device.

```
uint16_t load_key_ids(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

With this function, the **SE*cube*™** scans its internal flash memory looking for keys. A list of key metadata (key ID and key length) is populated, then the list is sent back to the host PC that issued the request. Depending on who is issuing the request on the host side (i.e. the **SE*key*™** KMS or a user that manually manages his keys) some keys may not be returned in the list, this depends if you choose to handle keys manually (there is a specific key ID range for that) or automatically (with **SE*key*™** , there is a specific range of IDs also in this case).

```
uint16_t challenge(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Generate a login challenge on the device, to be used by the login procedure of the host towards the **SE*cube*™** .

```
uint16_t login(uint16_t req_size, const uint8_t* req, uint16_t*
   resp_size, uint8_t* resp)
```

Respond to the challenge and complete the login of the host towards the **SE*cube*™** .

```
uint16_t logout(uint16_t req_size, const uint8_t* req, uint16_t*
    resp_size, uint8_t* resp)
```

[19] https://www.secube.eu/resources/open-sources-sdk/

Logout from the **SE*cube*™** device.

4.1.3 Communication Core

```
int32_t se3_proto_recv(uint8_t lun, const uint8_t* buf, uint32_t
    blk_addr, uint16_t blk_len)
```

Function to process data incoming from the host PC. This is low level, it is not aware of the actual meaning of the data coming from the host.

```
int32_t se3_proto_send(uint8_t lun, uint8_t* buf, uint32_t
   blk_addr, uint16_t blk_len)
```

Process data that must be sent to the host PC. This is low level, it is not aware of the actual meaning of the data.

4.1.4 Smart Card Driver

This set of functionalities shall provide all the APIs necessary for interacting whit the SmartCard. Due to the very specific expertise and know-how required to develop low-level security modules for smart cards, the Smart Card Driver is still to be implemented.

4.1.5 Security Core

```
uint16_t crypto_init(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Initialize a cryptographic context. Low level functionality directly used to implement L1 level APIs.

```
uint16_t crypto_update(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Use a cryptographic context. Low level functionality directly used to implement L1 level APIs.

```
uint16_t crypto_list(uint16_t req_size, const uint8_t* req,
   uint16_t* resp_size, uint8_t* resp)
```

Get a list of available crypto algorithms implemented on the device (i.e. AES-256, SHA-256).

4.1.6 SD Card Driver

This driver provides all the functionalities required for communicating with the SD Card connected to the **SE*cube*™** .

4.1.7 USB Driver

This driver provides all the functionalities required for interacting with the USB connection. The **SE*cube*™** needs to know when something have been written through the USB connection (on the SD Card) in order to check if it is a request from the Host or other type of data.

4.1.8 SE*key*™ Core

This is the device-side implementation of the software that is required to interact with the **SE*key*™** Key Management System. Each of these functions is bounded to a specific function on the host-side.

4.2 Host-Side Libraries

4.2.1 L0 Libraries

This level implements the basic functionalities to communicate with the **SE*cube*™** device. This layer mainly includes 3 families of APIs:

- **Provisioning** APIs
- **Communication** APIs
- **Commodities** APIs

The performed functionalities include, among the others:

- sending/receiving command and data packets from/to the device
- segmenting raw data streams into protocol-compliant packets
- low-level error management
- low-level data manipulation in dealing with possible endianness mismatches between the host side and the **SE*cube*™** embedded processor.

Provisioning APIs

```
uint16_t L0FactoryInit(const std::array<uint8_t, 32>& serialno)
```

It is used to initialize the device, as well as its serial number. Once initialized, the serial number of the **SE*cube*™** cannot be changed anymore. The serial number is, by default, empty (all 0s). The serial number must be initialized providing an array of bytes, where the value of each byte must belong to the ASCII range of alphanumeric characters. A similar `L1FactoryInit()` API is available so that it can be used by software that does not require to deal with the L0 level.

Communication APIs

```
void L0Open()
```

Open the communication with the **SE*cube*™** device. This API does not need to be called explicitly in order to use the **SE*cube*™** .

```
void L0Close()
```

Close the communication with the **SE*cube*™** device. This API does not need to be called explicitly in order to use the **SE*cube*™** .

```
void L0TXRX(uint16_t reqCmd, uint16_t reqCmdFlags, uint16_t
   reqLen, const uint8_t* reqData, uint16_t* respStatus,
   uint16_t* respLen, uint8_t* respData)
```

It is the main function for communicating with the **SE*cube*™** device. It sends a packet of data containing a command and the relative parameters and then reads the response written by the **SE*cube*™** .

```
uint16_t L0Echo(const uint8_t* dataIn, uint16_t dataInLen,
   uint8_t* dataOut)
```

It is the host-side API to send a packet of data to the device, which should then reply with the same data. Login is not required to communicate with the device via the echo service.

Commodities APIs

To exploit the security functionalities exposed by the **SE*cube*™** device, the Host must:

- Discover whether at least a valid and initialized device is plugged
- Choice the desired device among a list if more than just one device is plugged
- Discover the serial number of the desired device.

To implement the discovery functionality, the L0 service leverages on iterators, objects that enables a programmer to traverse a container, particularly lists.

```
void L0DiscoverInit()
```

It is the host-side API to initialize the iterator object to traverse a list of **SE*cube*™** devices.

```
bool L0DiscoverNext()
```

It is used to traverse the list, moving the iterator object anytime towards the following element of the list.

4.2.2 L1 Libraries

This level relies on L0 to provide the basic APIs for implementing secure applications, including multi-factor login, cryptographic and hashing algorithms, manual key management, and basic device-based diagnostic. The services exposed at this level includes several APIs to manage:

- login/logout
- security
  - manual key management (add key, delete key)
  - cryptographic algorithms
  - hashing algorithms
- device-based diagnostic
  - find a key inside the device
  - list all keys inside the device
  - list supported algorithms

Login/Logout APIs

```
void L1Login(const array<uint8_t, 32>& pin, se3_access_type
   access, bool force)
```

Login on the **SE*cube*™** device as administrator or user. Depending on the access level, different features of the firmware may or may not be available. Each access level is protected by a specific PIN code; both PIN codes are set to all zeroes by default. Apart from the Echo command, the other functionalities of the **SE*cube*™** can be accessed only after login.

```
bool L1GetSessionLoggedIn()
```

It is used to check whether the login has been executed (returns true) or not (returns false).

```
se3_access_type L1GetAccessType()
```

It is used to check the access privilege to the **SE*cube*™** (i.e., admin privilege or user privilege) after the login to the device.

```
void L1Logout()
```

Logout from the **SE*cube*™** .

Security APIs

```
void L1CryptoInit(uint16_t algorithm, uint16_t mode, uint32_t
   keyId, uint32_t* sessId)
```

Initialize the crypto context needed to perform an encryption or decryption operation. This is a low level function to use the crypto features of the SEcube. It can be ignored, we suggest using `L1Encrypt()`, `L1Decrypt()` and `L1Digest()` instead.

```
void L1CryptoUpdate(uint32_t sessId, uint16_t flags, uint16_t
   data1Len, uint8_t* data1, uint16_t data2Len, uint8_t* data2,
   uint16_t* dataOutLen, uint8_t* dataOut)
```

Use the crypto context initialized by `L1CryptoInit()` to perform the corresponding action on a specific portion of data. This is a low level function to exploit the crypto features of the SEcube. It can be ignored, we suggest using `L1Encrypt()`, `L1Decrypt()` and `L1Digest()` instead.

```
void L1Encrypt(size_t plaintext_size, shared_ptr<uint8_t[]>
   plaintext, SEcube_ciphertext& encrypted_data, uint16_t
   algorithm, uint16_t algorithm_mode, uint32_t key_id)
```

Encrypt some data according to a specific algorithm and mode (i.e., AES-256-CBC), using a specific key. It automatically takes care of every low-level detail (i.e., padding) and stores the result into the `SEcube_ciphertext` object passed as third parameter. Data encrypted with this API must be decrypted with the L1Decrypt() API.

```
void L1Decrypt(SEcube_ciphertext& encrypted_data, size_t&
   plaintext_size, shared_ptr<uint8_t[]>& plaintext)
```

Decrypt some data (first parameter) after they have been encrypted with `L1Encrypt()`. The result is stored in the last parameter, the length of the result is stored in the second parameter. The encrypted data to be processed are passed in input as first parameter. Low level details (i.e., padding and signature match if required) are automatically managed.

```
void L1Digest(size_t input_size, shared_ptr<uint8_t[]>
   input_data, SEcube_digest& digest)
```

Compute the digest of some data. Low level details are automatically managed by the `SEcube_digest` object.

```
void L1GetAlgorithms(vector<se3Algo>& algorithmsArray)
```

Retrieve the list of algorithms supported by the device.

```
void L1SetAdminPIN(array<uint8_t, 32>& pin)
```

Set the PIN to login on the **SE*cube*™** with administrator privilege.

```
void L1SetUserPIN(array<uint8_t, 32>& pin)
```

Set the PIN to login on the **SE*cube*™** with user privilege.

```
void L1KeyEdit(se3Key& k, uint16_t op)
```

Function to manually add or remove keys in the **SE*cube*™** device. This function is intended to be used to manually manage cryptographic keys; automatic key management can be performed through **SE*key*™** . Depending on the value of the second parameter (`L1Commands::KeyOpEdit`

for info) a new key can be stored on the **SE*cube*™** device (specifying manually its value or demanding a secure key generation to the internal TRNG of the **SE*cube*™**) or an existing key can be deleted from the device.

```
void L1KeyList(vector<pair<uint32_t, uint16_t>>& keylist)
```

List the keys stored inside the memory of a **SE*cube*™** device. This function is dedicated to manual key management, therefore only the keys that are not managed by **SE*key*™** will be listed. The result is a list of pairs, the first element is the ID of the key (an unsigned integer on 32 bits), the second element is the length of the key (in bytes).

```
void L1FindKey(uint32_t key_id, bool& found)
```

Check if the key with the specified ID is stored inside the **SE*cube*™** .

```
void L1SelectSEcube(array<uint8_t, 32>& sn);
```

Select the **SE*cube*™** with the serial number passed in input out of multiple **SE*cube*™** devices. The selected **SE*cube*™** will be the one that will be used to perform other actions based on L1 APIs and higher level APIs relying on L1.

```
void L1SelectSEcube(uint8_t indx);
```

Select the **SE*cube*™** with the index passed in input out of multiple **SE*cube*™** devices. The selected **SE*cube*™** will be the one that will be used to perform other actions based on L1 APIs and higher level APIs relying on L1. The index is the number assigned to the **SE*cube*™** by L0 API named `GetDeviceList()`.

4.2.3 L2 Libraries

This level relies on L1 to provide a set of APIs for implementing secure functionalities at a higher level of abstraction. In particular, L2 APIs offer optimized and easy-to-use functionalities to ease the development of applications that create and manage entities that are fully protected (i.e., authenticated and confidential), without being forced to understand all the low-level details. The provided APIs include a Key Management System (**SE*key*™**), a data-at-rest protection facility (**SE*file*™**), and a data-in-motion protection facility (**SE*link*™**), briefly outlined in the sequel.

4.3 SE*file*™

In its kernel space, any operating system provides an abstraction layer, used to separate file system generic operations from their actual implementation. This is performed by defining a clean Virtual File System (VFS) interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems locally mounted. Even though data protection is provided at this level of abstraction by means of a dedicated security engine, it is still possible to exploit existing flaws in the user application or in the file system implementation in order to access to confidential data.
Cryptography is usually identified as a possible solution to this problem. However, the adoption of cryptography implies that the application and the file system must deal with encrypted data. Managing encrypted data is difficult for two reasons:

- Encryption and decryption processes must be performed using a key that must be stored somewhere;
- Sooner or later, the decrypted value of the data must be stored at least in RAM memory in order to perform any action on those data.

In order to mitigate the problems that come with encryption and, at the same time, increase the overall security of the data and of the entire system, dedicated hardware devices can be used to encrypt the data, therefore guaranteeing data confidentiality, and to keep the cryptographic keys used for encryption inside a secure environment where they cannot be accessed without proper authentication. Thanks to this approach, in general, a dedicated secure device can guarantee data protection also when the host machine is compromised.
SE*file*™ is a library that implements a file system interface specific for the **SE*cube*™** device. It uses the features exposed by L1 libraries (cfr. 4.2.2) and other functionalities of the device in order to offer to developers and users a complete file system interface that natively operates on files that are always encrypted. **SE*file*™** replaces the standard file system interface calls (i.e. `open()`, `close()`, `read()`, `write()`) with secure primitives that are totally similar in terms of semantics, but very different in terms of implementation, because they natively work with encrypted files.
SE*file*™ is only compatible with files that have been specifically created and encrypted with the APIs of **SE*file*™** itself. Therefore, it cannot be used with files encrypted with a generic software or device. Moreover, thanks to the **SE*cube*™** device, the keys used to encrypt the data are phisically separated from the data. Encrypted data are stored on the disk of the host machine, the keys are protected inside the internal flash memory of the **SE*cube*™** . When the **SE*cube*™** is physically unplugged from the host machine, there is no way to access to the keys required to decrypt the data.
SE*file*™ has been developed having in mind the need to ensure both simplicity of usage and security for data at rest; it allows secure storage, retrieval and usage of information that need to be protected. For this reason, **SE*file*™** guarantees confidentiality, integrity and authentication of data. **SE*file*™** is deeply analyzed in Chapter 6.

4.4 SE*link*™

Rather than securing files in the data at rest scenario, the **SE*link*™** library is focused on securing data in the data in motion scenario.
SE*link*™ is a very simple library with few APIs that allow any application to encrypt data using powerful algorithms (i.e. AES-256-HMAC-SHA-256); **SE*link*™** also offers specific functions to serialize the data in order to transfer them using any protocol (i.e. TCP).
Basically, **SE*link*™** allows developers and users to implement a secure end-to-end communication channel between multiple parties. It is indipendent from the protocol that is needed to transmit the data, but it requires developers to modify the source code of the software that needs to send and receive encrypted data (similarly to **SE*file*™**).
Consider a scenario where two users are communicating by means of an open source messaging client. Even if the client may offer security features (i.e., end-to-end encryption), the user may not trust the company distributing the software, or the company implementing the infrastructure to make the client work. In that case, **SE*link*™** can be used to encrypt in advance any data that must be sent, granting confidentiality even in untrusted environments.
Notice that **SE*link*™** grants confidentiality, integrity and authentication of all data. **SE*link*™** is deeply analyzed in Chapter 7.

4.5 SE*key*™

SE*key*™ is a library used to implement a simple Key Management System (KMS), whose target is to allow an easy and secure management of the cryptographic keys stored in the **SE*cube*™** device. **SE*key*™** offers a set of APIs that can be mainly used to handle the entire lifecycle of cryptographic keys, from creation to distribution and usage. Just like any other **SE*cube*™** related software, **SE*key*™** has been developed according to the *security-by-design* principle.

Other than the management of the cryptographic keys, the KMS also allows to a single administrator to decide how the keys should be distributed and used. Depending on the needs of the specific environment where **SE*key*™** needs to be deployed, the administrator adds users to the key management system, users are then organized in groups, each one characterized by a specific security policy depending on the security requirements of the group.
SE*key*™ is an essential component of the **SE*cube*™** software ecosystem because it removes the issue of key management from the set of skills that are required to each individual user of the **SE*cube*™** device. L1 APIs already allow a manual management of cryptographic keys but they are intended to be used only in very simple scenarios, for example for testing and debugging purposes. A manual approach to key management is usually unfeasible, especially in environments where many keys need to be deployed in order to secure huge volumes of encrypted data.
Consider a hard drive storing thousands of files that have been encrypted with **SE*file*™** . Even though **SE*file*™** automatically takes care of memorizing the unique identifier of the keys, the absence of a KMS would required to the user to keep track of every single cryptographic key because it may happen, for example, that a key is compromised because of a cyber attack. Reacting "manually" to such an event would be extremely complicated, because it requires to notify about the cyber attack every single person that has access to the compromised key. **SE*key*™** automatically implements all these procedures, along with many others (i.e., key deactivation, key deletion, key usage suspension, etc.).
I n conclusion, a KMS such as **SE*key*™** is much more flexible, scalable and secure with respect to a manual approach. **SE*key*™** is deeply analyzed in Chapter 8.

4.6 L3 Libraries

This level relies on L2 services to develop secure applications. For example, the Secure Database library for the **SE*cube*™** is based on **SE*file*™** and it is natively compatible with **SE*key*™** . It is also based on the SQLite database engine, it basically implements an encrypted SQL database where the security features are completely transparent to the users because the database itself can be easily managed using the standard SQLite C interface. The Secure Database library is described in Chapter 6.6.
As of today, no other L3 library has been developed.

5 Exploiting the internal FPGA

As already presented, among its hardware resources the **SE*cube*™** offers a powerful flexible FPGA by Lattice Semiconductor.
The FPGA is assumed to be configured to include one or a set of *IP Cores*. Such cores implement a set of functions, as fast computing of dedicated algorithms or enabling the device to expand its interfacing capabilities. Each core present onto the FPGA should have its own *SW Driver* included in the SDK. The driver is in charge of the communication between the processor and the FPGA.

5.1 FPGA-CPU connection

The processor and the FPGA within the **SE*cube*™** platform are connected via a bus as depicted in Figure 14.
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 original intent of the pin configuration within the **SE*cube*™** is treating 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[20], as in this case. With this configuration, each pin assumes a specific behaviour, its value and transitions being directly managed by the FMC.
The available pins within the bus are:

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

Actually, it is possible to consider address and data pins as normal serial lines, and it is possible to program the interconnections as normal GPIO, to be driven in *bit-banging* for implementing a custom communication protocol. Of course, a proper software driver is to be written for supporting this communication, unlike what happens for the FMC, which is already supported by the hardware.

[20] 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/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

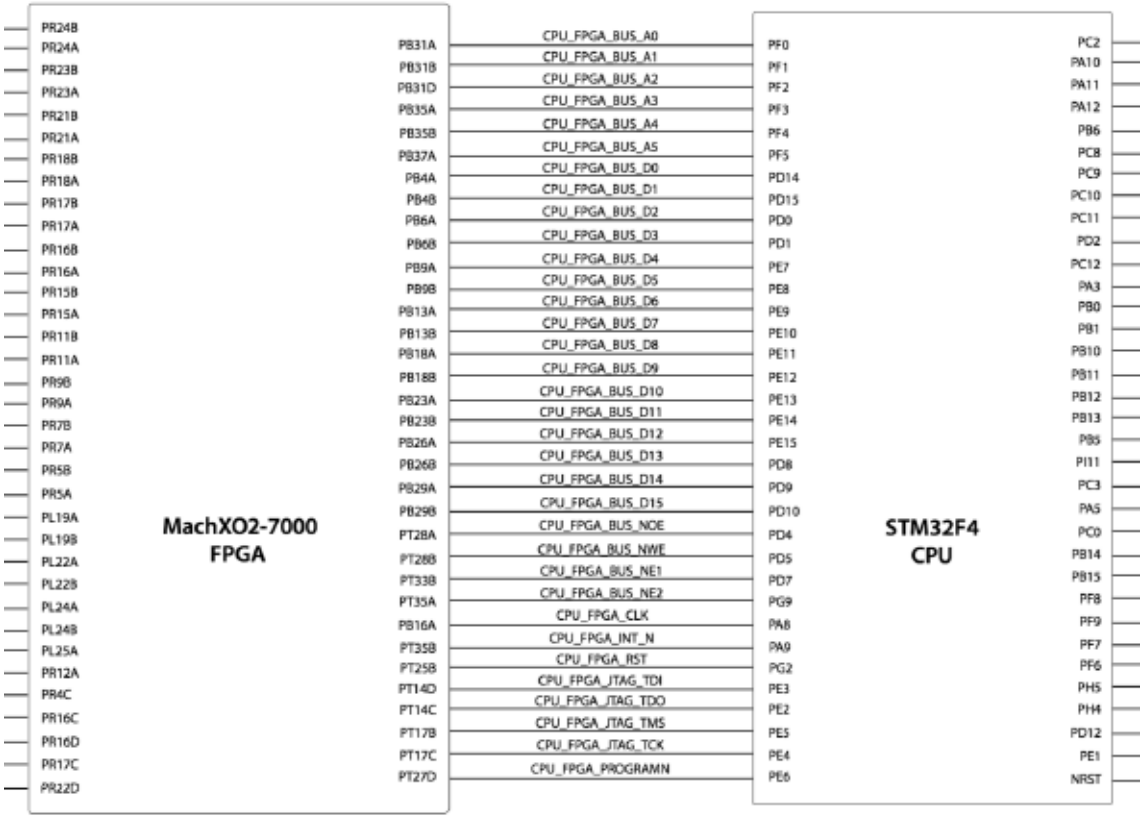


Figure 14: FPGA-CPU connections within **SE*cube*™**

5.2 The Flexible Memory Controller

The Flexible Memory Controller (FMC) present on the STM32F429 processor is a useful block that can be programmed to interface with an external memory. In the **SE*cube*™** case, the FPGA can be seen as a block of external PSRAM (Pseudo-Static RAM), as already mentioned. According to the schematic presented in the Reference Manual, the FMC disposes of the following PSRAM interface:

- Address - 26 pins (FMC_A25:0)
- Data - 32 pins (FMC_D31:0)
- Bank Enable - 4 pins (FMC_NE1:4)
- Output Enable - 1 pin (FMC_NOE)
- Write Enable - 1 pin (FMC_NWE)

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. Bits [25:0] of the same bus are instead interpreted as the actual external memory address.
The pinout seems to allow only a 6-bit address by the CPU. Actually, it is possible to program the FMC data bus in *Turnaround* mode, so that it can forward to the external memory both the address and the data, allowing a greater addressing space.
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.

5.2.1 Configuring the FMC

Within the Open SDK, the subroutine encharged of initializing the Flexible Memory Controller is `MX_FMC_Init()`. This function fills the members of 2 structures, `Timing` and `Init`, both for reading and for writing, which give the FMC the information necessary to being initialized.
`Init` provides all information on the type of external memory connected, the data width, whether it is synchronous or asynchronous, possible support for burst modes, wait cycles, bus turnarounds and so on. Through `Timing` instead, the programmer sets the *setup* and *hold* times for address and data, the possible number of turnaround cycles and one of the *access modes* available, each one having its own specific timing diagram with respect to the fundamental FMC signals.
Just as a matter of example, Figures 15 and 16 below show the timing diagram of Access Mode A, both for writes and reads.

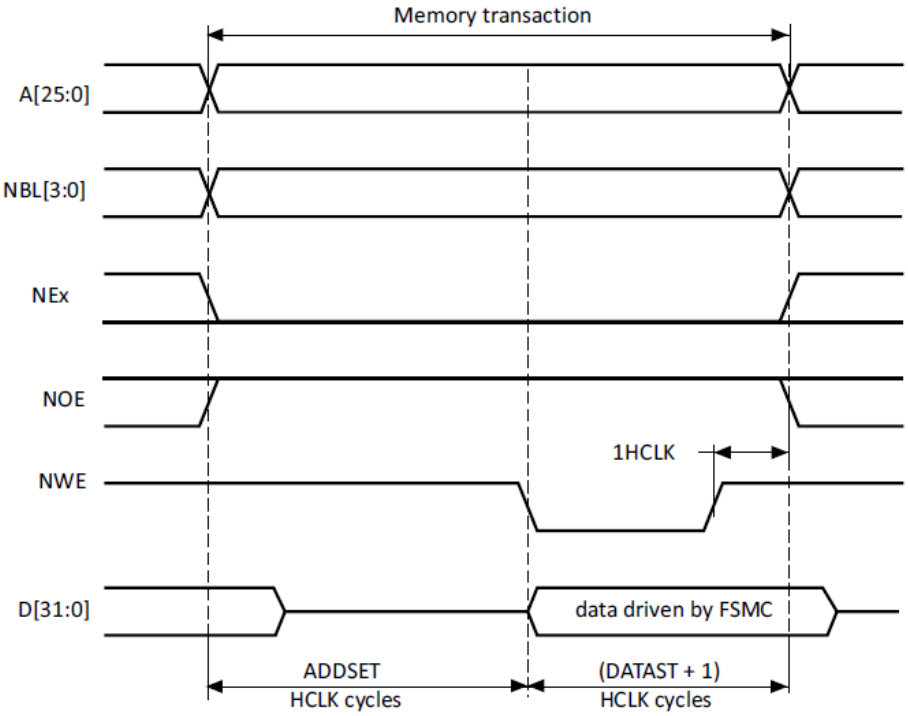


Figure 15: Access Mode A timing diagram for WRITE

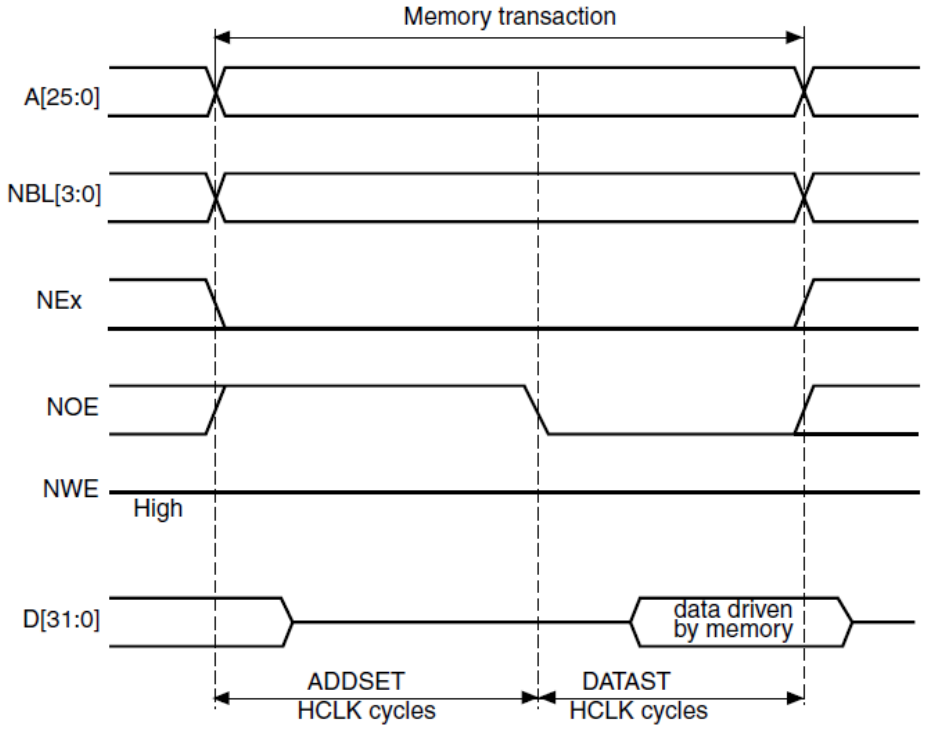


Figure 16: Access Mode A timing diagram for READ

When NEx is low, a memory operation is taking place. The address is forwarded and the 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 the 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, decided through setting the `Timing` initializing structure. Times are to be carefully set according to the specification of the interfacing memory and the ratio between the CPU and the memory speed, in this case the FPGA design maximum speed, provided by the synthesis tools.

Once the memory controller is configured, the external GPIO pins of the microcontroller have to be set up to host the FMC interface. While for fundamental control pins the task is completed by the GPIO initializer function `MX_GPIO_Init()`, a sample of settings for the data/address interface are included in the `HAL_FMC_MspInit()` call, which is invoked inside the `MX_FMC_Init()`. Pin initialization is done through several calls to the HAL function `HAL_GPIO_Init()`, which consolidate the settings contained in a `GPIO_InitStruct` structure. This allows to set preferences for a single or a group of GPIO pins, choosing the *mode* (input, output, alternate), the *pull* (default pullup, pulldown or no pull), the maximum speed at which they have to react, and the module which controls them. Here is an example of how GPIO pins should be configured for supporting the FMC, as it is preset in `HAL_FMC_MspInit()` (cfr. Figure 14 for pin reference).

```
/*Configure GPIO pins : PF0 PF1 PF2 PF3 PF4 PF5 */
GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 |
   GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF12_FMC;
```

```
HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);

/*Configure GPIO pins : PE7 PE8 PE9 PE10 PE11 PE12 PE13 PE14
   PE15 */
GPIO_InitStruct.Pin = GPIO_PIN_7 | GPIO_PIN_8 | GPIO_PIN_9 |
   GPIO_PIN_10 | GPIO_PIN_11 | GPIO_PIN_12 | GPIO_PIN_13 |
   GPIO_PIN_14 | GPIO_PIN_15;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF12_FMC;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/*Configure GPIO pins : PD8 PD9 PD10 PD14 PD15 PD0 PD1 PD4 PD5
   PD7 */
GPIO_InitStruct.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10 |
   GPIO_PIN_14 | GPIO_PIN_15 | GPIO_PIN_0 | GPIO_PIN_1 |
   GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_7;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF12_FMC;
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);

/*Configure GPIO pin : PG9 */
GPIO_InitStruct.Pin = GPIO_PIN_9;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF12_FMC;
HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
```

5.3 Configuring the FPGA

5.3.1 Programming through JTAG interface

To be used, the FPGA has first to be loaded with a design, transmitted in the form of a *bitstream*. A bitstream is a long array of bytes, which encodes the content of every Look-Up Table (LUT) present onto the FPGA, and how they must be connected each other (routing information). The bitstream is produced by the synthesis tool, which can be instructed to output a C-compatible file, so that the arrays can be statically saved within the application memory image of **SE*cube*™** as constant data.
Within the firmware source files, under “Project/Inc/Device”, an example of bitstream file (“TEST_FPGA.h”) can be found. The folder also contains a header file, “FPGA.h”, which must be included in a **SE*cube*™** device application which wants to exploit the FPGA. The corresponding source, “FPGA.c”, is under “Project/Src/Device”. This header contains the function `B5_FPGA_Programming()`, which must be called at the beginning of the application to program the FPGA. The function controls in bit-banging the JTAG interface between the CPU and the FPGA to put it in a programming state and to send the bistream bytes one after the other.
The FPGA programming phase needs the 5 pins of the JTAG interface to be correctly set through

apposite GPIO configuration. The GPIO initialization function `MX_GPIO_Init()` should then contain the following lines:

```
/*Configure GPIO pins : PE3 PE5 PE4 PE6 */
GPIO_InitStruct.Pin = FPGA_TCK_Pin | FPGA_TDI_Pin | FPGA_TMS_Pin
    | FPGA_PROGRAMN_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = FPGA_TDO_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(FPGA_TDI_GPIO_Port, &GPIO_InitStruct);
```

A detailed step-by-step guide to correctly initialize the FPGA can be found in Section 9.

5.3.2 Reset signal

A reset signal (CPU_FPGA_RST) has been allocated in the interface in order to give the CPU the possibility of bringing the architecture implemented on the FPGA to a known starting state, immediately after programming or, in general, whenever the design is to be restarted. The corresponding GPIO pin (PG2) can be configured as a normal output pin:

```
/* Set pin PG2 as reset for the FPGA */
GPIO_InitStruct.Pin = FPGA_RST_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
```

When a reset is to be asserted, the GPIO pin can be controlled in bit-banging through the dedicate HAL primitives:

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

5.3.3 Clock signal

The clock signal to be provided to the FPGA (CPU_FPGA_CLK) has been assigned to pin PA8 in the interface. The pin must be driven by the Reset and Clock Controller (RCC) module of the processor[21]. The FPGA can be fed with a submultiple of the nominal clock of the processor (HCLK), which is 180 MHz at maximum. The value of the clock divisor must be chosen in accordance with the maximum speed reachable by the synthesized architecture, indicated by the synthesizer. The following settings show how to give a 60 MHz output clock out from pin PA8.

```
/* Enable clock for GPIOA */
__HAL_RCC_GPIOA_CLK_ENABLE();
```

[21] For additional details, please refer to STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs Reference Manual, Chapter 6: https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

```
/* Enable clock for SYSCFG */
__HAL_RCC_SYSCFG_CLK_ENABLE();

/* Set pin PA8 as clock for the FPGA */
GPIO_InitStruct.Pin = GPIO_PIN_8;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
GPIO_InitStruct.Alternate = GPIO_AF0_MCO;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* Set MCO1 output = PLLCLK with prescaler 3 = 180 MHz / 3 = 60
   MHz */
__HAL_RCC_MCO1_CONFIG(RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_3);
```

The second parameter of the macro `__HAL_RCC_MC01_CONFIG()` 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.

5.4 Programming FPGA-based applications

The presence of the FPGA inside the **SE*cube*™ Chip** is certainly a plus for the programmer's possibilities. The CPU-FPGA system can be seen as a *processor-coprocessor* system, where the programmable logic is entrusted with a series of frequently used routines, among which encryption and hashing certainly stand out, while the control and decision-making part remains to the microcontroller. Besides the advantages brought by this parallelism, there is also the guarantee of reaching the necessary levels of determinism and performance, in the case of time-critical applications. The flexibility the FPGAs dispose of allows to keep up with innovation in communication standards: in fact, converters and controllers can be implemented for external interfaces which may be not natively present on the processor.
With the abstraction created by the employ of the FMC, the FPGA is seen by the processor as a normal peripheral, with a given set of shared locations starting from address `0x60000000` in the memory map. Thus, the communication with the architecture present onto the FPGA is basically composed of *reads* and *writes*, which form a master-slave protocol of queries and responses in a high-level view.
The communication can be held in polling or in interrupt mode. In polling mode, the master writes its input stream and after finishing awaits the end of the computation by continuously checking the status (which can be "something to communicate", or "nothing to communicate"). This is classically done with a continuous read of one of the shared locations. In interrupt mode, instead, the master send the slave its inputs and then continues its own computation, while the slave reads the input, performs its task and in the end triggers an interrupt request to the master.
For a correct communication, a complete driver to set up the communication should be composed of *two layers*:

- a high-level layer, composed of the API for managing the tasks of the IP core implemented (e.g., *encrypt, sign, send, receive*, etc.)
- a low-level layer, containing the low-level functionalities for the communication with the

FPGA, as initializing and resetting the FPGA, reading and writing a word on the shared storage, feeding the device with the clock, etc.

It is likely that more than one functionality is required from the FPGA, so that a single IP core is not enough. To accomplish this, a *central manager block* may be necessary to redirect CPU requests to the correct core, and to ensure that each communication with the CPU is exclusive with a single component (*transaction*). An example of such a system has been developed and is freely downloadable at https://www.secube.eu/resources/open-source-projects/, under the name **IP-core Manager for FPGA-based design**.

6 The SE*file*™ Library

6.1 Introduction

The present section provides a detailed presentation of the **SE*file*™** library available for the **SE*cube*™** *Open Security Platform*. The purpose of **SE*file*™** is to manage encrypted files with the **SE*cube*™** . In particular, **SE*file*™** allows to any software to work on encrypted files exploiting the features of the **SE*cube*™** while keeping the data constantly encrypted on disk.
Any OS provides an abstraction layer in its kernel space, used to separate file system generic operations from their implementation. This is performed thanks to a clean Virtual File System (VFS) interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally. Data protection is provided at this level of abstraction, by means of a dedicated security engine, hereinafter referred to as *secure layer*.
A possible approach is to develop a secure layer which operates in the user space (Figure 17 on the left). This approach allows the developer to provide security functionalities without modifying the underlying operating system, which it is not always permitted. On the other hand, those secure functions do not override the standard ones, instead proposing themselves as a secure alternative. An interesting feature in this case is given by the possibility to develop a portable layer, meaning that it is valid for different Operating Systems (OSs).
Typically, OSs vendors follow another approach, based on a security level lying under the virtual file system, hence not guaranteeing portability (Figure 17 on the right). The secure layer, in this case, is transparent to the application/user.

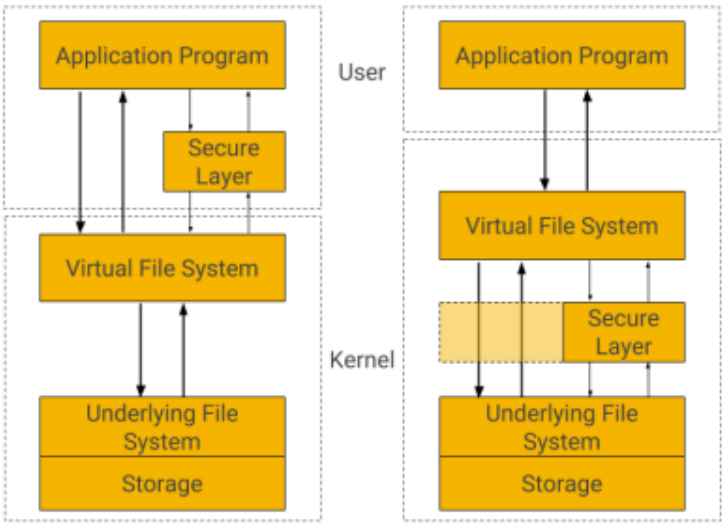


Figure 17: Secure Layer and Virtual File System: two different approaches.

In any case, whichever is the chosen approach, malicious user, or software, may still exploit existing flaws in the application accessing to the secure layer or even in the secure layer itself. A countermeasure to protect effectively the data, thus, resorts to hardware key management techniques applied to powerful embedded systems that can perform complex cryptographic operations while, at the same time, increasing the confidence of data security. A secure device can guarantee data protection also when the host machine is compromised.
SE*file*™ exploits the APIs Level L1 and other functionalities from the **SE*cube*™** device (Figure 18). It has been developed having in mind the need to ensure both simplicity of usage and security for *data at rest*: it allows secure storage, retrieval and usage of information that require to be

protected by means of encryption.

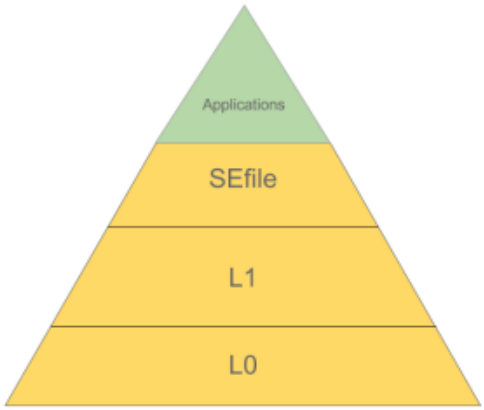


Figure 18: **SE*file*™** hierarchic organization.

Conceptually, **SE*file*™** targets any user that, by moving inside a secure environment, wants to perform basic operations on regular files. It must be pointed out that all encryption functionalities are demanded to the **SE*cube*™** in their entirety. In addition, **SE*file*™** does not expose to the host device details about what or where it is reading/writing data: thus, the host OS, which might be untrusted, is totally unaware of what it is writing.

6.2 Data Confidentiality

One of the most important considerations a Secure File System deals with is the way in which files are encrypted. A Secure File is made up of several sectors which are encrypted and signed (using AES256-HMAC-SHA256) in order to grant confidentiality, integrity and authenticity. The first sector is dedicated to the header, which provides information about the file itself (i.e., the ID of the key and of the algorithm used to encrypt the file, the length of the file, the name of the file, and other metadata) and contains padding if needed, while the other sectors contain the actual data of the file (Figure 19).

Figure 19: Secure File structure.

This structure has the great advantage of allowing data manipulation on parts of a file by interacting with a subset of its sectors. In this case, there is no need to decrypt and encrypt the whole file.

Therefore, the time overhead is considerably lowered, especially in the common case of a read or write operation involving a single sector or few sectors. However, it is important to notice that, when a sector of a file encrypted with **SE*file*™** is being used, the decrypted content of that sector will be stored in the RAM memory of the host machine to which the **SE*cube*™** is connected. This behaviour is necessary because, in the end, the software running on the host machine (i.e., a text editor) must be able to access to the actual content of the encrypted file. A simplified overview about how **SE*file*™** works is available in Figure 20.

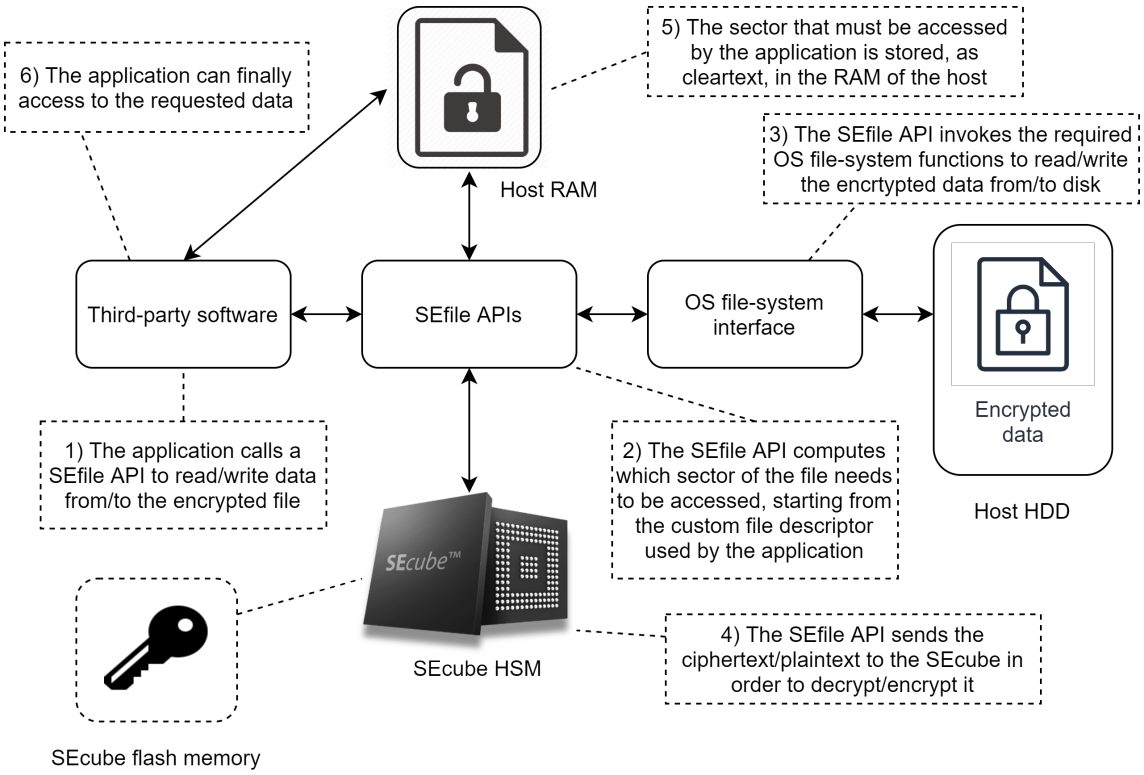


Figure 20: Simplified step-by-step **SE*file*™** overview.

6.3 Encryption Algorithm

The Advanced Encryption Standard (AES), also known by its original name *Rijndael*, is a specification for the encryption of data established by the U.S. National Institute of Standards and Technology (NIST) in 2001[22]. AES has been adopted by the U.S. government and is now used worldwide, becoming a de-facto standard for guaranteeing data confidentiality. It is a block cipher, since it is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation of blocks of fixed size, and is fast in both software and hardware.
However, a block cipher by itself is only suitable for the secure cryptographic transformation (encryption or decryption) of one fixed-length group of bits (i.e., a block). Then, a mode of operation is an algorithm that uses a block cipher to provide an information service such as confidentiality or authenticity. A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.
Most modes require a unique binary sequence, often called initialization vector (IV), for each encryption operation. The IV should be non-repeating and, for some modes, random as well. The initialization vector is used to ensure distinct ciphertexts are produced even when the same plain-

[22] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)". Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.

text is encrypted multiple times independently with the same key. Block ciphers have one or more block size(s), but during transformation the block size is always fixed. Block cipher modes operate on whole blocks and require the last part of the data to be padded to a full block if it is smaller than the current block size.
Currently, the open source APIs of the **SE*cube*™** (L1 and L0) support AES only as cipher algorithm (supported key sizes of 128, 192, 256 bits). **SE*file*™** leverages it by using the Counter (CTR) mode of operation. The simplest encryption mode is the Electronic Codebook (ECB) mode. The message is divided into blocks, and each block is encrypted separately.
Counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a "counter". The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. Today, CTR mode is widely accepted and any problems are considered a weakness of the underlying block cipher, which is expected to be secure regardless of systemic bias in its input[23].
This said, **SE*file*™** uses an encryption scheme as follows. Each sector, except the header, is encrypted using **AES-256-CTR**, meaning that each block cipher depends on an ascending counter which starts from a randomly selected initialization vector, generated using ad-hoc functions provided by the crypto engine of any OS.
The header sector, instead, is encrypted using **AES-256-ECB**, to be independent from any initialization vector. Moreover, the header sector is not entirely encrypted because, in order to make the **SE*file*™** library easily usable, each file must disclose the ID of the key and the ID of the algorithm that were used to encrypt the file itself. This approach is adopted to speed-up the interaction with the **SE*file*™** library, in fact anyone can read the first bytes of the header sector (because they are stored as clear-text) to easily find out which key (not its actual value) and which algorithm were used to encrypt it, then only the people with a **SE*cube*™** device containing that specific key will be able to decrypt the rest of the header and the other sectors. For example, given a security domain where all involved actors use the same key ID nomenclature (i.e., they all agree about the value of the key with ID equal to '1', '2', '3', and so on) the header of a certain file may contain the key ID '10' and the algorithm ID '3'; this means that only the owners of **SE*cube*™** devices containing the key with ID '10' will be able to decrypt the file (by using the correct algorithm, of course). Notice that, if someone else has got a **SE*cube*™** which stores a key with that same ID '10' but with a different key value, obviously that person will not be able to decrypt the file.

6.4 Data Authentication

On the other hand, there exists the problem of guaranteeing the integrity of the whole file against malicious attackers. Therefore, each sector is signed so the integrity and the authenticity of each one can be easily checked.

6.4.1 Algorithms

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).
SHA-3 uses the sponge construction, in which data is "absorbed" into the sponge, then the result is "squeezed" out[24]. In the absorbing phase, message blocks are XORed into a subset of the state, which is then transformed. In the "squeeze" phase, output blocks are read from the same subset

[23] Helger Lipmaa, Phillip Rogaway, and David Wagner. Comments to NIST concerning AES modes of operation: CTR-mode encryption. 2000
[24] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. "The Keccak sponge function family: Specifications summary".

of the state, alternated with state transformations. The size of the part of the state that is written and read is called the "rate" (often denoted r), and the size of the part that is untouched by input/output is called the "capacity" (often denoted c). The capacity determines the security of the scheme. The maximum-security level is half the capacity.
Finally, a keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. It may be used to simultaneously verify both the data integrity and the authentication of a message, as with any MAC. Any cryptographic hash function, such as MD5 or SHA-3, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output, and on the size and quality of the key. An iterative hash function breaks up a message into blocks of a fixed size and iterates over them with a compression function. For example, MD5 and SHA-1 operate on 512-bit blocks. The size of the output of HMAC is the same as that of the underlying hash function (128 or 160 bits in the case of MD5 or SHA-1, respectively), although it can be truncated if desired.
Within **SE*file*™** each sector, including the header, is signed using an authenticated signature obtained with **SHA-256-HMAC**, meaning that the signature depends on both the data contained in the sector itself and on a chosen encryption key. To use two different keys to encrypt data and to digest authentication, a feature increasing overall system security, **SE*file*™** leverages on the `pbkdf2()` function already implemented within the SDK. This function, provided with a 32 byte long salt vector (randomly chosen), is used to generate parameters needed for the secure sessions, such as a new key and the number of iterations of the authentication procedure. This mechanism is important to enhance security, because even if one key is unveiled, the second one would be too difficult to obtain.
The procedure enforced within **SE*file*™** to ensure data protection and confidentiality is hereby described. Firstly, the file is divided into sectors containing each exactly 512 bytes (constant defined as `SEFILE_SECTOR_SIZE` that can be modified as long as it is bigger than 256 and it is a power of 2).
Except for the first (the header), each sector is divided into three main fields: data, length and signature. The length field is composed of 2 bytes and stores the number of valid user data bytes contained in the data field. The signature field is composed of 32 bytes and stores the result of the authenticated signature, to check if the sector is corrupted or not and if the data stored in the sector were written by an authorized user.
The data field represents the effective payload where the user data are stored, it can be computed as `SEFILE_LOGIC_DATA = SEFILE_SECTOR_SIZE − 2 − 32` (bytes).
The first sector of the Secure File follows a different structure from the others and is not used to store user data; instead it contains necessary information about the file itself, organized as follows:

```
typedef struct {
  uint8_t nonce_pbkdf2[32];
  uint32_t key_id;
  uint16_t algorithm;
  uint8_t padding[10];
  uint8_t nonce_crt[16];
  uint8_t magic;
  uint8_t ver;
  uint8_t uid;
  uint8_t uid_cnt;
  uint8_t fname_len;
} SEFILE_HEADER;
```

Within this structure, `nonce_pbkdf2` is a 32 byte long *salt* used for generating a different key to authenticate digest. `key_id` and `algorithm` are the IDs of the key and of the algorithm used to encrypt the file. `padding` is just a 10 byte array used to pad the size of the data to be encrypted to a multiple of the cipher's block size. `Nonce_crt` is the random initialization vector used as counter for encrypting all the data sectors of the secure file. The `fname_len` field contains the length of the filename which is written right after the header fields.
The `magic` field might be used for representing what type of file has been encrypted. The ver field is used for representing with what version of **SE*file*™** it has been encrypted. The `uid` and `uid_cnt` fields, finally, are designed to host information about the user who encrypted the file and its permission. However, all these features are not supported yet.
All the unused bytes for padding of the header sector, and all the unused bytes obtained when any sector is not filled up to its capacity, are randomly chosen to avoid a *known plain-text attack*, an attack model for cryptanalysis where the attacker has access to both the plaintext (called a crib), and its encrypted version (ciphertext). These can be used to reveal further secret information such as secret keys and code books.

6.5 The SE*file*™ Class

SE*file*™ was originally written in C, however, it has been ported to C++ in order to take advantage of many useful features. In particular, **SE*file*™** is now object oriented and it exploits paradigms like the RAII to grant usability along with correct memory management. The SEfile class allows to create **SE*file*™** objects, each object is used to manage a specific file with **SE*file*™** . A **SE*file*™** object is characterized by several attributes, the most important are:

- the `l1` attribute is a pointer to the L1 object that is used to communicate with the **SE*cube*™** connected to the host machine;
- the `EnvKeyID` attribute is a 4 byte unsigned integer that identifies the key to be used to encrypt or decrypt the data of the file;
- the `EnvCrypto` attribute is a 2 byte unsigned integer that identifies the algorithm to be used to encrypt or decrypt the data of the file.

These attributes can be set using the constructors available for the **SE*file*™** object but they can be set later as well using specific APIs. Notice that the constructors of the **SE*file*™** object do not automatically open or create any file, that must be done manually using dedicated APIs. On the other hand, the destructor of the **SE*file*™** object automatically closes the file managed by the object, if the file is still open, and it also releases all the resources required by the object itself.
Each **SE*file*™** object exposes several APIs that can be used to open, close, read, and write the underlying file. These APIs emulate the functionalities available in the file system interfaces of most OSs, however, their content is specialized to take into account the special structure of each file.

6.6 SE*file*™ Implementation for Encrytped SQL Databases

SE*file*™ is a L2 library, so it is the starting point to develop new libraries belonging to the L3 abstraction layer. While developing **SE*key*™** , there was the need to store the metadata of the Key Management System in a secure way, outside of the memory of the **SE*cube*™** device (because its size is too small). Starting from this requirement, the L3 library for the Secure Database was implemented.
The Secure Database is based on several elements: the **SE*cube*™** , a customized version of the **SE*file*™** library, the SQLite database engine and a custom virtual file system interface for SQLite.

A secure database, in this case, is a normal SQL database that can be managed with the standard SQLite C interface, so nothing changes for the end users. The real change is hidden inside SQLite, because the custom virtual file system interface integrate in the database engine has been implemented in order to use the APIs of **SE*file*™** . The result is that, in a completely transparent way, applications resorting to this library can use SQL databases exactly as they always did but, this time, the database files are constantly encrypted on disk. It is clear that this approach implies an overhead in terms of performance; however, our goal is to ensure security above everything else. Notice that, because of some problems found at the interface between the standard **SE*file*™** library and SQLite, a customized version of the **SE*file*™** library has been developed for the Secure Database. This customized version is equal to the standard one, the only difference is the internal structure of the encrypted sectores of the database file. The source code of the Secure Database library is included with **SE*file*™** library; if you simply want to try **SE*file*™** and you do not need to manage encrypted SQL databases, then you can simply ignore the APIs related to the secure database. On the other hand, if you want to use **SE*file*™** to store some data into encrypted SQL databases and the SQLite database engine is sufficient for your goal, then you know that there are specialized **SE*file*™** APIs for that purpose so that you do not need to worry about low level details. The Secure Database library is also needed, as well as **SE*file*™** , for the **SE*key*™** library.

6.6.1 The Interface Between SQLite and SE*file*™

The interoperability between SQLite and **SE*file*™** is granted by a software layer called Custom Virtual File System (CVFS). This layer allows the developers to implement a customized file system interface upon the platform where the SQLite engine is supposed to run, in the case of **SE*file*™** the CVFS was developed to work with Windows and Linux. The purpose of the CVFS is to force SQLite to use the abstractions provided by **SE*file*™** instead of the traditional file system interface offered by the OS; in this way we can automatically obtain a SQL database with all the security properties of **SE*file*™** .
The starting point of this work has been the official template offered as example for implementing a CVFS interface distributed with SQLite. This template, by default, is compatible only with Unix-based OSs but it was tweaked to work also with Windows. Moreover, the CVFS implements a simple software cache for reducing the number of disk accesses, increasing the overall performance. Precompiler definitions are set so that the SQLite engine is forced to use this CVFS.
Hereby it is listed a subset of the most important CVFS interface functions that have been implemented so that SQLite uses **SE*file*™** to manage files and databases. Notice that you never need to call these functions, in fact they are only used by the SQLite engine. The only reason you may want to take a look at the details of these functions is if you want to improve the custom virtual file system.

```
SQLITE_API int SQLITE_STDCALL sqlite3_os_init(void)
SQLITE_API int SQLITE_STDCALL sqlite3_os_end(void)
```

These two functions are respectively used to assign and release the data structure made up by pointers to the rest of VFS interfaces that has been associated with common I/O operations.

```
static int SecureDirectWrite(SecureFile *p, const void *zBuf,
   int iAmt, sqlite_int64 iOfst)
```

This function is used to wrap a write operation, it accepts as parameters a custom file descriptor, the buffer that should be written, the number of bytes to be written and the offset from the start of the file. This function checks if the software cache should be flushed to disk, change the file pointer to `iOfst` and the issue a `secure_write()` call.

```
static int SecureRead(sqlite3_file *pFile, void *zBuf, int iAmt,
    sqlite_int64 iOfst)
```

This function is used to wrap a read operation, it accepts as parameters a custom file descriptor, the buffer that should be read, the number of bytes to be read and the offset from the start of the file. This function checks if the software cache should be flushed to disk, change the file pointer to `iOfst` and the issue a `secure_read()` call.

```
static int SecureTruncate(sqlite3_file *pFile, sqlite_int64 size
   )
```

This function is used to change the size of the pointed file `pFile` to `size`. In this case it simply issues a `secure_truncate()`.

```
static int SecureSync(sqlite3_file *pFile, int flags)
```

This function is used to flush OS buffers (and not the software cache) to disk thanks to `secure_fsync()`. In this case the flags are ignored.

```
static int SecureFileSize(sqlite3_file *pFile, sqlite_int64 *
   pSize)
```

This function firstly writes the pending cache to disk, then it returns the current size of the file thanks to `secure_getfilesize()`. Since `sqlite3_file` is highly customizable, the path was added to the file descriptor to be compatible to the **SE*file*™** API.

```
static int SecureOpen(sqlite3_vfs *pVfs, const char *zName,
   sqlite3_file *pFile, int flags, int *pOutFlags)
```

This function is used to manage opening/creating of a secure database thanks to `secure_open()`. In this case `pVfs` and `pOutFlags` were ignored, while `zName` is the path to the file that should be opened, `pFile` is the pointer to the file descriptor obtained, and `flags` is used to determine how the file should be opened.
Pay attention to the following flags combinations:

- The combination (`SEFILE_READ`, `SEFILE_NEWFILE`) is not allowed and therefore fails;
- The combinations (`SEFILE_READ`, `SEFILE_OPEN`) and (`SEFILE_WRITE`, `SEFILE_OPEN`) work only if the file already exists.

```
static int SecureDelete(sqlite3_vfs *pVfs, const char *zPath,
   int dirSync)
```

This function is used to delete a file pointed by `zPath` and it was developed as a wrapper to `unlink()` in Unix and `DeleteFile()` in Windows, thanks to `crypto_filename()` function. In this case `dirSync` parameter is ignored.

```
static int SecureFullPathname(sqlite3_vfs *pVfs, const char *
   zPath, int nPathOut, char *zPathOut)
```

This function is used to retrieve the full path of a secure database pointed by `zPath` by writing at most `nPathOut` bytes to `zPathOut`. In this case `pVfs` is ignored.

6.7 SE*file*™ APIs

Here we provide a simplified and high-level overview about the **SE*file*™** APIs, notice that there are other functions inside the **SE*file*™** library that are used for internal purposes. The APIs listed here are everything you need to profitably use the library; however, please refer to the Doxygen documentation to find out more details about the APIs and the other functions of **SE*file*™** .
Inside the source code of the **SE*file*™** library, you will also find APIs developed to be used exclusively with the SQLite database engine. The name of these functions always begins with 'securedb'. Some of these functions belong to the `SEfile` class, they should not be used explicitly because they are automatically called by the traditional SQLite C interface (i.e., `sqlite3_open()` internally calls `securedb_secure_open()`). The only **SE*file*™** functions reserved to SQLite that you may want to use directly are `securedb_ls()` and `securedb_recrypt()`.

```
uint16_t secure_init(L1 l1, uint32_t keyID, uint16_t crypto)
uint16_t secure_finit()
```

These functions are used to setup the basic attributes of each SEfile object in association with a file encrytpted with **SE*file*™** . In particular:

- the `l1` parameter is a pointer to the L1 object that is used to communicate with the **SE*cube*™** connected to the host machine;
- the `keyID` parameter is a 4 byte unsigned integer that identifies the key to be used to encrypt or decrypt the data of the file;
- the `crypto` parameter is a 2 byte unsigned integer that identifies the algorithm to be used to encrypt or decrypt the data of the file.

These three attributes are specific to each SEfile object, obviously a dedicated SEfile object is required for each file that needs to be managed by **SE*file*™** . The `secure_init()` is used to initialize those attributes, the `secure_finit()` is used to reset those attributes to default values (i.e. `NULL` for the **SE*cube*™** pointer, 0 for the key and the algorithm).
Notice that the only attribute that you always need to setup is the `l1` pointer because **SE*file*™** must communicate with a **SE*cube*™** . On the other hand, the `keyID` and `crypto` attributes need to be set only when you want to create a file. If the file already exists and you simply want to access to it, then **SE*file*™** will automatically adjust the key and the algorithm according to the key ID and algorithm ID specified in the header sector of the file itself (provided that you have the right key stored on your **SE*cube*™**).
The usage of these APIs is not mandatory, you can simply use the constructors available for the SEfile object. Notice that the destructor of the SEfile object automatically calls the `secure_finit()` so that the user does not have to worry about resources deallocation.

```
uint16_t secure_open(char *path, int32_t mode, int32_t creation)
```

This function, given the name of a file as plaintext (relative path or absolute path), is used to open an existing file or create a new one. The name of the file is modified with the `crypto_filename()` function, which transforms it into its digest (64 hex chars) computed with SHA-256. Notice that there is also another function, called `secure_create()`, that is used automatically by the `secure_open()` to create a new file; the `secure_create()` should never be called directly because it is intended to be used exclusively by the `secure_open()`.
The `mode` parameter is used to specify read-only or read-write privilege, the `creation` parameter is used to specify the opening policy (i.e., `SEFILE_NEWFILE` forces the creation of the file, `SEFILE_EXISTING` opens an existing file). A real write-only mode has not been implemented

since a dedicated `secure_write()` function exists. Notice that you must specify in advance if you want to create the file or if you want to open it; there is not a mode to open it if exists or create it if it does not exists (you can implement it by yourself generating the encrypted file name with `crypto_filename()` and checking if that file exists or not).
If a new file is created, the header sector is filled with appropriate information (i.e. the ID of the encryption key of the SEfile object upon which the method is called, the ID of the algorithm, the name of the file, etc.), then the header sector is encrypted and signed (except for the `key_id`, `algorithm`, and `nonce_pbkdf2`, as it is needed to check the signature of the header sector itself) before writing it to disk.
If an existing file is opened, the clear text part of the header sector is read to set the correct key ID and algorithm in the SEfile object, then the rest of the header is decrypted and the signature is checked; if everything is correct and the key can be used for decryption, the file can be used.
Independently from the actual behaviour of the `secure_open()` function, if it succeeds the file pointer is set to the first byte of the first sector placed after the header.
The following algorithm demonstrates how the `secure_open()` works.

Algorithm 1 How a secure file is opened or created

```
function SECURE_OPEN(in path, in mode, in creation)
    if creation == SEFILE_NEWFILE then
        return SECURE_CREATE()
    end if
    // existing file (SEFILE_OPEN) from now on
    generate encrypted filename with crypto_filename()
    OS system call to open the encrypted file according to mode
    set the key ID and algorithm ID according to the header content
    check if the inherited key ID can be used for decryption
    decrypt header and check signature
    return
end function
```

Algorithm 2 How a secure file is created

```
function SECURE_CREATE(in path, in hFile, in mode)
    check if specified key can be used for encryption
    generate encrypted filename with crypto_filename()
    OS system call to create the encrypted file according to mode
    populate the header of the file
    encrypt and sign the header sector
    write the header sector to disk
    move the file pointer to the first byte after the header sector
    return
end function
```

```
uint16_t secure_close()
```

This function simply closes the file associated to the SEfile object and deallocates all the resources that were acquired. This function is called automatically by the destructor of the SEfile object, therefore you do not need to call it manually all the time (but you are suggested to, because it is good practice).

```
uint16_t secure_read(uint8_t *dataOut, uint32_t dataOut_len,
```

```
    uint32_t *bytesRead)
```

The `secure_read()` function works as the `read()` in Unix and the `ReadFile()` in Windows, adding all the needed operations related to the secure file management.
The number of bytes requested as clear text is provided in `uint32_t dataOut_len` while the actual number of read bytes is stored in `bytesRead`. In details, the operations performed are: starting from the position pointed by the file pointer the function extracts sequentially all the sectors related to the requested portion of data to be read, check for its integrity by looking at the signature, decrypts the sector and concatenates the read data in the output buffer `dataOut`. After that, the file pointer points after the last byte read. A read operation issued requesting a number of bytes that is not aligned to the sector size and is not a multiple of `SEFILE_LOGIC_DATA` will lead to performance degradation, since it still needs to decrypt the whole sector.
The implemented functionality is shown in the following algorithm.

Algorithm 3 How a secure file is read

function *SECURE_READ(out dataOut, in dataOut_len, out bytesRead)*
  check if specified key can be used for decryption
  set number of read bytes to zero
  do
    read, decrypt and verify signature of current sector
    append decrypted data do dataOut
    bytesRead = bytesRead + data read
    dataOut_len = dataOut_len - data read
    go on with next sector if required
  while *dataOut_len > 0*
end function

```
uint16_t secure_write(uint8_t *dataIn, uint32_t dataIn_len)
```

The `secure_write()` function masks the `write()` in Unix and the `WriteFile()` in Windows, adding all the needed operations related to the secure file management. The function writes in the file the data sent as clear text in the buffer. In particular, the function divides the buffer into sectors, then it encrypts and signs each sector and writes it in the specified position in the file. After this operation, the file pointer points after the last byte written.
In this case, it has been chosen to not return the actual number of written bytes since if the operation fails in writing `dataIn_len` bytes it would result as an error.
If a `secure_write()` is issued requesting to write a number of bytes that is not aligned to the sector size and is not a multiple of `SEFILE_LOGIC_DATA`, since it still needs to decrypt the whole sector, it will lead to performance degradation.
The implemented functionality is shown in the following algorithm.

Algorithm 4 How a secure file is written

function *SECURE_WRITE(in dataIn, in dataIn_len)*
  check if specified key can be used for encryption
  if *file pointer not aligned to sector size* ***then***
    read, decrypt and verify signature of current sector
    store inside the buffer the sector to be written
  end if
  do
    append data from dataIn to output buffer
    encrypt and sign the sector to be written
    write the sector to disk

decrement the amount of data still to be written
while *dataIn_len > 0*
end function

```
uint16_t secure_seek(int32_t offset, int32_t *position, uint8_t
    whence)
```

This function moves the file pointer by `offset` bytes, taking care of the effective byte of user data and skipping the bytes related to the overhead introduced by **SE*file*™** itself (i.e. header sector, signature field and data length). The parameter `whence` is used to choose if the user wants move the file pointer from the beginning of the file, from the current position, or from the end of the file. The `position` parameter is used to store the logic value where the file pointer is set after issuing `secure_seek()`.
If the destination exceeds the file size, the file is resized by adding zeros until the specified position. This function has proper mechanisms to avoid jumping inside the header sector. The implemented functionality is shown in the following algorithm.

Algorithm 5 How a secure file pointer is moved

function *SECURE_SEEK(in offset, out position, in whence)*
retrieve the size of the file using get_filesize()
if *offset > file size* ***then***
move the file pointer to the last sector using OS system call
add as many bytes equal to zero as necessary to reach a file size equal to the offset
return *position = current file pointer position*
end if
computer file pointer destination according to whence
move the file pointer to destination using OS system call
return *position = destination*
end function

```
uint16_t secure_truncate(uint32_t size)
```

This function resizes the file to `size` bytes. It takes care of the sectors and leaves the file pointer to the end of the file (after the last byte of user data).
If the specified file is bigger than the original, sectors are filled with zeros, otherwise data in excess are lost. The implemented functionality is shown in the following algorithm.

Algorithm 6 How a secure file is truncated

function *SECURE_TRUNCATE(in newsize)*
retrieve the size of the file using get_filesize()
if *newsize > file size* ***then***
return *SECURE_SEEK(newsize - file size, nullptr, end of file)*
end if
compute which sector will become the last one of the file
move file pointer to the computed sector
read, decrypt and verify the last sector
keep only the data of that last sector that must be preserved by the truncation
truncate the file using the OS system call
write back the previously saved data with the secure_write()
end function

```
uint16_t secure_sync()
```

This function is used in case it is needed to be sure that the OS buffers are correctly flushed to the physical file.

```
uint16_t get_secure_context(std::string& filename, std::string *
    keyid, uint16_t *algo)
```

This function, given the clear text name of a file, returns the ID of the key and the ID of the algorithm used by **SE*file*™** to encrypt and authenticate that file. This is useful in many situations, for example when working with other functions like `secure_ls()`.

```
uint16_t secure_recrypt(std::string path, uint32_t key, L1 *
    SEcubeptr)
```

This function is used to decrypt and encrypt again, with a new key, a file managed by **SE*file*™** that was encrypted with a key that is considered not secure anymore. This function ideally should be used together with the **SE*key*™** KMS; however, it can be easily used also without having the KMS running (as long as you resort to keys which are not in the range of IDs managed by the KMS).
The function takes as parameters the clear text name of the file, the key to be used for the new encryption and the pointer to the L1 object used to communicate with the **SE*cube*™** .
If the function succeeds, the old file will be replaced with a new file whose content is identical and encrypted with the new key; the old file will be deleted. If the function fails, no changes are applied.
Notice that the same function is available also for encrypted SQLite databases under the name of `securedb_recrypt()`.

```
uint16_t crypto_filename(char *path, char *enc_name, uint16_t *
    encoded_length)
```

This function computes the encrypted name of the file specified at position `path` and writes the result to `enc_name`; the quantity of bytes written is saved in `encoded_length`. The filename is computed using the SHA-256 algorithm, so there is no decryption function to obtain its clear text name unless the header sector is decrypted. Since the service which computes the SHA-256 works with 32 Bytes block, its result is always on 32 bytes, and it is represented as hexadecimal values in ASCII encoding, meaning that for each byte there will be 2 character, resulting in a 64 characters length.
In any case, this function takes care of parsing `path` so in `enc_name` will be copied everything that comes before a "/" or "\" character to compute just the hash of the filename to encrypt.

```
secure_getfilesize(char *path, uint32_t * position, L1 *
    SEcubeptr)
```

This function is used to retrieve the total logic size (how many bytes of valid data, excluding the **SE*file*™** overhead) of an encrypted file pointed by `path`, the result is stored in `position`. The `SEcubeptr` parameter is a pointer to the L1 object used to communicate with the **SE*cube*™** connected to the host machine. This function does not need to be called upon a SEfile object but can be normally used simply passing the clear text name of the file. Notice that the logic size of the file will always be smaller than the physical size given the overhead introduced by **SE*file*™** .
The implemented functionality is shown in the following algorithm.

Algorithm 7 How a secure file size is computed

function *SECURE_GETFILESIZE(in path, out position, in SEcubeptr)*
open the file pointed by path using secure_open()
move the file pointer to the last sector of the file using OS system call
if *number of sectors of the file = 1* ***then***
return *position = 0*
end if
read, decrypt, verify the last sector of the file
*position = ((total file size / sector size) - 1) * valid bytes in each sector + valid bytes in last sector*
close the file pointed by path using secure_close()
return *position*
end function

```
secure_ls(string& path, vector<pair<string, string>>& list, L1 *
   SEcubeptr)
```

This function is used to list the content of a directory containing encrypted files and/or directories. The `path` parameter tells to the function where to search, the `list` parameter stores as first element of each pair the name of the file or directory as it appears to the user (i.e. the encrypted file name of a file managed by **SE*file*™**) and as second element the actual name of the file or directory. The third parameter is the pointer to the L1 object used to communicate with the **SE*cube*™** .
Notice that this function works with any file or directory. In particular, if the name to list is not recognized as a name belonging to the 'nomenclature' of **SE*file*™** , it is simply copied as it is. If this function finds encrypted files managed by **SE*file*™** APIs specific for the SQLite database engine, then their names will not be decrypted; to list their real names use instead the `securedb_ls()` function. The implemented functionality is shown in the following algorithm.

Algorithm 8 How to discover the names of encrypted files in a folder

function *SECURE_LS(in path, out list, in SEcubeptr)*
retrieve list of files and directories within specified directory using OS system call
do
if *current element in list is a directory* ***then***
decrypt directory name
if *decryption successful* ***then***
add decrypted name to list
else
add original name to list
end if
else if *current element in list is a file* ***then***
open the file and try to decrypt the header
if *decryption successful* ***then***
add clear text name to list
else
add original name to list
end if
end if
while *all files and directoris within specified directory have been processed*
end function

```
uint16_t secure_mkdir(string& path, L1 *SEcubeptr, uint32_t key)
```

This function masks the `mkdir()` function of the Unix environment and the `CreateDirectory()` function of Windows, but it does not implement the whole functionalities of those functions. Since directories are created using a wrapper to the OS system call, it is not possible to achieve a mechanism like the one employed for regular files, so it has been decided to use this encryption scheme, leveraging to `crypt_dirname()`, just for the name of the directory: the first 8 characters are the hexadecimal representation in ASCII of the key ID, the rest is obtained computing the AES-256-ECB of the name specified as clear text. The `SEcubeptr` parameter is, as usual, the pointer to the L1 object used to communicate with the **SE*cube*™** ; the `key` parameter is the ID of the key to be used to encrypt the name of the directory.

7 The SE*link*™ Library

7.1 Introduction

SE*link*™ is an API for the **SE*cube*™** platform whose aim is to secure any data belonging to the 'data in motion' domain. **SE*link*™** can be used to establish an encrypted communication channel between different machines (i.e., two computers), each one having access to a dedicated **SE*cube*™** device. This channel works like a VPN, each message that is exchanged through the encrypted channel is guaranteed to be secure because **SE*link*™** grants confidentiality, integrity and authentication of data.
Despite being completely transparent to the end users, **SE*link*™** requires an effort from the developers in order to be integrated into existing software. What the developers need to do is, basically, integrating the **SE*link*™** API into any software that needs a certain level of security for data in motion. With the API of **SE*link*™** , data can be encrypted and serialized before being sent to the recipients who, in their turn, can use the same API to deserialize and decrypt the data. This is a very simple way to implement a secure end-to-end communication channel.
The most important feature of **SE*link*™** , besides security, is the ease of use; there are only few functions inside the API, therefore the learning curve is very easy to overcome. Developers are immediately enabled to proficiently use the library in order to secure data and applications without the need of caring about low level security details. **SE*link*™** already takes everything into account: encryption, decryption, serialization, deserialization, data integrity, and data authentication.

7.2 SE*link*™ High-level Overview

The purpose of **SE*link*™** is to easily and efficiently secure data traffic in the 'data in motion' scenario by means of an encrypted communication channel. **SE*link*™** is completely independent from the underlying protocol that is used to actually transport the data because **SE*link*™** is embedded directly into the applications, so it is able to encrypt the data before they are processed by the lower levels (i.e. the software stack that implements the communication protocols, such as TCP). This very approach is reversed on the recipient side; the underlying protocols manage the received data without knowing what is actually going on because the data are encrypted. The encrypted communication then is passed to the higher levels up to the application that, embedding the **SE*link*™** API, is able to decrypt the data before using them for computation or presenting them to the user.
Notice that the **SE*link*™** library does not automatically intercept any kind of traffic, regardless of the protocol. Developers who want to develop applications based on secure communication channels are required to integrate the **SE*link*™** API directly into the source code of the applications.
Let us imagine that we want to use an open-source software to exchange simple text messages over the Internet with other people. Let us suppose also that everybody has access to a personal **SE*cube*™** device. We can customize the open source software so that it exploits **SE*link*™** to encrypt and decrypt the messages we exchange. Moreover, we can also use **SE*key*™** to increase the overall security and automation of the process by enabling key management. Therefore, the scenario is this: each person is registered to the **SE*key*™** KMS, each person has a dedicated **SE*cube*™** device and each person uses a customized application on his computer to send and receive encrypted text messages.
Suppose that Alice and Bob want to exchange messages with end-to-end encryption using their **SE*cube*™** devices and a software specifically developed for that purpose. This is how **SE*link*™** comes into play:

1. Alice writes a text message using the custom application, then she clicks the button to send the message to Bob.

2. The custom application, before actually sending the message over the Internet, calls the **SE*link*™** API to encrypt the message. This API exploits also **SE*key*™** to find the most secure key to be used, depending on who is the recipient (in this case Bob).
3. Once the payload is encrypted, the cipher text is serialized by the custom application using a specific **SE*link*™** method.
4. The custom application sends the serialized cipher text over the Internet to Bob.
5. Bob's custom application deserializes the data, then it calls the **SE*link*™** API to decrypt them.
6. The **SE*link*™** API decrypts the data and checks the signature, in order to grant integrity and authentication.
7. If everything goes fine, Bob can see the message that Alice sent.

This behavior is very simple, it basically implements an end-to-end cryptographic mechanism that can be used to protect any communication.
Notice that **SE*link*™** automatically takes care of low level cryptographic details such as padding computation, signature computation, signature verification, and so on.

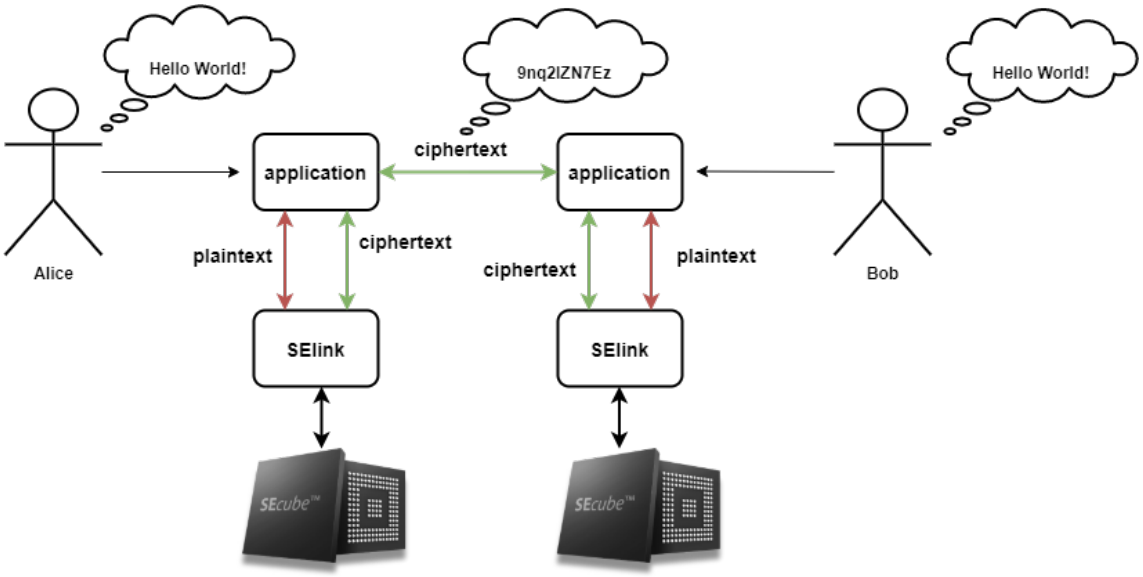


Figure 21: **SE*link*™** high-level overview.

7.3 The SE*link*™ class

The **SE*link*™** API heavily resorts on the features provided by the L1 layer. In order to manage the data encrypted by **SE*link*™** , the API is based on a 'SElink' object that encapsulates a `SEcube_ciphertext` object, the same that is used in L1 API. On top of this, **SE*link*™** introduces also a couple of methods to serialize and deserialize the data, simply because **SE*link*™** is focused on 'data in motion'. The serialization is done using the base64 encoding.

```
class SElink{
public:
  SEcube_ciphertext ciphertext;
  void serialize(unique_ptr<char[]>& serialized_data, size_t&
     serialized_size);
```

```
  void deserialize(unique_ptr<char[]>& buffer_b64, size_t
     buffer_b64_size);
};
```

In conclusion, a 'SElink'' object is, by all means, equal to the SEcube_ciphertext object already used by L1 APIs. The only different is that this new object also embeds the ability to be serialized and deserielized. Moreover, the encryption/decryption functions of **SE*link*™** are based on the 'SElink' object

7.4 SE*link*™ APIs

This section provides a brief overview about the **SE*link*™** APIs. For more details about their implementation, please refer to the extensive comments in the source code. You can also process the code through the Doxygen engine in order to generate the documentation.

```
int selink_encrypt_manual(L1 *l1, shared_ptr<uint8_t[]>
   plaintext, size_t plaintext_size, SElink& ciphertext,
   uint32_t key);
```

This function is used to encrypt a plain text with **SE*link*™** . The 'manual' in the name of the function means that the caller is supposed to provide a key ID (an unsigned integer on 4 bytes) to the function, if the **SE*cube*™** contains a key with the provided ID, then the plain text is encrypted. If the function succeeds, the 'SElink' object passed as fourth is filled with the cipher text and other attributes. Then, this object can be serialized in order to be sent to the receiver (serialization might not be always required). Finally, notice that the caller is supposed to provide the plain text as an array of bytes, passing also the size of the array.

```
int selink_encrypt_auto(L1 *l1, shared_ptr<uint8_t[]> plaintext,
    size_t plaintext_size, SElink& ciphertext, vector<std::
   string>& recipient);
```

This function is, by all means, equal to the previous one. The only difference is that the user is not supposed to provide the ID of the key to be used for encryption. The user, in fact, should only provide the list of recipients. This list might include one or multiple users from **SE*key*™** , or even an entire group from **SE*key*™** . The list must be filled with the IDs of the recipients. When issued, this function will automatically resort to **SE*key*™** to find the most suitable key for encryption, then it will fill the SElink object passed as fourth parameter with the encrypted data. As usual, serialization can be performed after encryption.

```
int selink_decrypt(L1 *l1, shared_ptr<uint8_t[]>& plaintext,
   size_t& plaintext_size, SElink& ciphertext);
```

This function is used to decrypt a payload that was encrypted by one of the two funcitons described above. Notice that the caller must provide the cipher text object, **SE*link*™** will decrypt the cipher text and it will automatically verify the signature, providing therefore integrity and authentication of the data. In case of success, the plain text will be stored in the byte array passed as second parameter. In case of error, for example because the signatures of the data do not match, an error is returned.

```
void serialize(unique_ptr<char[]>& serialized_data, size_t&
   serialized_size);
```

Serialize the encrypted data stored by a SElink object using the base64 encoding. This method belongs to the SElink class. The serialized buffer is stored in the first parameter, the size of the serialized buffer is stored in the second parameter.

```
void deserialize(std::unique_ptr<char[]>& buffer_b64, size_t
   buffer_b64_size);
```

Deserialize a data buffer that was previously serialized with the corresponding method of the `SElink` class. The first parameter is the buffer containing the serialized data, the second parameter is the size of the serialized data. After completion, this method stores the deserialized data inside the `SElink` object upon which the deserialize function is called. After deserialization, the 'SElink' object can be passed as input to the `selink_decrypt()` function in order to decrypt the data. This method belongs to the `SElink` class.

8 The SE*key*™ library

SE*key*™ is a Key Management System (KMS) specific for the **SE*cube*™** *Open Security Platform*. The target of **SE*key*™** is to allow an easy and secure management of *cryptographic keys* to be used by applications based on the **SE*cube*™** device. **SE*key*™** supports the secure generation, distribution, and usage of keys in domains characterized by multiple users who need to share data that require to be protected by means of cryptographic algorithms running on the **SE*cube*™** device.

8.1 Key Management System

A Key Management System (KMS) is, unsurprisingly, a system that aims at managing cryptographic keys in the most secure and efficient way. All Key Management Systems provide common functionalities:

- cryptographic keys life cycle management: creation, storage, usage, deletion;
- protocols for the distribution of cryptographic keys;
- authentication and authorization of actors in order to limit the access to the cryptographic keys;
- accountability.

Besides these basic requirements that are, in general, met by any KMS, each system implements other features that tackle security issues with a highly specialized and customized approach, for example the tecniques that are used to protect the cryptographic keys from a physical and logical point of view.
According to the National Institute of Standards and Technology (NIST)[25]: "*The proper management of cryptographic keys is essential to the effective use of cryptography for security. Keys are analogous to the combination of a safe. If a safe combination is known to an adversary, the strongest safe provides no security against penetration. Similarly, poor key management may easily compromise strong algorithms.*"[26].
Cryptographic keys must be protected, otherwise they become useless because they are only as good as the security used to protect them. A secure Key Management System relies on policies, procedures, and components (both hardware and software) carefully selected and implemented.

8.2 Cryptographic Keys

Cryptographic keys used within the **SE*cube*™** *Open Security Platform* are always focused on *symmetric* encryption algorithms, such as AES-256. Symmetric encryption algorithms use the same

[25] https://www.nist.gov
[26] https://nvlpubs.nist.gov/nistpub/SpecialPublications/NIST.SP.800-57pt1r4.pdf

key for encryption and decryption, they are fast and resilient against quantum cryptopgraphy attacks but they require both parties to know the same key.
SE*key*™ is designed to manage any type of key, independently from its type (i.e. symmetric or asymmetric). However, since the **SE*cube*™** supports only AES and AES-HMAC-SHA-256 (both with key lengths of 128, 192, 256 bits), the APIs of the KMS have been implemented and tested taking into account the eclusive usage of the AES and AES-HMAC-SHA-256 algorithms.
Each key is characterized by several attributes:

- **ID**: a unique value that is used to identify the key within the KMS, the format of the ID is '*K123*' where '*123*' can be replaced by any number between 0 and $2^{32} - 1$;
- **label**: a human-readable text string describing the key (i.e. *Key June 2020 Customer Care Department*);
- **owner**: the ID of the owner of the key, meaning the entity to which the key is related (i.e., a key may be reserved to be used by a specific group of users, so the group is the owner of the key);
- **status**: the current state of the key (i.e., active, suspended, deactivated, compromised, destroyed, etc.);
- **type**: the type of the key (i.e., symmetric data encryption);
- **algorithm**: which cryptographic algorithm is coupled to the key (i.e. AES-256);
- **length**: number of bit composing the key;
- **generation**: date and time at which a key is generated;
- **activation**: date and time at which a key is activated;
- **expiration**: date and time at which a key is expired;
- **deactivation**: date and time at which a key is deactivated;
- **suspension**: date and time at which a key is suspended;
- **compromise**: date and time at which a key is compromised;
- **destruction**: date and time at which a key is destroyed;
- **cryptoperiod**: time span during which a key can be used both for encryption and decryption, it is a time interval that is added to the activation time to compute the expiration time of the key.

During its lifecycle, a key can go through many states, from its generation to its destruction. The state of a key determines the spectrum of usage of the key, for example a key that is not active cannot be used to encrypt data. Figure 22 shows the key state diagram and the possible state transitions. Notice that these transitions are not mandatory by any means, in fact the only state that is always assumed by a key is the **Pre-Active** because that is set by default during the key generation function. Once a key is created, all other transitions are optional (i.e. a key may stay forever in the default state while another one may go through all states).
Notice that a key can always be set to the compromised or destroyed state, this is required for security reasons.

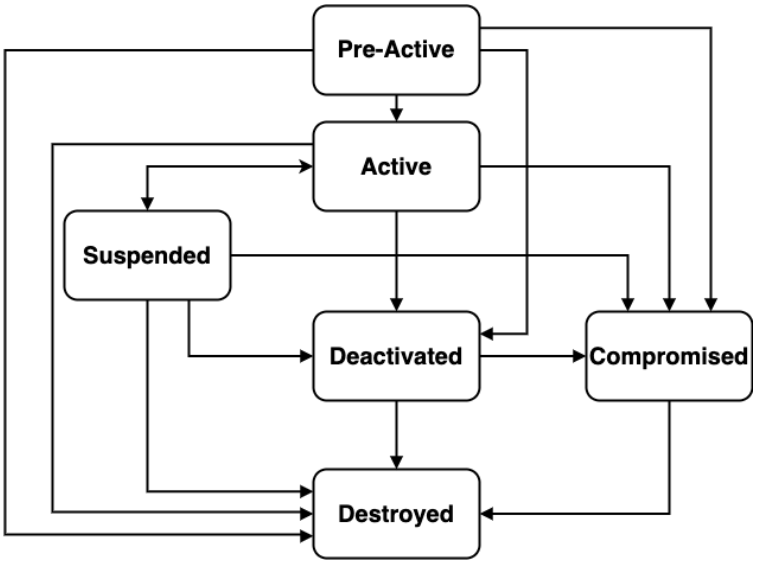


Figure 22: Key State diagram.

- **Pre-Active**: a key in this state cannot be used neither for encryption nor for decryption. This is the default state for every new key.
- **Active**: a key in this state can be used for encryption and decryption. This implies that it was activated in the past and that it will be deactivated in future.
- **Suspended**: a key can be in this state only if it was active and then it was suspended for some reason. A suspended key can be used for decryption but not for encryption, it can also be activated again. Notice that the time during which the key is suspended still contributes to the time that determines the expiration of the key.
- **Deactivated**: a key in this state cannot be used for encryption but only for decryption of old files once encrypted with the key when it was active. Notice that a key may be set to the deactivated state even without being active in the past, on the other hand a key which is active will be set to deactivated once it reaches its expiration time.
- **Compromised**: a key in this state may have been stolen by some attacker or may have been leaked outside of **SE*key*™** . This key is not secure anymore, therefore it must not be used for encryption; it should be used as soon as possible to decrypt the files which were encrypted, in order to encrypt them again with a new, secure key. Old files related to the compromised key should be deleted.
- **Destroyed**: a key in this state is not part of **SE*key*™** anymore. Its metadata are still retained by **SE*key*™** ; however, the actual key value has been deleted and cannot be recovered so all files which are still encrypted with a destroyed key are lost forever.

8.3 The Architecture of SE*key*™

From a physical point of view, **SE*key*™** has been developed to be distributed among multiple **SE*cube*™** devices and multiple actors: one actor with a dedicated **SE*cube*™** device acts as the unique administrator of the KMS, everyone else is a user, and there is one **SE*cube*™** device for each user of the system. **SE*key*™** is not a traditional Key Management System where a key server handles requests coming from the users. In **SE*key*™** in fact, it is the administrator that automatically pushes the data managed by the KMS to the users using a dedicated and secure update

channel. Thanks to this approach, the users are automatically provided with the data (i.e. keys) that they need.

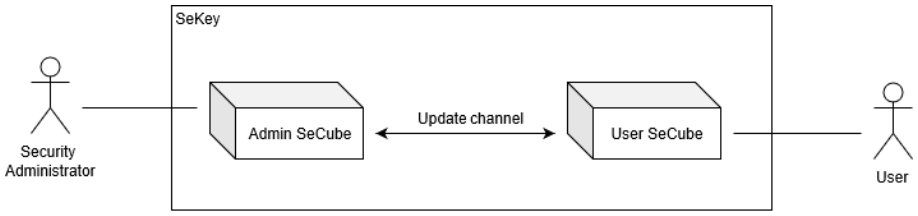


Figure 23: **SE*key*™** context diagram.

As a consequence, **SE*key*™** can be considered a distributed KMS also in terms of the data that are actually managed. Is important to point out, however, that the users are not able to make changes to the data of the KMS; only the administrator has that level of privilege.
Besides cryptographic keys, the KMS also has to manage other metadata in order to work properly, these metadata contain information about the keys, the users, and so on. These data are mainly stored inside the **SE*cube*™** device belonging to the administrator, then they are automatically sent to the devices of the user (with the update channel in Figure 23).
Since the users are automatically provided with the data of the Key Management System by the administrator, they are enabled to work also in 'offline' scenarios because they everything they need on their **SE*cube*™** devices. Notice that the administrator provides the data to the users according to the 'Least Privilege' principle [27]. This means that the administrator discloses, to each user, only the minimum amount of information that is required in order to enable that specific user to use the Key Management System according to the security policies and rules established by the administrator.
At this point, it should be clear that the data managed by the KMS are stored on each **SE*cube*™** involved in the system. From a physical point of view, the following strategy is adopted to store the data:

- the microSD card provided with each **SE*cube*™** device stores an encrypted SQL database (implemented with SQLite and **SE*file*™** , see 6.6) containing most of the metadata of the Key Management System (information about users, groups, keys, etc.);
- the internal flash memory of each **SE*cube*™** device stores, as clear-text, the actual values of the cryptographic keys managed by **SE*key*™** .

This physical data organization scheme is valid both for the administrator and for the users. The only difference, given the different privilege level of the two roles, is that the **SE*cube*™** of the administrator stores each single bit of data that is managed by **SE*key*™** (all keys and all metadata); while the **SE*cube*™** devices of the users only store a fraction of those data, according to the 'Least Privilege' principle (i.e., if a user is not allowed to access to a certain key, the value of that key will not be stored in his **SE*cube*™**).

8.4 Administrator and Users of SE*key*™

Any Key Management System requires to define very clearly which are the roles played by the various actors involved in the KMS. **SE*key*™** provides only two, simple roles: administrator and user.
From a theoretical point of view, the role of the administrator should be played by different entities (i.e., different people) in order to split the knowledge and the access to the system with the

[27] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-57p2.pdf

maximum privilege among different people. To keep things simple, **SE*key*™** provides that there is only one person in charge of managing the KMS, so there is only one administrator. The responsibility of the administrator is to manage the KMS using specific APIs; the administrator manages the users, the groups of users, the cryptographic keys, the security policies, and so on.
On the other hand, the users do not have an active role with respect to the Key Management System. They can use a subset of the APIs of **SE*key*™** in order to perform specific actions (i.e., searching the most secure key to be used in a specific context) but they do not have the possibility to make changes to the KMS (i.e., create a new key). The data of **SE*key*™** that are automatically sent to the users by the administrator are immutable, meaning that they are 'read only' from the perspective of the users. The only scenario that provides the possibility of changing those data is the reception, from the user side, of new updates coming directly from the administrator (i.e., a specific key has been deactivated).
A person who is involved in the Key Management System cannot be both user and administrator at the same time, at least not from the point of view of the hardware resources. If the administrator of **SE*key*™** also needs to be able to act as a normal user of the system, then two different **SE*cube*™** devices must be used; one for acting as administrator and one for acting as user. Notice that the APIs that are reserved to the administrator cannot be called by a user and, similarly, the APIs reserved to the users cannot be called by the administrator.
Besides the usage of a dedicated **SE*cube*™** device for each actor involved in the Key Management System, the definition of the roles within the KMS is carried out by resorting to the login mechanism of the **SE*cube*™** device. Each **SE*cube*™** , in fact, is protected by two PIN codes that can be set during the initialization of the device. Each PIN code grants the access to a different privilege level: administrator or user. Since this mechanism was already implemented by the L1 APIs, **SE*key*™** simply resorts to the login privilege in order to distinguish if a given **SE*cube*™** and the **SE*key*™** APIs are being used in administrator mode or in user mode.
The administrator of **SE*key*™** executes the login on his **SE*cube*™** using the admin PIN, the users of **SE*key*™** execute the login on their **SE*cube*™** devices with their user PIN codes. Depending on the PIN and, therefore, on the privilege access to the **SE*cube*™** , the device will expose different functionalities and **SE*key*™** will enable only selected APIs (i.e. to prevent the possibility of data tampering by the users). Notice that even if a user executes the login as administrator on his **SE*cube*™** , he will not be able to attack the infrastructure of **SE*key*™** because he will not be able to act as administrator with respect to other users. So the overall integrity of the KMS is preserved and, in the worst case scenario, there will just be one user having troubles using properly the KMS (but the system has specific asynchronous procedures to recover the integrity of the data of a user whose **SE*cube*™** contains corrupted or tampered **SE*key*™** metadata and keys).
Notice that the PIN codes of each **SE*cube*™** involved in the KMS are supposed to be initialized by the **SE*key*™** administrator. Moreover, the administrator should disclose to each user only the user PIN of the **SE*cube*™** , while the admin PIN will be stored into the encrypted SQL database containing the metadata of **SE*key*™** . As a result of this approach, the **SE*cube*™** of the administrator will memorize (inside the metadata database of **SE*key*™**) all the PIN codes of all the **SE*cube*™** devices involved in the KMS (except for the **SE*cube*™** of the administrator himself) while the **SE*cube*™** devices of the users will only memorize (inside the metadata database of **SE*key*™**) the single PIN code that can be used to access to the **SE*cube*™** with the user privilege level.

8.5 Logical Overview about Users, Groups, and Keys of SE*key*™

The goal of **SE*key*™** is to allow the secure management and sharing of cryptographic keys between users who have a personal **SE*cube*™** device and belong to a certain environment. Within this context, users can be split into groups, each one associated to a security policy. This approach is useful to replicate the hierarchies found in real-world scenarios, for example in a company where the employees are divided into departments. Notice that each group may have zero, one or multiple

users, a specific user may belong to several groups; moreover, groups may overlap. The security policy associated to each group determines fundamental parameters of that group, in particular:

- the maximum number of keys to which the group can be related (each group owns a set of keys);
- the algorithm used by the keys of the group (so that a group may implement a more strict security policy with respect to another one);
- the default validity period ('cryptoperiod') of the keys of the group (which is adopted if a key does not have a specific cryptoperiod or if its cryptoperiod is longer than the default one).

SE*key*™ is designed so that a single user is never able to access directly to a key, meaning that users cannot 'own' a key. Keys are related to the groups, not to the users, according to the following rules:

- each group is owner of a specific set of cryptographic keys (from 0 to N, where N is the maximum number of keys for that group determined by the security policy);
- a user obtains the access to a cryptographic key only by being part of the group that owns that specific key;
- users who are member of the same group share the access to every cryptographic key of that group;
- each key is owned by only one group (a key cannot be owned by two or more groups, the ownership of a key cannot be transferred from a group to another);
- when a group is cancelled from **SE*key*™** , if there are still key belonging to that group, those keys are flagged as 'zombie' (they become unusable for applying cryptographic protection and their owner is set to 'zombie'; they are not destroyed because they may still be required to decrypt some data).

In conclusion, the logical implementation of **SE*key*™** is quite simple. Users are split in groups with different dimensions and security policies, to each group is associated a specific set of cryptographic keys. A user can access only to the keys associated to the groups to which he belongs, therefore users who have at least one group in common also have common keys (symmetric encryption keys) that can be used to encrypt data to be shared among them (i.e., real-time communications or files). Notice that users can also encrypt data for private usage and not only for data sharing, they simply need to belong to a group with a single member: themselves.
The interconnection of users, groups, and keys allows us to implement security features based on the dimension of each group. The idea is that a smaller group is more secure because less people have access to the keys of that group; according to this assumption, **SE*key*™** offers to its users APIs to retrieve the most secure key to be used in a given scenario (i.e. ,to encrypt data for private usage, to encrypt a real-time communication with another user, to encrypt a file to be shared with an entire group or with multiple users, etc.).

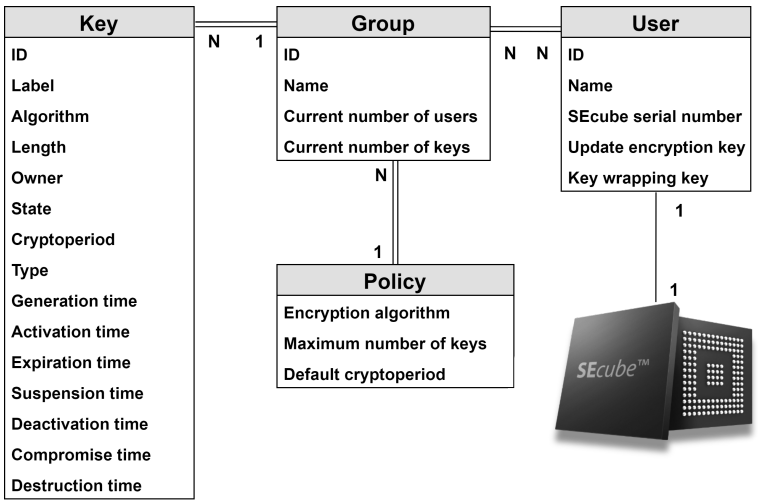


Figure 24: **SE*key*™** logical overview.

8.6 Use Cases

8.6.1 Administrator Use Cases

Here are some use cases that are interesting from the point of view of the **SE*key*™** administrator.

Use case UC1: Add Group
Application: **SE*key*™**
Actor: administrator
Pre-conditions: the administrator is already logged in his **SE*cube*™**
Post-conditions: a new group is added to **SE*key*™**
Main scenario:

1. the administrator calls the API to add a new group to the KMS, providing the required parameters (i.e., the name of the group, the security policy, etc.);
2. the API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e., the administrator requested a group ID that is already in use);
3. if the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new group that is empty and is not owner of any key;
4. a suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

Use case UC2: Add User
Application: **SE*key*™**
Actor: administrator
Pre-conditions: the administrator is already logged in his **SE*cube*™** .
Post-conditions: a new user is added to **SE*key*™** ; an encrypted update file is prepared for the **SE*cube*™** of the new user.
Main scenario:

1. the administrator calls the API to add a new user to the KMS, providing the required parameters (i.e., the name of the user, the desired ID, etc.);

2. the API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e., the administrator requested a user ID that is already in use);

3. if the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new user who still does not belong to any group;

4. a suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

Use case UC3: Add Key
Application: **SE*key*™**
Actor: administrator
Pre-conditions: the administrator is already logged in his **SE*cube*™** .
Post-conditions: a new key is added to **SE*key*™** ; a specific encrypted update file is prepared for the **SE*cube*™** of each user belonging to the group owner of the new key.
Main scenario:

1. the administrator calls the API to add a new key to the KMS, providing the required parameters (i.e., the name of the key, the desired ID, the group owner of the key, etc.);

2. the API validates the correctness of the request and checks for collisions in the current KMS configuration (i.e., the administrator requested a key ID that is already in use);

3. if the request of the administrator is valid, the API modifies the database containing the metadata of **SE*key*™** adding a new key;

4. after storing the metadata of the new key in the encrypted database, the API issues a request to the firmware of the **SE*cube*™** to internally generate a key of the specified length using the True Random Number Generator of the **SE*cube*™** ;

5. the **SE*cube*™** of the administrator generates the key using the TRNG, then the key is stored inside the flash memory of the **SE*cube*™** without being exposed outside of the device;

6. the API checks if the key was successfully generated; if so then it prepares a specifically encrypted update file for each member of the group that has been specified as owner of the new key, inserting into the update file the metadata of the key;

7. the API must also insert the actual key value inside the update, in order to distribute the key to the entitled users; this is done issueing a request to the firmware of the **SE*cube*™** which returns to the API the value of the new key, wrapped using AES-256 with another key that is symmetric and unique for each pair of user and administrator;

8. the API, without having access to the actual value of the new key because it is encrypted, writes it into the encrypted update file for each involved user (notice the double encryption layer around the value of the key, the first one provided with the wrapping of the **SE*cube*™** and the second one provided by **SE*file*™**);

9. a suitable result code is returned to the administrator, who then can assess if the operation was completed or not.

8.6.2 User Use Cases

Here are some interesting use cases from the perspective of the users of **SE*key*™** .

Use case UC4: Encrypt Data

Application: SE*key*™
Actor: user
Pre-conditions: the user is already logged in his SE*cube*™ .
Post-conditions: SE*key*™ returns to the user the ID of the key to be used for encryption.
Main scenario:

1. the user calls one of the APIs of SE*key*™ to retrieve the most secure key to be used providing the recipient(s) (i.e., a single user, unicast communication);
2. SE*key*™ checks the validity of the requests, searches the encrypted database for the smallest group containing both the sender and the receiver, then looks for the active key (i.e. usable for encryption) with the shortest cryptoperiod (assuming that a shorter cryptoperiod implies higher security);
3. if a suitable key is found, the API returns the ID of that key to the user, otherwise an error code is returned;
4. the user checks the return code and, if possible, specifies that key ID as a parameter to the encryption API to encrypt the required data.

Use case UC5: Retrieve known users
Application: SE*key*™
Actor: user
Pre-conditions: the user is already logged in his SE*cube*™ .
Post-conditions: SE*key*™ returns to the user the list of known users.
Main scenario:

1. the user calls the API of SE*key*™ to retrieve the list of known users, meaning the users who have at least one group in common with the current user;
2. SE*key*™ selects the metadata (i.e., the name and the ID) of all the users from the encrypted database stored in the SE*cube*™ device of the current user (notice that the database contains exclusively users with at least one group in common);
3. SE*key*™ fills a list containing all the details about the known users, then it returns an appropriate result code;
4. the user checks the return code and, if possible, uses the details of the known users for his own purposes (i.e., displaying a list of 'contacts' that are available for an encrypted communication).

8.7 SE*key*™ APIs

The API of SE*key*™ is made of many functions; however, some of them are used exclusively for internal purposes. The functions listed in this section are what is generally needed to instantiate and use the Key Management System. For more details about the implementation of the SE*key*™ API, please refer to the Doxygen-based documentation.

```
int sekey_start(L0& l0, L1 *l1ptr);
int sekey_stop();
```

These functions are used to initialize and release the resources that are needed in order to run the KMS. These resources include the pointer to the encrypted database containing the metadata of SE*key*™ , the privilege level (admin or user), and many other details. The Key Management System

cannot work if the `sekey_start()` function is not called; similarly, the `sekey_stop()` function must be called when the KMS needs to be shut down.
Notice that the `sekey_start()` function takes as input parameters the pointers to the L0 and L1 objects that are necessary to communicate with the **SE*cube*™** device connected to the host computer. These two objects should be created by the application that runs the instance of the Key Management System (the end user application in a simple scenario, the deamon containing a system-wide instance of the KMS in a more complex scenario where the KMS is integrated as a service of the operating system).

```
int sekey_admin_init(L1& l1, vector<array<uint8_t, 32>>& pin,
   array<uint8_t, 32>& userpin, array<uint8_t, 32>& adminpin);
```

This function is used to physically initialize the **SE*cube*™** device of the **SE*key*™** administrator. Clearly, this function must be executed by the **SE*key*™** administrator only once, before starting the KMS for the first time. Users of **SE*key*™** cannot execute this function.

```
int sekey_init_user_SEcube(string& uid, array<uint8_t, 32>&
   userpin, array<uint8_t, 32>& adminpin, vector<array<uint8_t,
   32>>& pin);
```

This function is used to physically initialize the **SE*cube*™** device of the user identified by the ID specified as first parameter. The second and third parameter are the two PIN codes to be set on the **SE*cube*™** (only the user PIN should be disclosed to the user, the admin PIN of that specific **SE*cube*™** device is known only by the administrator of **SE*key*™**). The last parameter is a list of PIN codes to be used to login to the **SE*cube*™** of the user before the initialization (by default, both PIN codes of the **SE*cube*™** are 32 bytes long, each byte is initialized to zero).
Notice that the initialization of the **SE*cube*™** device of the user must be done after (but not necessarily immediately after) calling the `sekey_add_user()` function.
This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_add_user(string& user_id, string& username);
```

This function lets the administrator adding a new user to **SE*key*™** , providing a specific user ID and username. If the ID is already used, an error is returned. By default, each user ID is a string beginning with 'U' and followed by a number (i.e., 'U1', 'U342'). By default, a new user does not belong to any group.

```
int sekey_delete_user(string& userID);
```

This function can be used by the administrator to delete a user from **SE*key*™** . Any data related to the user is completely deleted from the **SE*cube*™** of the administrator, then the required changes are propagated to the other users of the KMS (including the user who has just been deleted) by means of the traditional encrypted update channel. Any information related to the deleted user is automatically deleted from the **SE*cube*™** devices of the other users of the KMS. Similarly, any **SE*key*™** related information is completely and automatically cancelled from the **SE*cube*™** of the user who has been deleted.

```
int sekey_add_user_group(string& userID, string& groupID);
```

With this function the administrator can add a user to a group. When the user is added, the other members of the group are automatically notified of the presence of a new user (the administrator

sends the details of the new user to the other members of the group). Similarly, the new user is provided with the details about all the other members of the group and with the keys associated to that group.

```
int sekey_delete_user_group(string& user_id, string& group_id);
```

With this function the administrator can delete a user from a group. After the deletion, the user simply does not belong anymore to the group, but he is still part of **SE*key*™** . Since the user is deleted from the group, all the metadata and cryptographic keys related to the group are cancelled from the **SE*cube*™** of the user. Notice that the KMS automatically removes from the **SE*cube*™** of the user also all the information about other users whose only group in common was the one involved by the deletion procedure.

```
int sekey_user_change_name(string& userID, string& newname);
```

With this function the administrator can change the name of a user.

```
int sekey_user_get_info(string& userid, se_user *user);
int sekey_user_get_info_all(vector<se_user> *users);
```

These functions are used to retrieve the details about a specific user or about all users. The retrieved data are stored into specific objects. This function can be called by the administrator and by the users of **SE*key*™** .

```
int sekey_add_group(string& groupID, string& group_name,
   group_policy policy);
```

This function lets the administrator adding a new group to **SE*key*™** , providing a specific group ID, group name and security policy. If the ID is already used, an error is returned. By default, each group ID is a string beginning with 'G' and followed by a number (i.e., 'G1', 'G342'). By default, a new group does not include any user and does not own any cryptographic key.

```
int sekey_delete_group(string& groupID);
```

With this function the administrator can delete an entire group from **SE*key*™** . Any data related to the group is deleted from the **SE*cube*™** of the administrator and, thanks to the automatic propagation of the changes through the encrypted update channel, from the **SE*cube*™** devices of the users who belong to the group that has to be deleted. The cryptographic keys that were belonging to the deleted group are not cancelled from any **SE*cube*™** device because they may still be needed to decrypt some data. For this reason, these keys are set to the 'deactivated' status (if they are not already in a status that is deactivated, compromised or destroyed) and their owner is set to 'zombie'.

```
int sekey_group_change_name(string& groupID, string& newname);
```

With this function the administrator can change change the name of a group.

```
int sekey_group_change_max_keys(string& groupID, uint32_t
   maxkeys);
```

With this function the administrator can change the maximum number of keys that a group is allowed to own. The provided number of keys must be equal or grater to the current number of

keys.

```
int sekey_group_change_default_cryptoperiod(string& groupID,
   uint32_t cryptoperiod);
```

With this function the administrator can change the default cryptoperiod of the keys belonging to a group. The new cryptoperiod will be used to compute the expiration time of keys that will be activated after the change implied by this function. The expiration time of keys that were activated in the past is not affected by these changes.

```
int sekey_group_get_info(string& groupID, se_group *group);
int sekey_group_get_info_all(vector<se_group> *groups);
```

These functions are used to retrieve the details about a specific group or about all groups. The retrieved data are stored into specific objects. This function can be called by the administrator and by the users of **SE*key*™** .

```
int sekey_add_key(string& key_id, string& key_name, string&
   key_owner, uint32_t cryptoperiod, se_key_type keytype);
```

This function lets the administrator adding a new cryptographic key to **SE*key*™** , providing a specific key ID, key name, key owner, cryptoperiod and key type. If the ID is already used, an error is returned. By default, each key ID is a string beginning with 'K' and followed by a number (i.e., 'K1', 'K342'). The cryptoperiod of the new key is the minimum between the cryptoperiod specified as parameter to this function and the default cryptoperiod of the group that owns the key (if the cryptoperiod parameter is 0, then the default cryptoperiod of the group is used). The key type is always fixed to *symmetric_data_encryption*.

```
int sekey_activate_key(string& key_id);
```

The administrator can use this function to activate a key. When activated, the key becomes usable for applying cryptographic protection to data (i.e., encryption). Following activation, the expiration time of the key is set simply adding the cryptoperiod (kind of like a 'validity time') to the current 'epoch' time. A key can be activated only if its status is 'pre-active' or 'suspended' (in this last case it is a re-activation).

```
int sekey_key_change_status(string& key_id, se_key_status status
   );
```

This function is used to change the status of a cryptographic key, according to Figure 22. Notice that to change the status to 'active', there is the dedicated function called `sekey_activate_key()`.

```
int sekey_key_change_name(string& key_id, string& key_name);
```

This function is used by the administrator to change the name of a key.

```
int sekey_key_get_info(string& key_id, se_key *key);
int sekey_key_get_info_all(vector<se_key> *keys);
```

These functions are used to retrieve the details about a specific key or about all keys. The retrieved data are stored into specific objects. This function can be called by the administrator and

by the users of **SE*key*™** .

```
int sekey_find_key_v1(string& chosen_key, string& dest_user_id,
   se_key_type keytype);
int sekey_find_key_v2(string& chosen_key, string& group_id,
   se_key_type keytype);
int sekey_find_key_v3(string& chosen_key, vector<string>&
   dest_user_id, se_key_type keytype);
```

These functions allow the users to find the most secure key to be used in different scenarios:

- the first function is used in a 1-to-1 communication (i.e. 'U1' wants to send an encrypted message to 'U2') , therefore both parties must share at least one common group;
- the second function is used in a 1-to-N communication, where the sender is a single user and the recipients are all the members of a specific group;
- the third function is used in a 1-to-N communication, where the sender is a single user and the recipients are a set of users. Notice that, if all the users involved (sender and recipients) do not share at least one common group, this function will fail because it will be impossible to find a single key that can be used for all recipients.

The '*keytype*" parameter must be '*symmetric_data_encryption*" (it was added explicitly to make the KMS able to handle also other types of keys in the future). The general rule to find the most secure key is to find an active key belonging to the smallest group containing all the users involved. This does not automatically imply that only the involved users have access to that key; two users may have only one group in common, but that group may also include other users who have access to the chosen key (because they also belong to the group owner of the key) and who are able to decrypt the data. Creating small groups of 2 users to enable secure 1-to-1 communication is considered to be a responsibility of the administrator. These functions can be used only by the users of **SE*key*™** .

```
int sekey_readlog(string* sn, string& output);
```

This function has been added for debugging purposes. It reads the content of the encrypted log file generated by the **SE*cube*™** device with the serial number passed as first parameter and copies its content into the string passed as second parameter. The string can be printed to a normal text file for further analysis. Notice that the log file is generated automatically by the administrator and by each user of **SE*key*™** , the log file is useful to record which actions have been performed in a specific moment. This function can be executed exclusively by the administrator of **SE*key*™** .

```
int sekey_recovery_request(string& user_id, string&
   serial_number);
```

This function can be used by the administrator in order to generate an encrypted update file that contains all the **SE*key*™** information related to a specific user. The user, then, can process the update file in order to restore the integrity of the data of the Key Management System on his **SE*cube*™** .
The purpose of this function is to allow to the administrator to fix, somehow, any error that may arise on the **SE*cube*™** devices of the users. For example, if the user accidentally deletes the metadata database of **SE*key*™** , he can ask to the administrator to issue a recovery procedure carried out by this function. When the user starts again his KMS, any recovery file is processed automatically and the integrity of the KMS is restored.

```
int sekey_check_expired_keys();
```

This is a simple function that iterates over all the keys stored within **SE*key*™** , checking if the expiration time of the key has been reached or not. If it has been reached and the key has not been deactivated yet, then it deactivates the key. Notice that this function has no effect if the key is already in the status 'deactivated', 'compromised', or 'destroyed'. This function is used extensively within **SE*key*™** and **SE*file*™** to ensure that expired keys are never used when they are not supposed to be used. However, this function can be safely called also by higher levels from time to time (but it is not strictly necessary); notice that it can be called by the administrator and the users.

```
int sekey_update_userdata();
```

This function is reserved to the users of **SE*key*™** , it is used internally by the KMS in order to check if there is any update about the Key Management System coming from the administrator. If an update is found, it is processed automatically. The end-user application is not required to call this function explicitly.

9 Getting Started

9.1 The SE*cube*™ System Setup

In this Section, we outline the set of both hardware and software resources required 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.

9.1.1 Hardware resources

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

1. a PC
2. the **SE***cube*™ Open Source SDK
3. the **SE***cube*™ DevKit

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

- 2+ GiB[28] 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[29],
- 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[30]. Your purchase should comprise the following items (Figure 25):

- The St-Link/v2 programmer
- USB 2.0 A to Mini-B cable
- JTAG to SWD cable
- SWIM cable (not needed to program the **SE***cube*™ DevKit)

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

Figure 25: 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 projects 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[31] and Nucleo[32] 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[33].
Your purchase should comprise the following items, depicted in Figure 26:

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

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

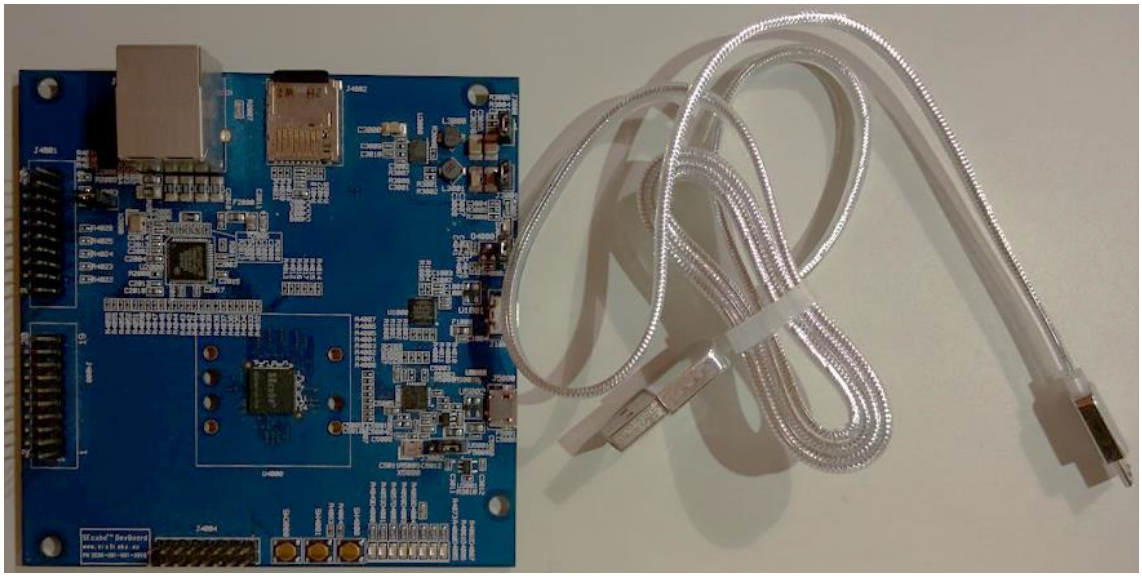


Figure 26: Components purchased with the **SE*cube*™** DevKit

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).

9.1.2 Software resources

You need the following tools:

1. Operating System
2. An IDE of your choice, between the currently supported:
   - Eclipse
   - STM32CubeIDE
   - Keil uVision5
3. Java Runtime Environment (only for Eclipse)
4. AC6 Tools: GNU ARM Embedded Toolchain (only for Eclipse)
5. STM32CubeMX - STM32Cube initialization code generator (only for Eclipse)
6. Lattice Diamond Software (available only on Linux and Windows)
7. ST-Link/v2 drivers (only on Windows)
8. ST-Link Utility (only for Windows) or STM32CubeProgrammer (available for Windows, Linux and MacOS)
9. Open Source SDK

Two **Operating Systems** are currently **fully** supported:

- Windows 7 (or later)[34]

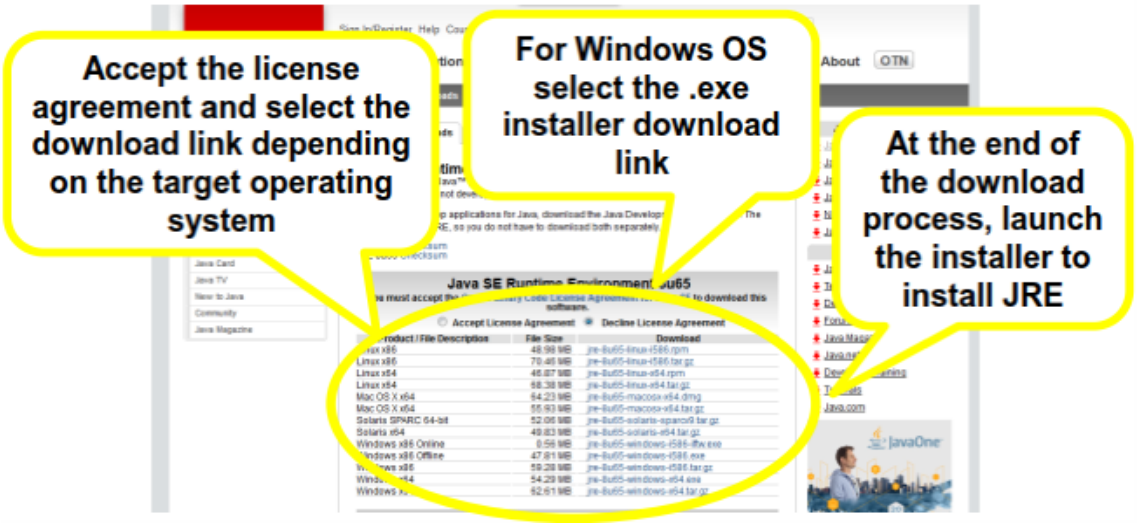
[34] This procedure has already been tested with Windows 10 Professional x64.

- Linux with Kernel 2.6 (or later)[35]
- Mac OS 10.1X [36]

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[37].
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".
What it's going to be used for
Eclipse will be used to develop applications that will run on the STM32F429 processor of the **SE*cube*™** DevKit.

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[38].
Installation hints
Visit the download link and follow the instructions as in the following screenshot:

[35] This procedure has already been tested with Linux Ubuntu 18.04 LTS x64.
[36] This procedure has already been tested with Mac OS Sierra 10.12. Mac OS is currently supported only for the firmware development, while the host-side of the SDK is only supported on Linux and Windows.
[37] https://www.eclipse.org/downloads/
[38] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html



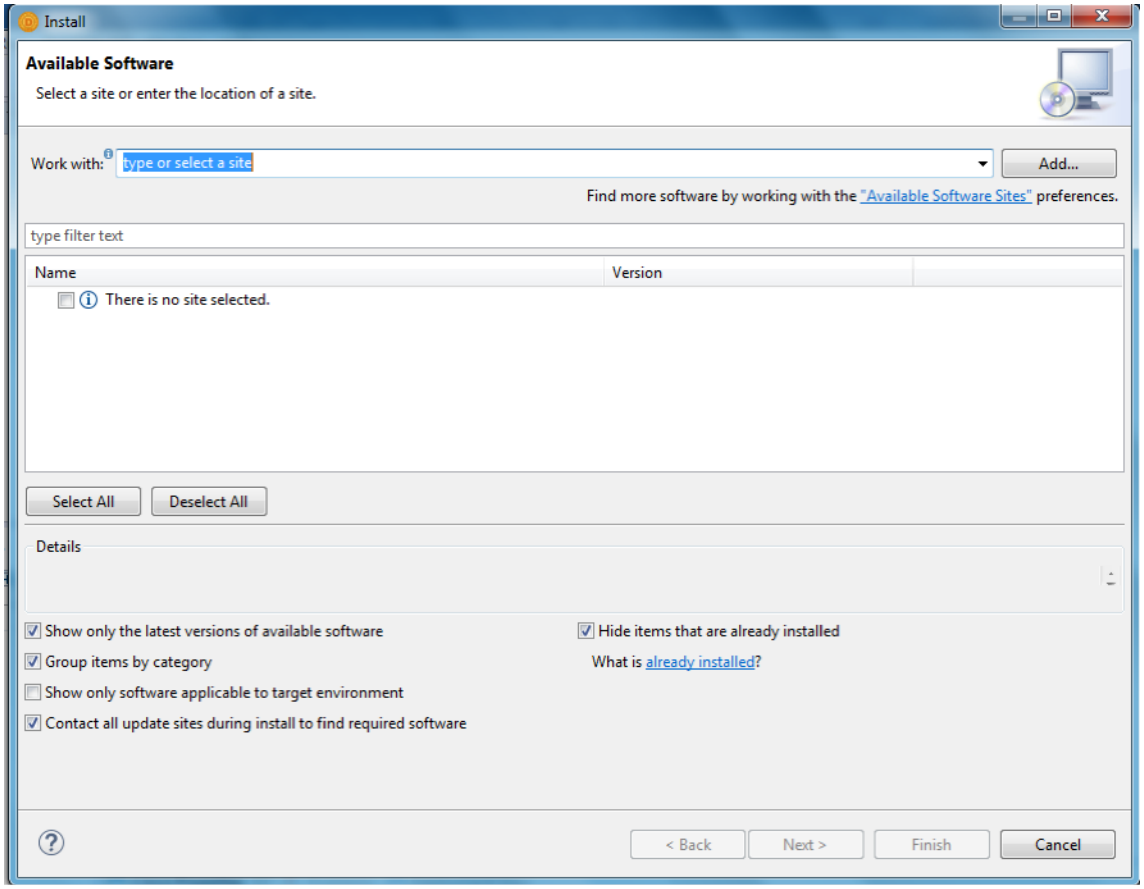
What is going to be used for
The Java Runtime Environment is required for Eclipse to work properly.

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»



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[39].
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

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

STM32CubeIDE is an all-in-one multi-OS development tool, which is part of the STM32Cube software ecosystem.
STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. It is based on the Eclipse/CDT framework and GCC toolchain for the development, and GDB for the debugging. It allows the integration of the hundreds of existing plugins that complete the features of the Eclipse IDE. STM32CubeIDE integrates STM32 configuration and project creation functionalities from STM32CubeMX to offer all-in-one tool experience and save installation and development time.
Version Required
Version 1.0.2 (or later).
How to get it
The program is available free of charge from the STMicroelectronics website[40].
Installation hints
Visit the download link and follow the instructions.
No additional plugins are required to use STM32CubeIDE for development of **SE*cube*™** applications.

[39] 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.
[40] https://www.st.com/en/development-tools/stm32cubeide.html

What it's going to be used for
STM32CubeIDE will be used to develop applications that will run on the STM32F429 processor of the **SE*cube*™ DevKit**.

The **Keil uVision5** IDE combines project management, run-time environment, build facilities, source code editing, and program debugging in a single powerful environment. µVision is easy-to-use and accelerates your embedded software development. µVision supports multiple screens and allows you to create individual window layouts anywhere on the visual surface.
Version Required
Keil µVision5 (or later).
How to get it
The program is available on the Keil website[41], a license must be acquired to use this product; the limit on the project size imposed by the free version is below the size of the **SE*cube*™** firmware.
Installation hints
Visit the download link and follow the instructions.
What it's going to be used for
Keil µVision5 can be used for the development of the **SE*cube*™** firmware. This tool, in particular the compiler MDK-ARM is required if you plan to load your firmware on the **USE*cube*™ Stick**.

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 online[42].
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.
This is only required on Windows.
Version required
Version 4.0.0 (or later).
How to get it
The software is downloadable free of charge online[43].
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.

[41] https://www.keil.com/download/
[42] http://www.latticesemi.com/Products/DesignSoftwareAndIP/FPGAandLDS/LatticeDiamond. In the webpage, also installation guides and instructions on how to retrieve the license are available. Lattice Diamond® is only available for Windows and Linux.
[43] http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link009.html

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[44].
Linux user, instead, can resort to the Open On-Chip Debugger (OpenOCD); this software is downloadable free of charge online[45]. Or can use the STM32CubeProgrammer (see below).
What it's going to be used for
To speed up the usage of the ST-Link/v2 programmer.

The **STM32CubeProgrammer** (STM32CubeProg) is an all-in-one multi-OS software tool for programming STM32 products. It provides an easy-to-use and efficient environment for reading, writing and verifying device memory through both the debug interface (JTAG and SWD) and the bootloader interface (UART, USB DFU, I2C, SPI, and CAN). STM32CubeProgrammer offers a wide range of features to program STM32 internal memories (such as Flash, RAM, and OTP) as well as external memories.
How to get it
The software is downloadable free of charge online from the ST Microelectronics website[46].
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 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/.
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**.

9.1.3 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 27.
At the end of this Section you will have acquired a clear overview of the procedures to follow to start using the environment.

[44] http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link004.html
[45] https://sourceforge.net/projects/openocd/files/openocd/0.9.0/
[46] https://www.st.com/en/development-tools/stm32cubeprog.html

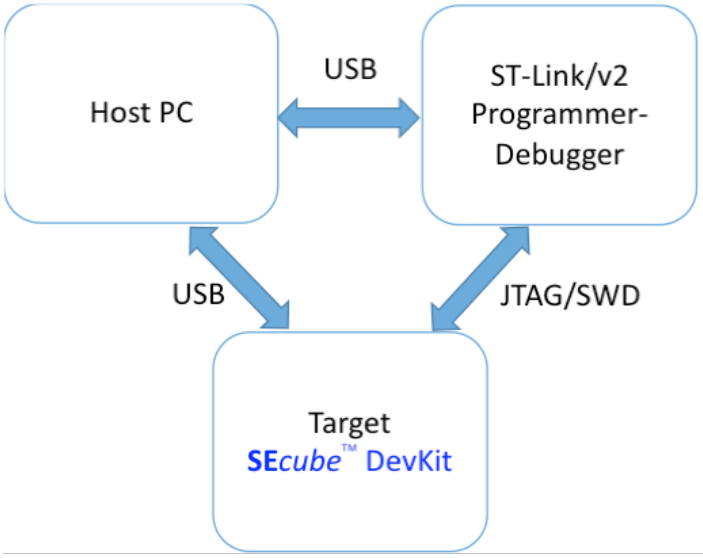


Figure 27: System Architecture

9.1.4 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 28, while a close-up on the JTAG connection is in Figure 29.

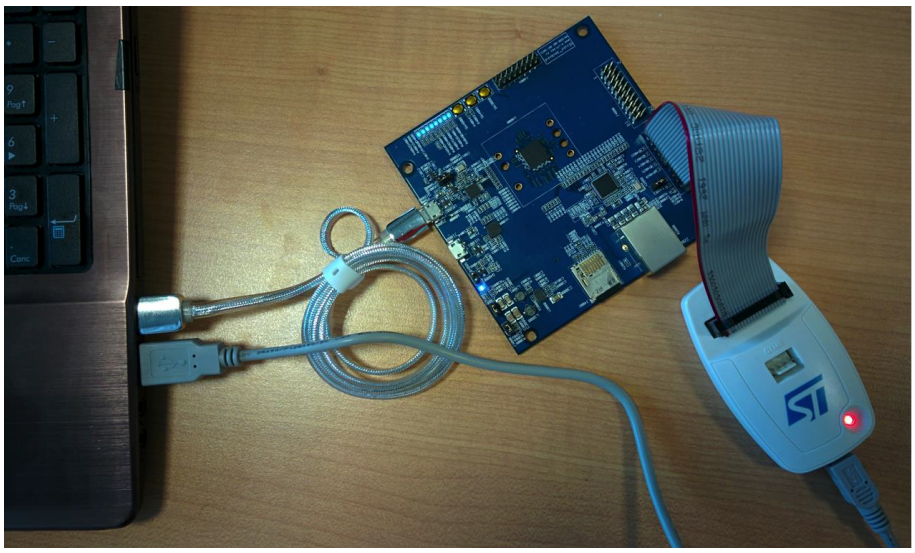


Figure 28: Connection between the STLink/v2 programmer and the **SE*cube*™** DevKit

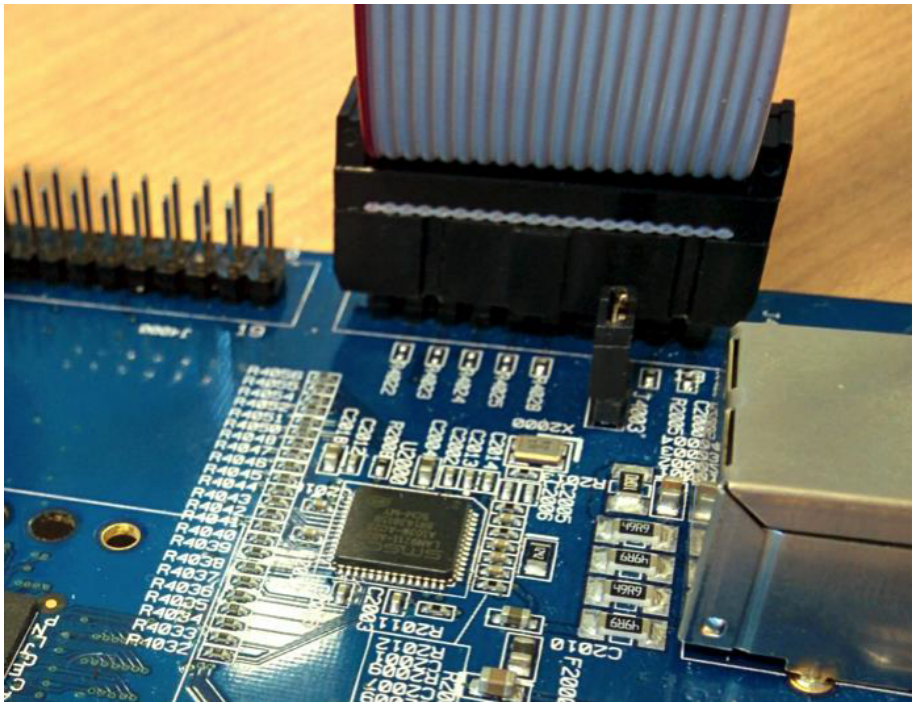


Figure 29: Connection between the STLink/v2 programmer and the **SE*cube*™** DevKit, close-up (highlighted in red) on the JTAG connector orientation

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 31 and described in the STM32 Nucleo-64 boards user manual[47] .

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 30 .

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

[47] Available at the following link, please refer to Section 6.2.4 of the User Manual of the STM32 Nucleo board: https://www.st.com/resource/en/user_manual/dm00105823-stm32-nucleo-64-boards-mb1136-stmicroelectronics.pdf

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 30: Programmer SWD pins schema

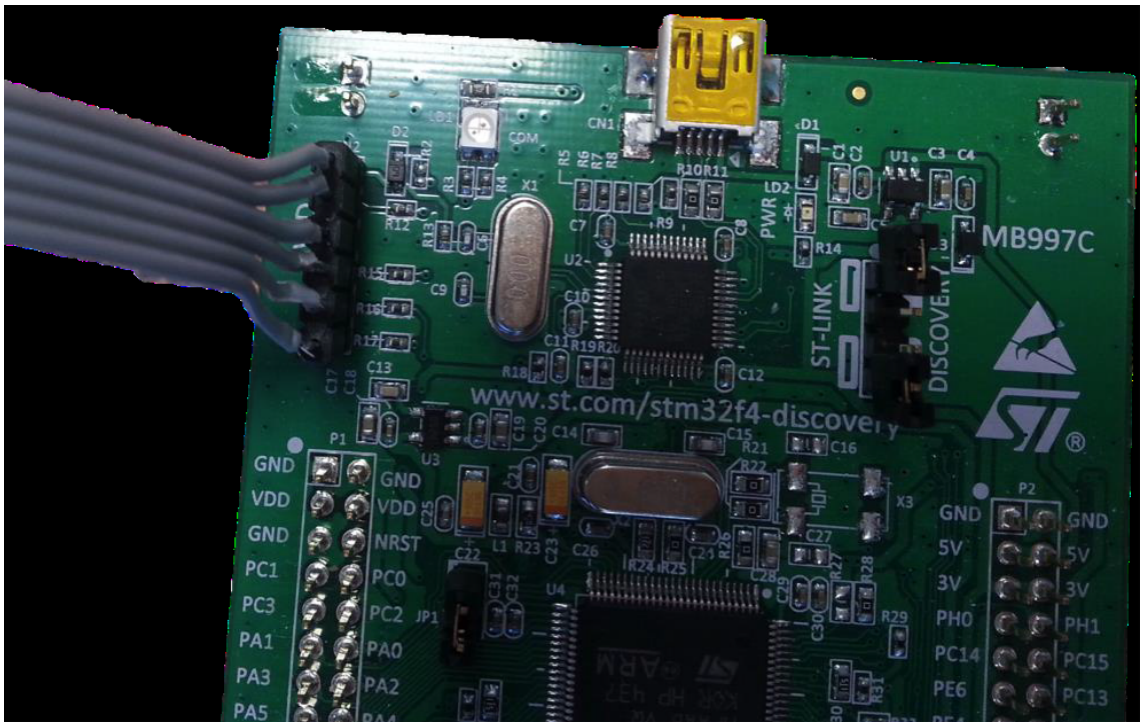


Figure 31: Jumpers configuration to isolate the ST-Link programmer on a Discovery board (highlighted in red, the same applies to Nucleo boards)

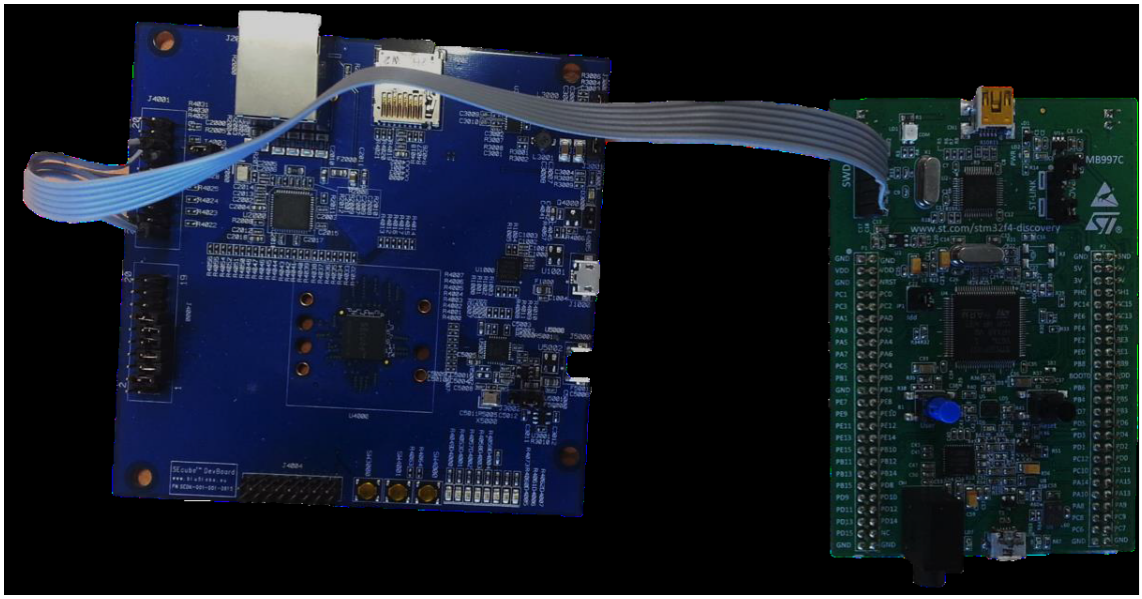


Figure 32: Connection between the Discovery board and the DevKit (the same applies to Nucleo boards)

9.1.5 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.

9.2 Installing the SE*cube*™ OpenSource Software Libraries

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.

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

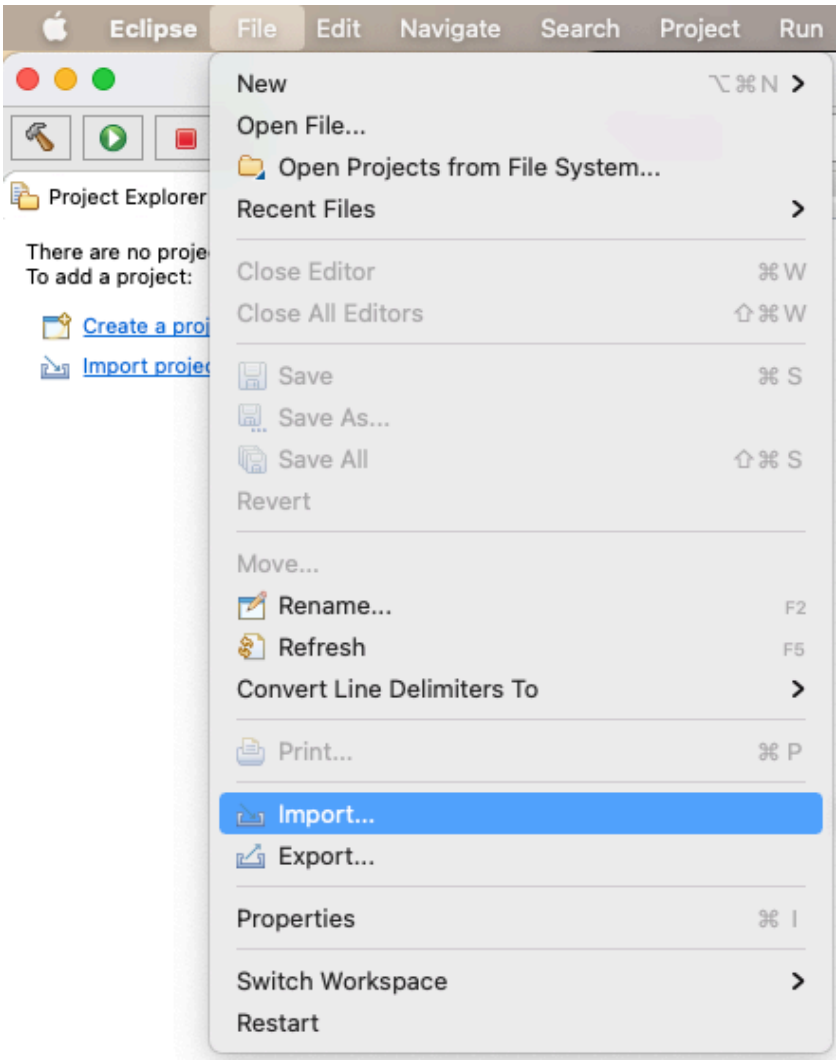
Hereby is listed a step-by-step guide to create the binaries files that will be executed on the **SE*cube*™** DevKit:

1. Download the SDK package containing the project from https://www.secube.eu/resources/open-sources-sdk/
2. Extract the the content of the archive “SEcube SDK v 1.5.1.zip” to a known location
3. From your location, go to “SEcube SDK v 1.5.1” and extract the content of the archive “SEcube USBStick Firmware.zip”

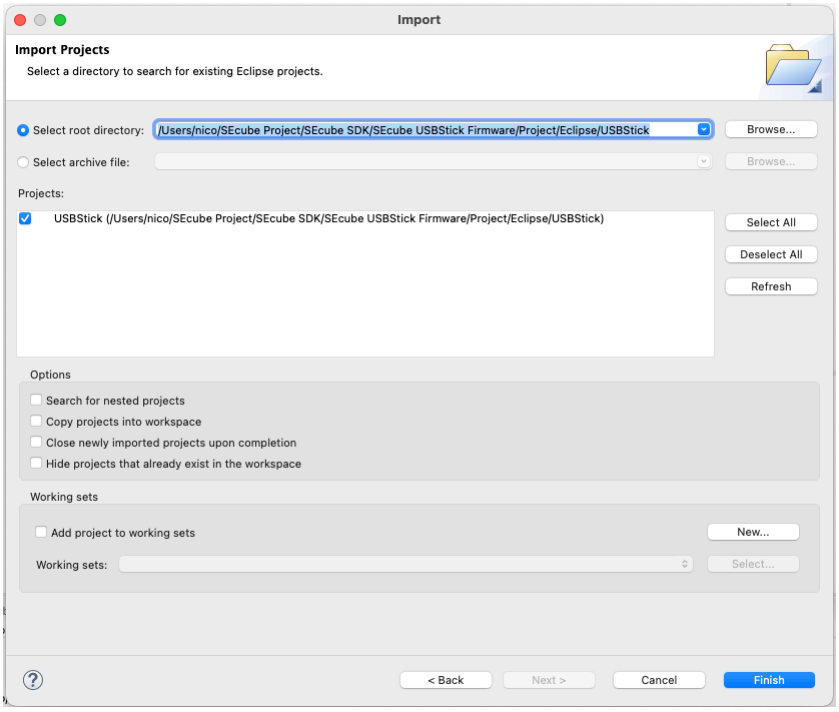
SE*cube*™ Eclipse Import Procedure

1. Launch Eclipse
2. Change Eclipse perspective to C/C++ selecting «Window » Perspective » Open Perspective » Other… » C/C++»

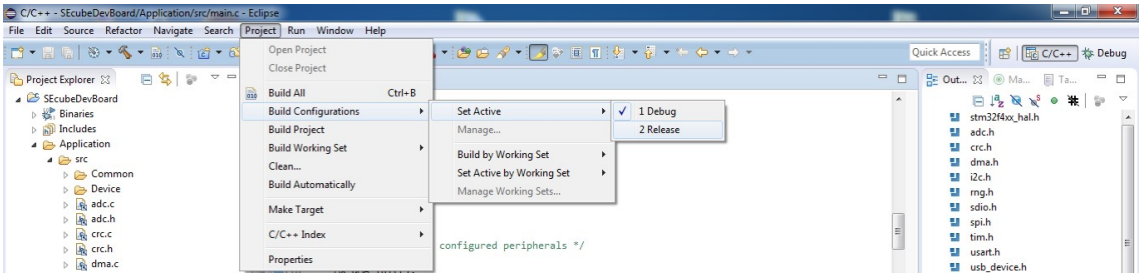
3. Select «File » Import»



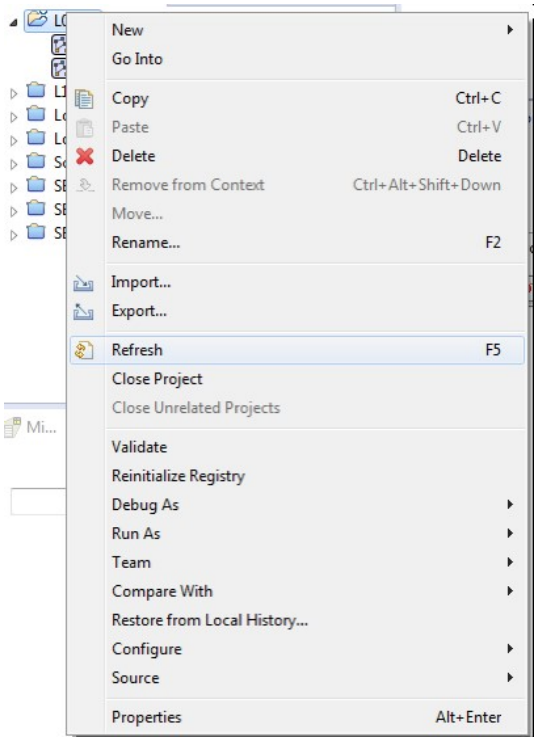
4. In the newly opened window, select «General» and from the dropdown menu select «Existing Project into Workspace»
5. In the Import Project window select «Select root directory» and then «Browse»
6. Navigate in the previously extracted folder to “SEcube USBStick Firmware/Project/Eclipse/USBStick” and select it
7. At this point select the project and click on «Finish»



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



9. Build the project in Debug mode from «Project » Build All»
10. Set the Release configuration from «Project » Build Configuration » Set Active»
11. Build the project in Release mode from «Project » Build All»
12. Right-click on the project in the Project Explorer and select «Refresh»



13. During the compilation process, an error message may appear, indicating that gcc-arm-none-eabi cannot be found. The reason could be that the project is searching for the compiler binaries in the wrong folder. Select «Project » Properties» then navigate to «C/C++ Build » Environment» (as in Figure 33). Modify the "PATH" variable according to the right location for the compilers binaries; check which folders contains the binaries of the compiler[48].

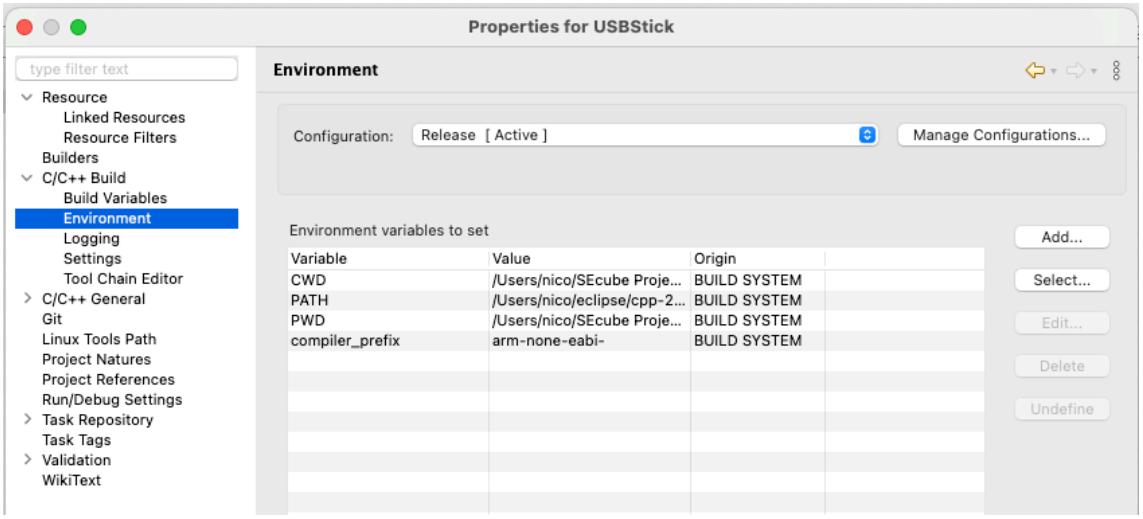
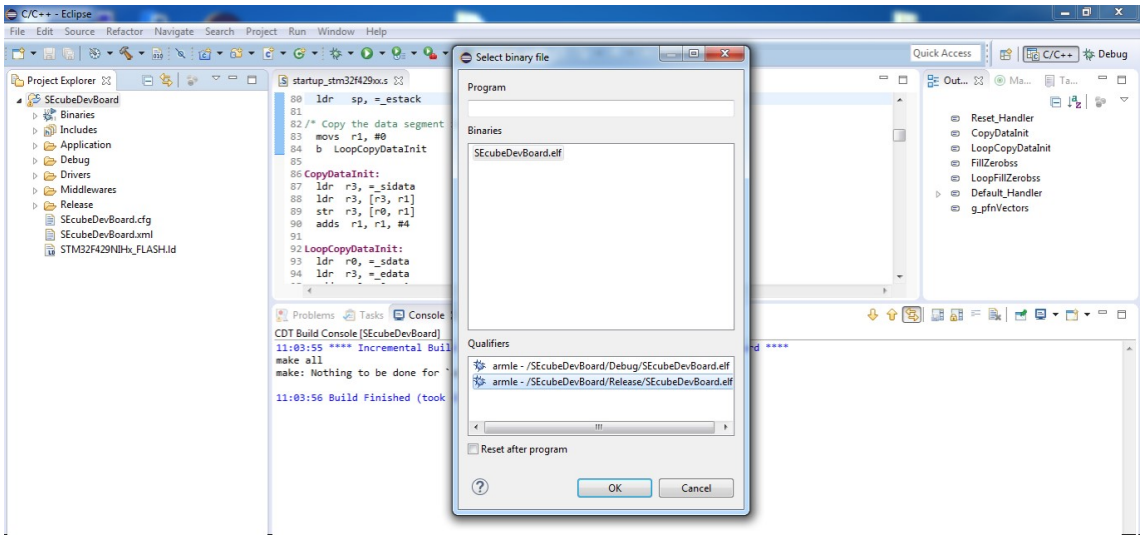


Figure 33: Project Environment Variables

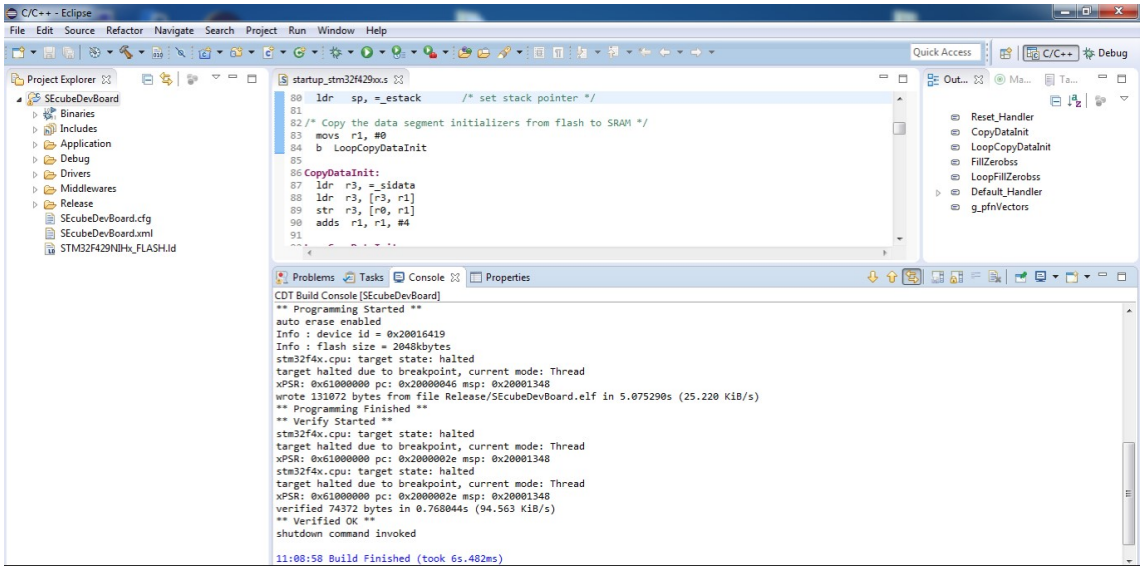
[48] e.g., on Linux <eclipse install dir>/eclipse/cpp-2018-12/eclipse/../../../.p2/pool/plugins/fr.ac6.mcu.externaltools.arm-none.linux64_1.17.0.201812190825/tools/compiler/bin to <eclipse install dir>/eclipse/cpp-2018-12/eclipse/../../../.p2/pool/plugins/fr.ac6.mcu.externaltools.arm-none.linux64_1.13.1.201703061524/tools/compiler/bin.

14. Connect the DevKit as described in previous section

15. 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"). It is always advised to flag the «Reset after program» option



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



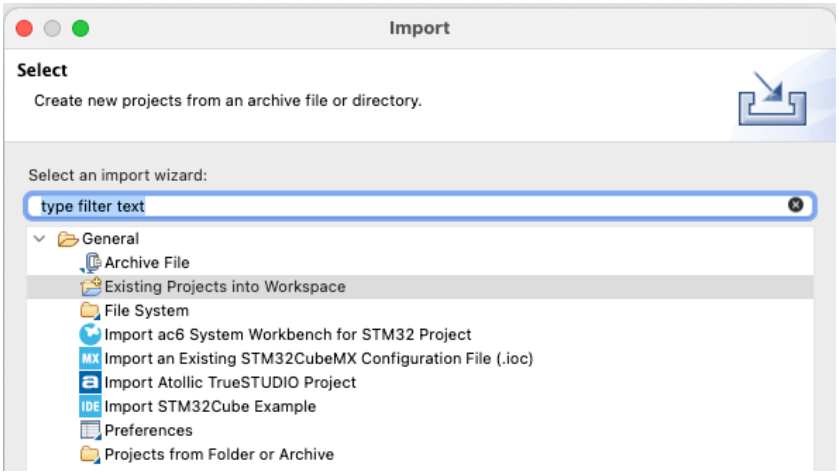
SE*cube*™ STM32CubeIDE Import Procedure

1. Launch STM32CubeIDE

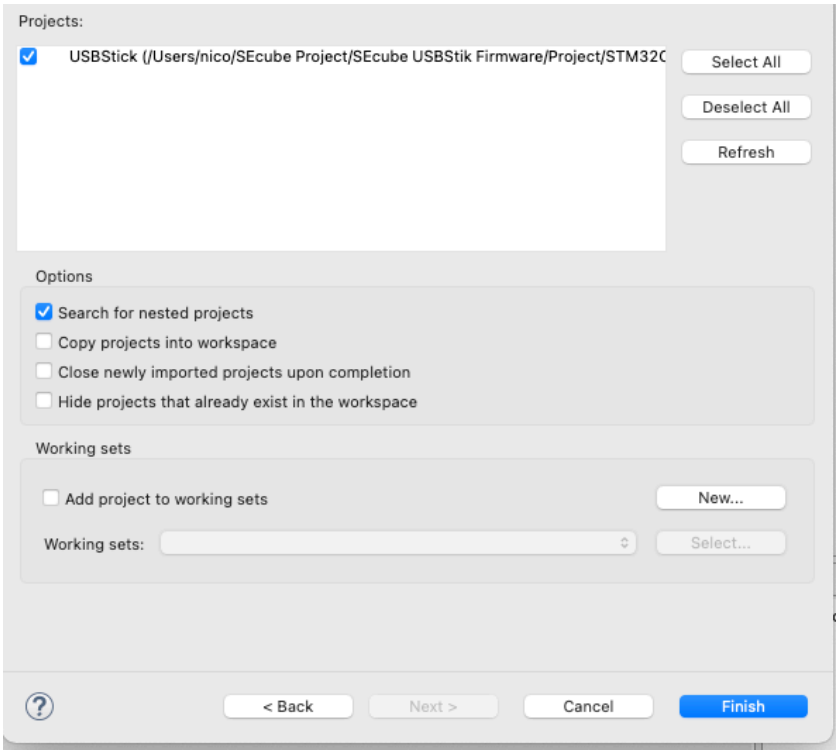
2. You can choose a workspace of your choice for importing the project.

3. Select «File » Import»

4. In the newly opened window ,select «General» and from the dropdown menu select «Existing Project into Workspace»



5. In the Import Project window select «Select root directory» and then «Browse»
6. Navigate in the previously extracted folder to “SEcube USBStick Firmware/Project/ STM32CubeIDE/USBStick” and select it
7. At this point select the project and click on «Finish»

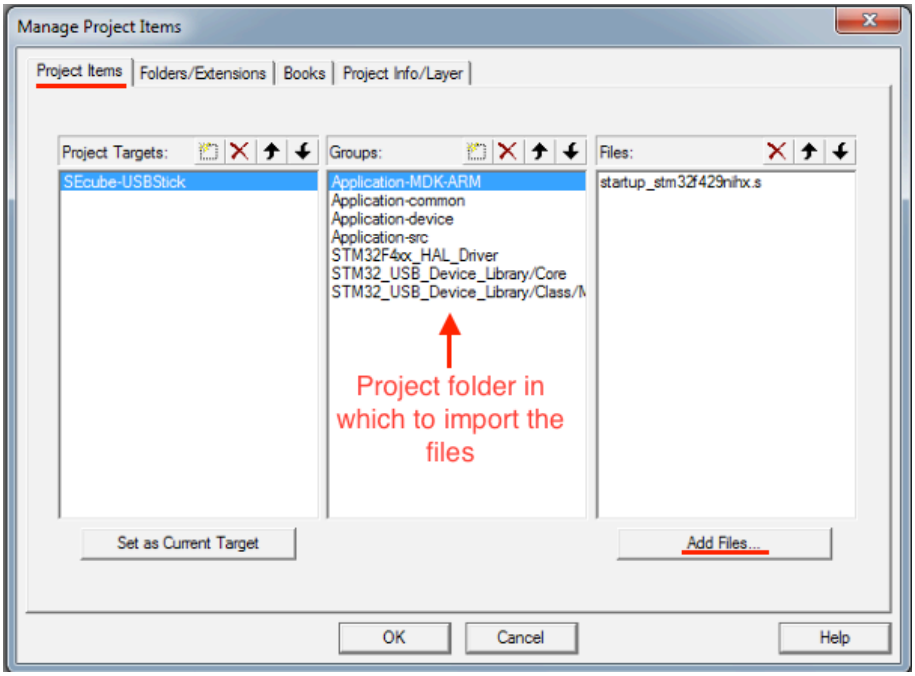


8. The compilation procedure is the same as for Eclipse, being STM32CubeIDE based on Eclipse.
9. Connect the DevKit as described in previous section
10. To load the firmware on the chip, it is possible either to select «Debug». In this way, the firmware is loaded and the chip starts in debug mode. Otherwise, by selecting «Run», the firmware is loaded and the chip execute normally.

SE*cube*™ Keil Import Procedure

1. In the previously extracted folder navigate to “SEcube USBStick Firmware/ Project/MDK-ARM” and then double-click on the file “USBStick.uvprojx”, Keil will be automatically started.
2. To compile the project select «Project » Build» or ”Rebuild”
3. Connect the DevKit as described in previous section
4. To flash the firmware on the device you can select «Flash » Download»

5. To add new files to the project you can select ”Manage Project Items”, you can then select the project and the folder inside the project to which add new files (see figure. At this point you can select ”Add Files” to search the files to add on your system.



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

9.2.2 SE*cube*™ Open Source Software Libraries – Host Side

Hereby is listed a step-by-step guide to create a C project to use leverage on the capabilities of the DevKit:

1. Repeat the steps indicated in 9.2.1 up to point 2
2. From your location, go to “SEcube SDK v 1.5.1/SEcube Host Libraries”
3. Launch Eclipse[49]
4. Change Eclipse perspective to C/C++ selecting «Window » Perspective » Open Perspective » Other… » C/C++»
5. Create a new project with the preferred name, possibly using the rapid choices offered on the «Project Explorer» view on the left
6. Right-click on the project and select «Import…», then select «File System» and press «Next»
7. Browse to the «SEcube Host Libraries» folder and press «Open», then you can select every file present or only a subset of them, and press «Finish»
8. Set your preferred configuration from «Project » Build Configuration » Set Active»
9. Set the compiler to use C++ 17 in «Project » Settings » C/C++ Build » Settings » Tool Setting » GCC Compiler » Dialect»
10. Build the project from «Project -> Build All»
11. Connect the **SE*cube*™** DevKit with the USB cable
12. Now you are ready to experience Device-Host interaction by running one of the two example `main()` present, or you can write your code within a new “main.c” file that you may want to create within the project, compile it and run it.

9.3 Running your first programs

This Section guides you to set up, edit, and use the basic examples and tutorials provided as simple entry points to the environment, leveraging the capabilities provided by the microcontroller. Into the zip file of the SDK that you downloaded, you can find the folder ”SEcube Host Libraries/examples”. Inside this folder you will find examples about:

- **SE*cube*™** device initialization (setup of PIN codes to login, etc...);
- ‘hello world‘ example (including how to login and logout);
- encryption example (how to use APIs to encrypt and decrypt data);
- digest example (how to use APIs to compute cryptographic hashes of data);
- manual key management example (how to use APIs for storing, deleting and using cryptographic keys on the **SE*cube*™** device);
- **SE*file*™** library example;
- **SE*link*™** library example;

[49] Or any other preferred IDE. The following steps are reported for Eclipse. You have to follow functionally-equivalent steps on an alternative development platform.

- **SE*key*™** library example.

Before running any of these examples, it is strongly recommended to carefully read the source code. Looking at the source code, you will be able to understand how to write a program for the **SE*cube*™** Open Source SDK.
An additional step is also required to compile the examples correctly: you need to rename the function that implements the example in the source code (i.e. `helloworld()`) to `main()`.

9.3.1 Device initialization example

The first example that you should run is called "device_init.cpp". This example is used to initialize the PIN codes on your **SE*cube*™** . Each **SE*cube*™** has 2 PIN codes, one for logging in as administrator and one for logging in as user. By default, each PIN code is simply set to a 32 byte array equal to 0.
Each **SE*cube*™** is also characterized by a *serial number*, this example initializes it as well. Here are the steps you need to follow:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file "device_init.cpp"
4. Rename the `device_init()` function to `main()`
5. Check out the value of the variable `pin`, which is the PIN that will be set both for admin and user privilege levels (you can use other PIN values if you want)
6. Check out the value of the variable `serial_number` and change it if you want to setup a different serial number on the **SE*cube*™**
7. Check out the entire source code of the example in order to understand how it works
8. Connect the DevKit with USB cable
9. Set the 'Release' configuration from «Project » Build Configuration » Set Active»
10. Build the project from «Project » Build All»
11. Run the project

If the example is completed correctly, then you have successfully setup your **SE*cube*™** device with the PIN codes and the serial number that you saw in the source code of the example itself. Now the **SE*cube*™** is ready to be used for the other examples.

9.3.2 Hello World example

The second example is a simple 'hello world' program that tests the usage of login and logout APIs. To run the example, follow these steps:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file "hello_world.cpp"
4. Rename the `helloworld()` function to `main()`

5. Check out the value of the variable `pin_admin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works
7. Connect the DevKit with USB cable
8. Set the 'Release' configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

This example, as already stated, is extremely simple. It just shows how to login and logout on the **SE*cube*™** device.

9.3.3 Manual key management example

This example shows how to use the APIs of the **SE*cube*™** SDK in order to manually manage cryptographic keys stored in the internal memory of the **SE*cube*™** device. This kind of key management is completely different with respet to what is offered by the **SE*key*™** library. In order to run this example, follow these steps:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file "manual_key_management.cpp"
4. Rename the `manualkey()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, pay attention to the APIs used to create, list and delete the keys
7. Connect the DevKit with USB cable
8. Set the 'Release' configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

Notice that key management should be done manually only if you have few keys to handle. Check out also the documentation about **SE*key*™** to discover what a Key Management System can do.

9.3.4 Encryption example

This example shows how to use the APIs of the **SE*cube*™** SDK in order to encrypt and decrypt data using keys stored in the internal memory of the **SE*cube*™** device. In order to run this example, you need to use a **SE*cube*™** device that has at least one usable key in its memory. To understand how you can create such a key, please refer to the manual key management example. Here are the steps to execute the digest example:

1. Launch Eclipse

2. Import the project as described in Section 9.2.2
3. Open the file “encryption_algorithms.cpp”
4. Rename the `encrypt_example()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, pay attention to the APIs and data structures used to encrypt and decrypt the data
7. Connect the DevKit with USB cable
8. Set the ‘Release‘ configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

9.3.5 Digest algorithms example

This example shows how to use the APIs of the **SE*cube*™** SDK in order to compute the digest of data using SHA-256 and HMAC-SHA-256 algorithms. In order to run the latter algorithm, you need to use a **SE*cube*™** device that has at least one usable key in its memory. To understand how you can create such a key, please refer to the manual key management example. Here are the steps to execute the encryption example:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file “digest_algorithms.cpp”
4. Rename the `digest_example()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, pay attention to the APIs and data structures used to compute the digest
7. Connect the DevKit with USB cable
8. Set the ‘Release‘ configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

9.3.6 SE*file*™ example

This example shows how to use the APIs of **SE*file*™** in order to create encrypted files with the **SE*cube*™** device. Here are the steps to execute the **SE*file*™** example:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2

3. Open the file “sefile_example.cpp
4. Rename the `sefile_example()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, this is absolutely crucial since **SE*file*™** includes many APIs
7. Connect the DevKit with USB cable
8. Set the ‘Release‘ configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

Please read the documentation about **SE*file*™** before running this example. It is strongly suggested to carefully analyze the source code of this example in order to understand how the **SE*file*™** APIs work. Notice that this example provides also a simple overview about how the Secure Database library can be used, since it is based on **SE*file*™** .

9.3.7 SE*link*™ example

This example shows how to use the APIs of the **SE*file*™** in order to encrypt ‘data in motion‘. Here are the steps to execute the **SE*file*™** example:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file “selink_example.cpp”
4. Rename the `selink_example()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, pay attention to the APIs and data structures of **SE*link*™**
7. Connect the DevKit with USB cable
8. Set the ‘Release‘ configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

Please read the documentation about **SE*link*™** before running this example. It is strongly suggested to carefully analyze the source code of this example in order to understand how the **SE*link*™** APIs work.

9.3.8 SE*key*™ example

This example shows how to use the APIs of **SE*key*™** in order to instantiate a Key Management System for the **SE*cube*™** device. It is strongly recommended to carefully read the documentation about **SE*key*™** before running this example. You should also analyze very carefully the souce code of the file "sekey_example.cpp" before running the code. Here are the steps to execute the **SE*key*™** example:

1. Launch Eclipse
2. Import the project as described in Section 9.2.2
3. Open the file "sekey_example.cpp"
4. Rename the `sekey_example()` function to `main()`
5. Check out the value of the variable `pin` and change it to the value of the PIN code that you set running the device initialization example
6. Check out the entire source code of the example in order to understand how it works, pay attention to the APIs and data structures of **SE*key*™**
7. Connect the DevKit with USB cable
8. Set the 'Release' configuration from «Project » Build Configuration » Set Active»
9. Build the project from «Project » Build All»
10. Run the project

9.3.9 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 led blink.
Hereby we list a step-by-step guide to run this program:

1. Import the project as described in Section 9.2.1
2. Edit the code in "main.c" file, adding the following include at the beginning:

```
#include "FPGA.h"
```

3. Be sure that functions `MX_GPIO_Init()` and `MX_FMC_Init()` are inserted in the `main()`
4. In the `main()` body, after the `device_init()` call, add a call to the `B5_FPGA_Programming()` function
5. Open the file named "gpio.c" and be sure that the following lines within the `MX_GPIO_Init()` function are present. These are needed for configuring the JTAG port used for programming the FPGA:

```
/*Configure GPIO pins : PE3 PE5 PE4 PE6 */
GPIO_InitStruct.Pin = FPGA_TCK_Pin|FPGA_TDI_Pin|FPGA_TMS_Pin
   |FPGA_PROGRAMN_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
```

```
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = FPGA_TDO_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(FPGA_TDI_GPIO_Port, &GPIO_InitStruct);
```

6. Save the changes to all files and build the project
7. Connect the DevKit as described in previous section
8. Run the project following the IDE-specific steps that are described in Section 9.2.1

At the end of the programming, you should notice all LEDs turning on in *weak on* state. This is characterized by a blue light which is not at its most. This indicates that the FPGA is under programming.
The programming process might require some time (1-2 minutes) to be completed. After that, you should see that:

- LED7 and LED6 are blinking;
- LED1, LED2 and LED3 are on.

This is due to the design preloaded in the "TEST_FPGA.h" bitstream.

9.4 From the SE*cube*™ DevKit to the USE*cube*™ Stick

To migrate your project from the **SE***cube*™ DevKit to the **USE***cube*™ Stick, you do not need any programmer. You have just to use the boot loader embedded in the **USE***cube*™ Stick and the **SE***cube*™ USB Loader Free Software available online[50].
This software allows injecting your binary image file into the internal **SE***cube*™ flash, starting from the address 0x08020040. Once you decided the starting address, be sure that the same address is configured in the linker.
In order to guarantee the maximum security level, the bootloader allows the final image to take the full control of the hardware. On this purpose your image shall remap the interrupts vector table as soon as possible, as shown in the following code:

```
/* new vector table in RAM */
uint32_t vectorTable_RAM[256] __attribute__(( aligned (0x200ul)
   ));

/* vector table ROM  */
extern uint32_t __Vectors[];

int main(void)
{
  // Hardware Abstraction Layer Initialisation
  HAL_Init();

  // Configure the system clock
```

[50] www.secube.eu

```
  SystemClock_Config();

  // Remapping Interrupt Vector (overload the USB Loader
     Interrupt Vector)
  uint32_t i;
  for (i = 0; i < 256; i++) {
    vectorTable_RAM[i] = __Vectors[i];
  /* copy vector table to RAM */
  }
  // Interrupt Remapping
  __disable_irq();
  SCB->VTOR = (uint32_t)&vectorTable_RAM;
  __DSB();
  __enable_irq();

  SystemCoreClockUpdate();
  ...
}
```

As shown in Figure 34, the injection can be executed through the USB Loader Tool, which comes together with the **USE*cube*™ Stick**.

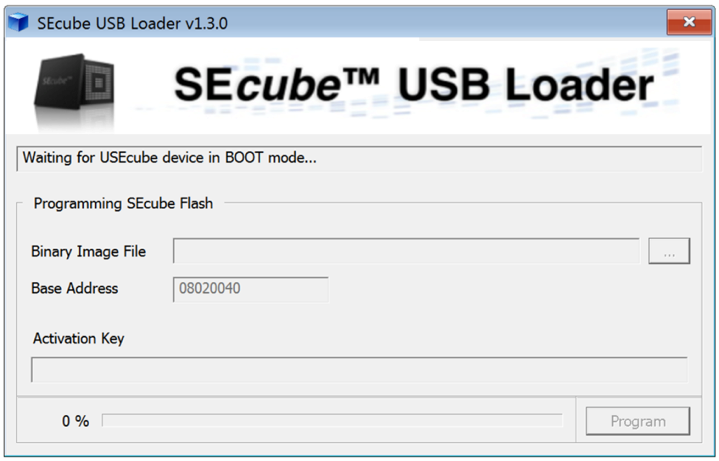


Figure 34: **SE*cube*™** USB Loader.

The USB Loader Tool recognizes the **USE*cube*™ Stick** in Boot Mode, which means that the firmware image has not been yet injected, and allows developers/users to inject a custom binary image in the internal flash. The base address is parametric and, as stated before, must be coherent with the linker settings.
To prevent unauthorized people to inject firmware in your **USE*cube*™ Stick**, a unique Activation Key is delivered when the device is acquired.
Please note that the procedure described in this Chapter is still under testing and it is provided as a reference example, only.
It is very important that your binary image is well tested to run properly on the final device. It is also suitable for your firmware to support an in-line programming functionality to allow users to update the **USE*cube*™ Stick** firmware in the future.
To demonstrate this functionality, the SDK provides `L0_bootmode_reset()`: a dedicated API,

that invalidates the signature of the firmware. Consequently, on the following startup, the **USE*cube*™** Stick will be again recognized as a device in Boot Mode, allowing the injection of a new firmware. An example of usage of this function is shown in the following code.

```
int main() {
    se3_disco_it it;
    se3_device dev;
    L0_discover_init(&it);

    while (L0_discover_next(&it)) {
        se3_device dev;

    // Open SEcube device
    if (SE3_OK != L0_open(&dev, &it.device_info, 1000)) {
        //ERROR
        return -1;
    }

    // BOOT MODE RESET
    if (SE3_OK != L0_bootmode_reset(&dev, my_sn)) {
        //ERROR
        return -2;
    }

    // Close SEcube device
    L0_close(&dev);
    //SUCCESS
    return 0;
}
```

9.5 Getting Started with configuring the internal FPGA

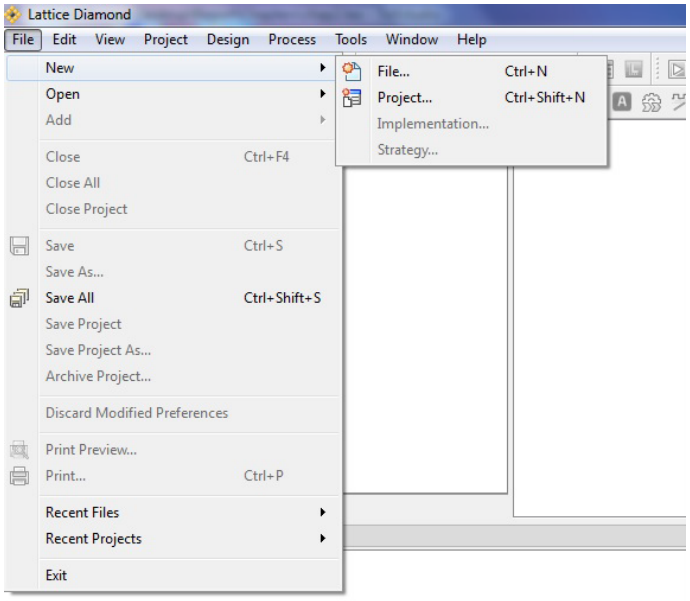
9.5.1 How to import your own project

By reading first Chapter 5 and then the example listed in 9.3.9, you should have understood how the CPU-FPGA communication system should work. Apart from the programming library for the FPGA (composed by the files “FPGA.h” and “TEST_FPGA.h” included in “Project/Inc” and by the file “FPGA.c” in “Project/Src”) and the correct GPIO settings for the FPGA, you 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 following.

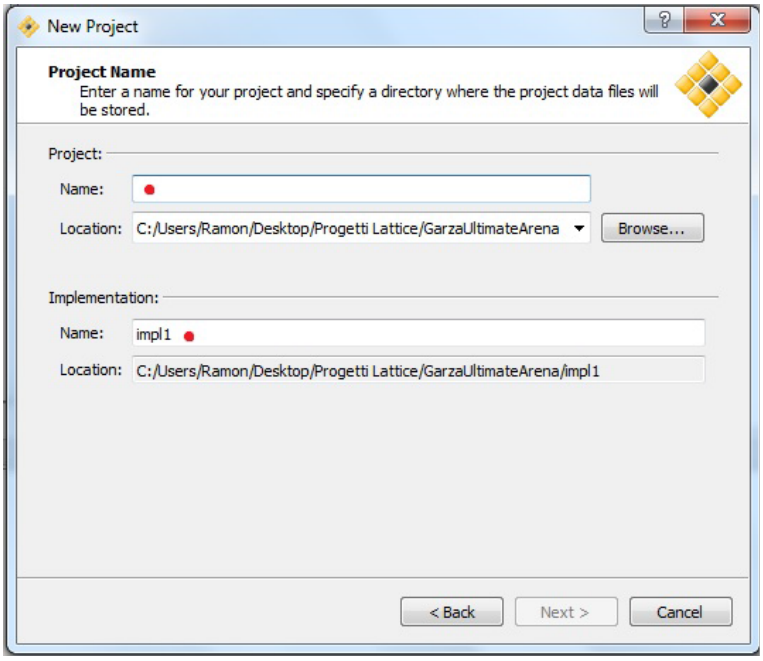
9.5.2 How to create a Lattice Project

Once your VHDL design has 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.

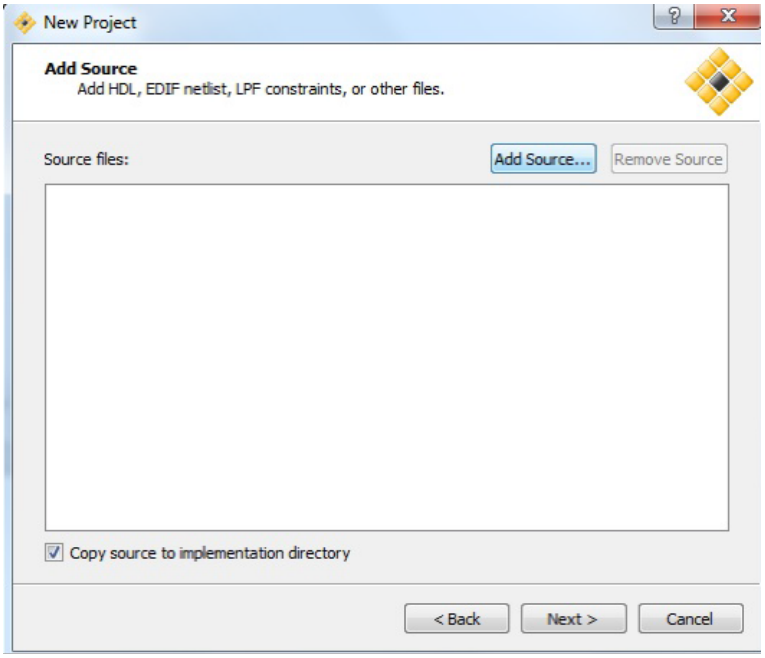
1. Create a new project



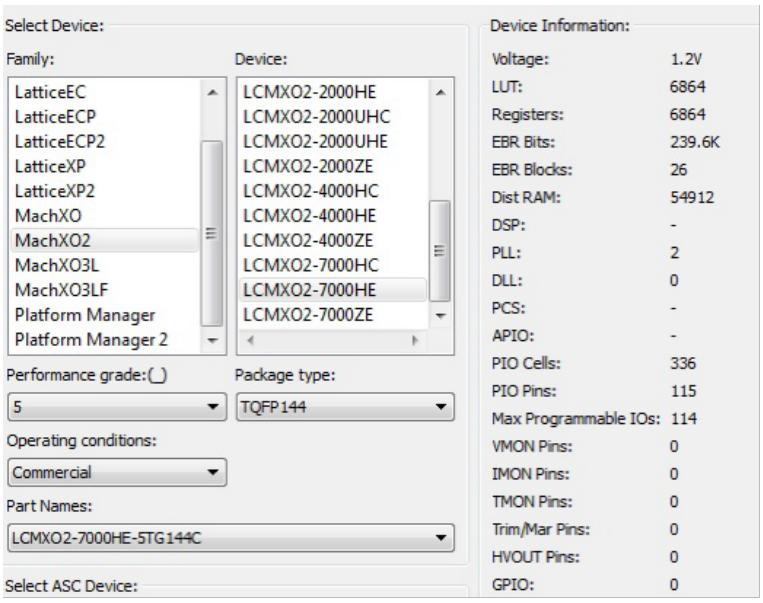
2. Browse to the location of your HDL project folder and insert an implementation name



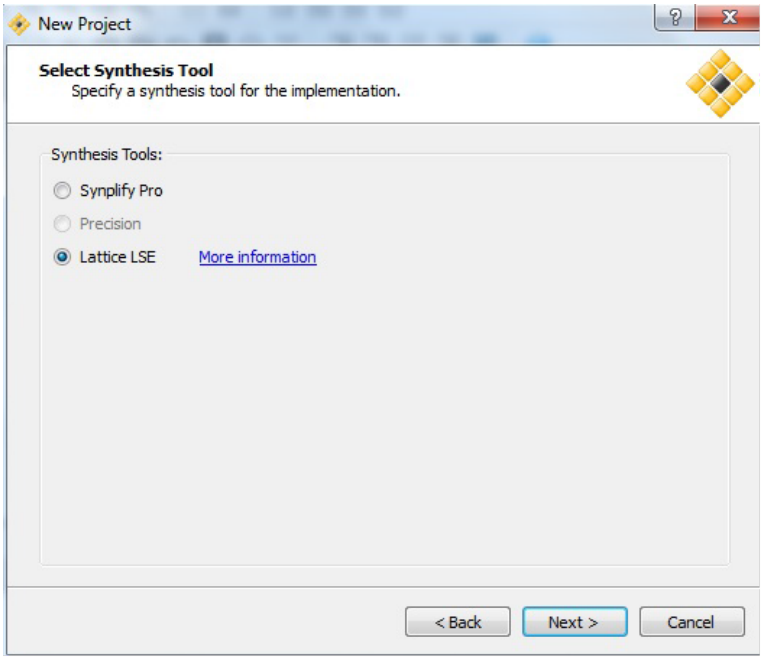
3. Add VHDL source files in this step or inside the project by right clicking on the input files folder



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



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



6. Open and edit the LPF source file in the section «LPF Constraint File». This file is really important, as it is used for mapping I/O signals to pins and for configuring parameters as the target clock frequency. The file is structured as a set of non-sequential commands. The command FREQUENCY is used to set the target speed of the design. The command LOCATE COMP is used to map ports of the top entity to I/O pins of the FPGA. The format is

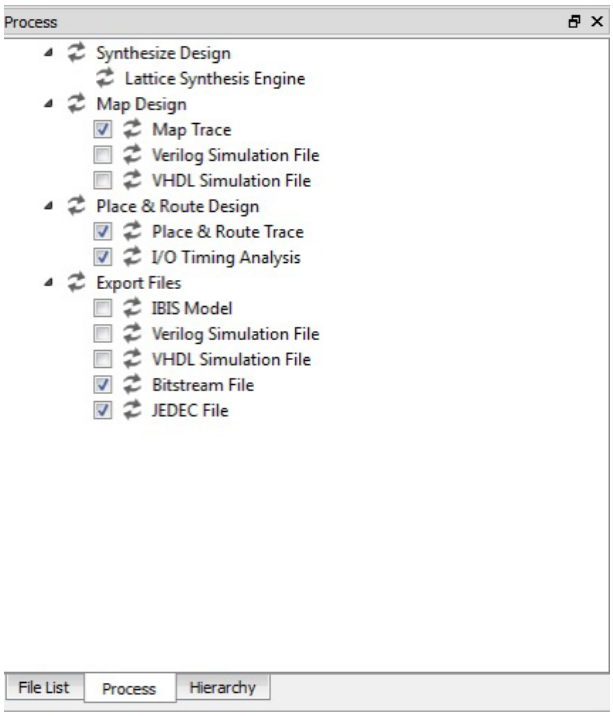
```
LOCATE COMP "<name_of_the_port[bit_number]>" SITE "<pin_name
   >"
```

   Please refer to Appendices A and B to see the corrispondance between pins and physical signals on the **SE*cube*™** DevKit.

7. Save all current changes

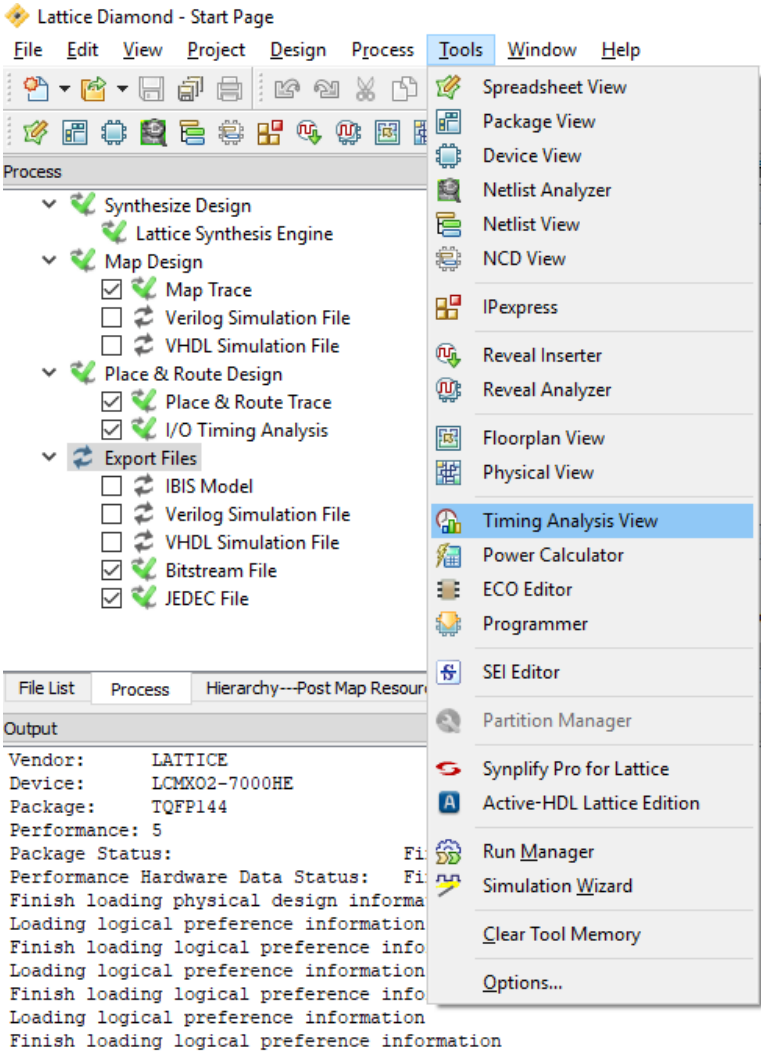
9.5.3 Synthesis Procedure

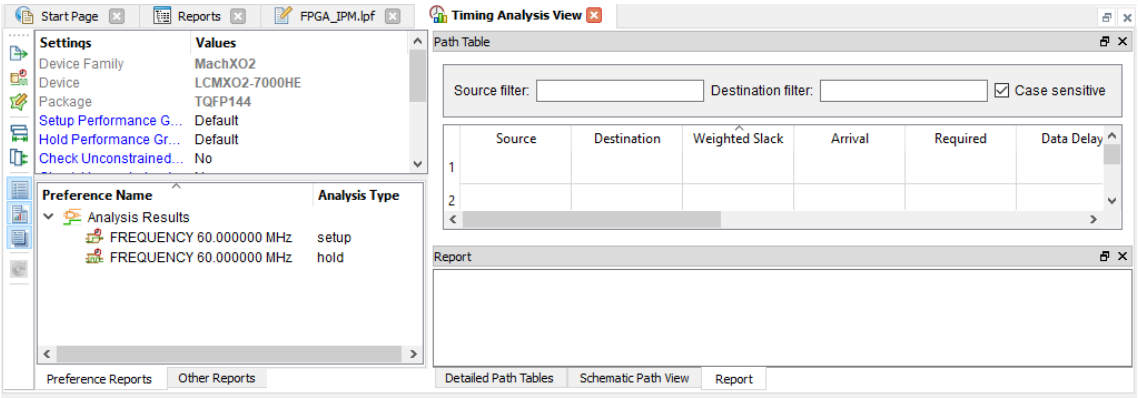
1. Go to «Process» tab of the Process Window and select the check marks as in the following.



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

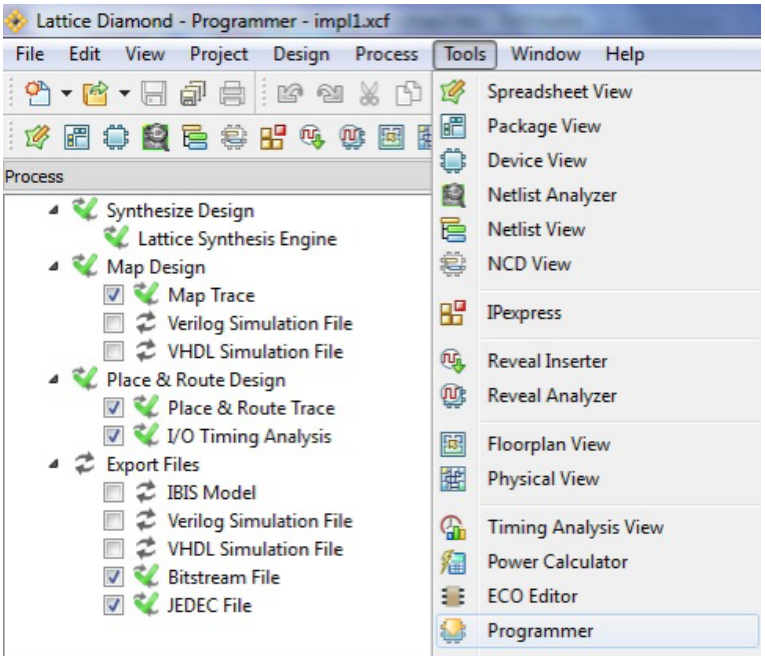
3. At the end of all synthesis steps, go to «Tools » Timing Analysis View» and check if there are no violation for setup and hold times



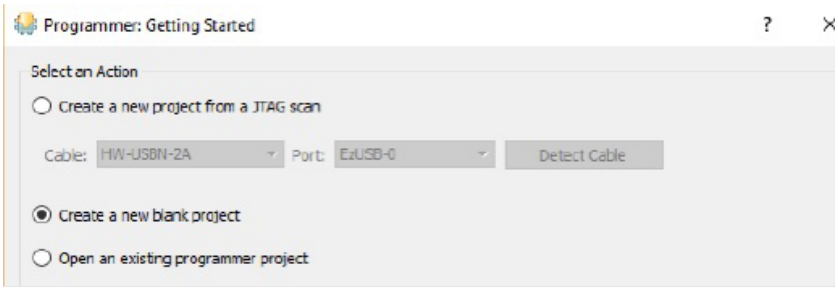


If times are respected, constraints are written in black. Otherwise, they are written in red. By clicking on the constraint, on the right it is possible to see details about the violating path

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

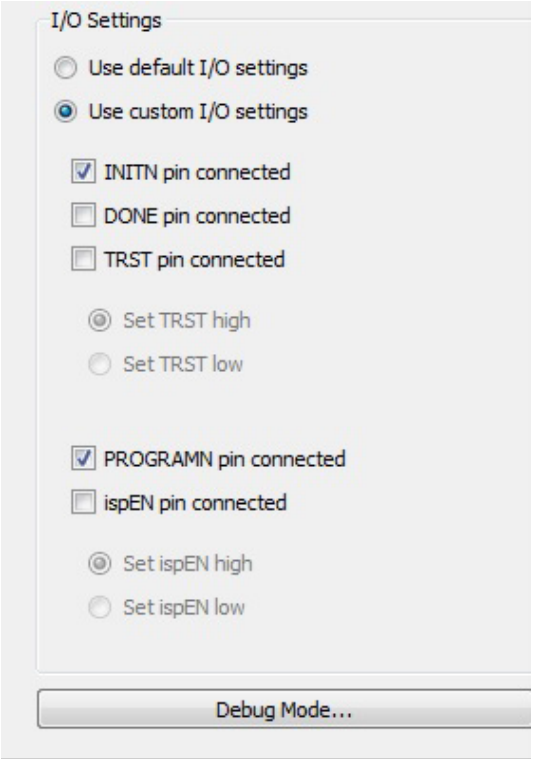


5. Select «Create a new blank project» on the appearing window



6. Verify the device family and the XCF file name

7. Check «I/O Settings» as in figure

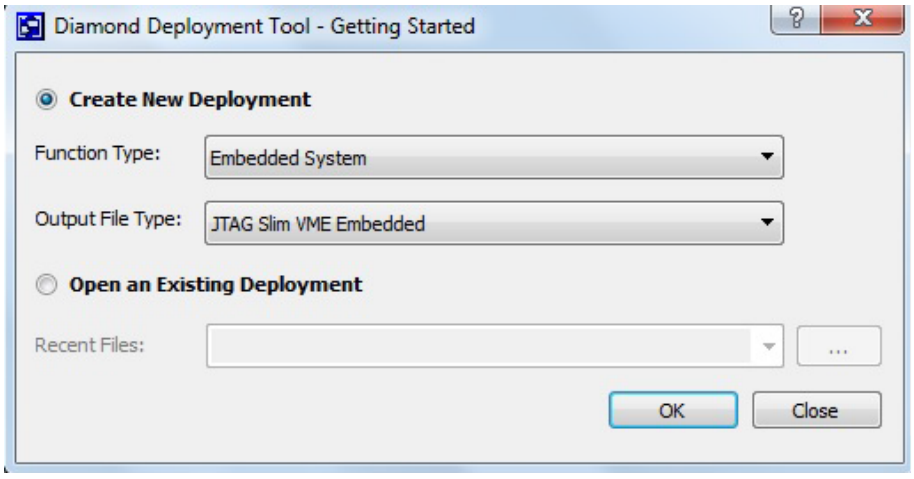


8. Save the changes applied. This file is necessary for Deployment Tool.

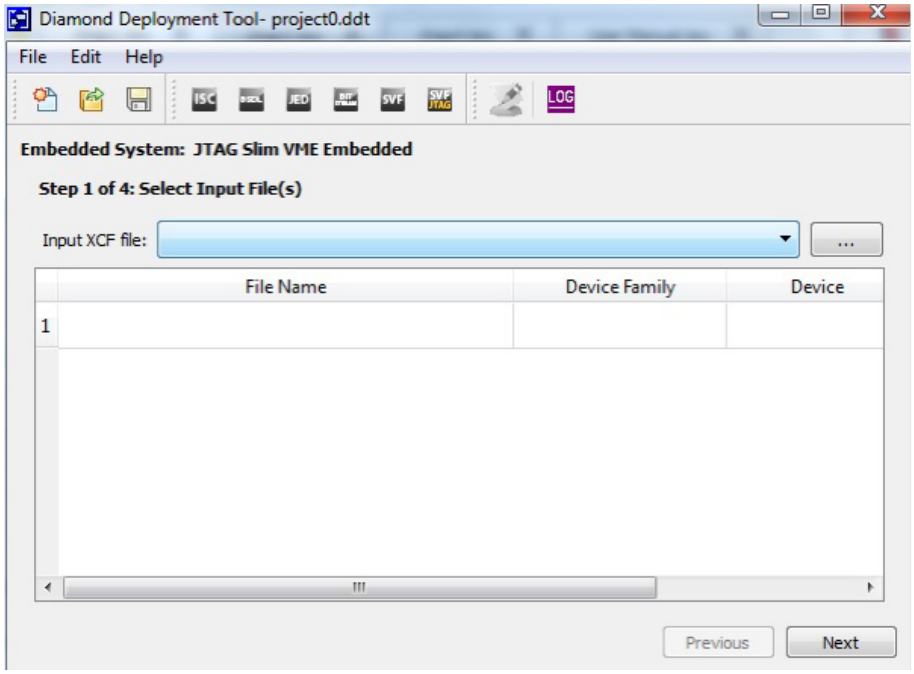
9.5.4 Deployment Tool usage

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

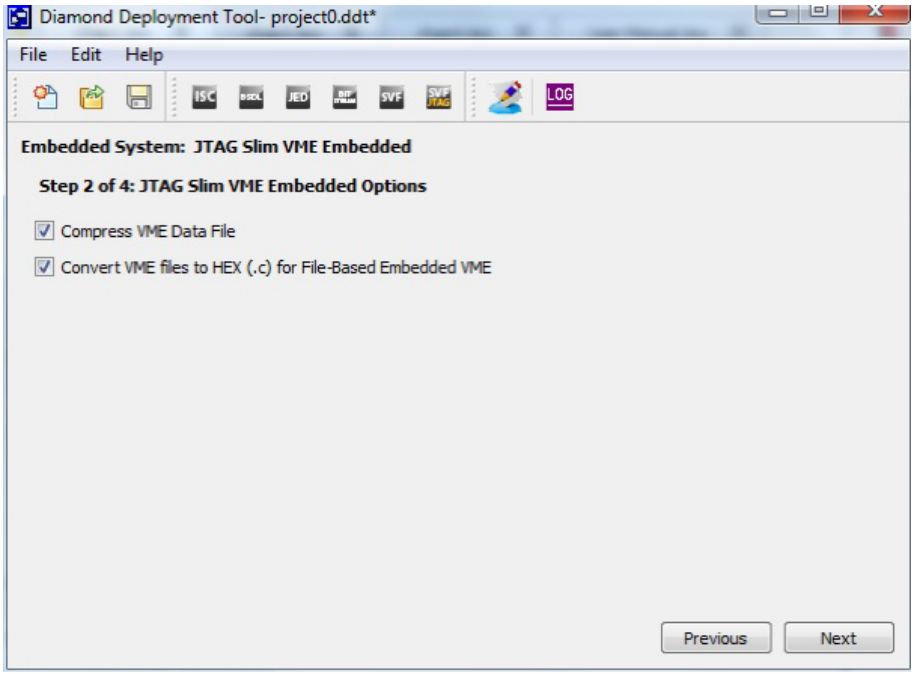
1. First, open the Deployment Tool and create a new project like in the following



2. Locate the XCF file and click on «Next»



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



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

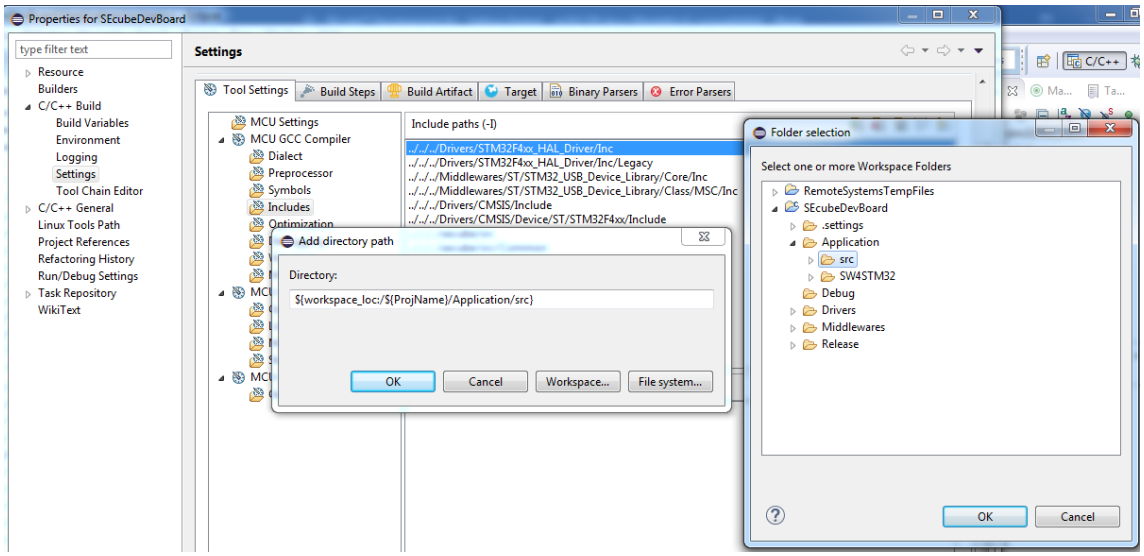
9.5.5 Putting all together

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. Set up a project as described in 9.3.9

2. You might want to import also your custom C/C++ files for interacting with your design present on the FPGA. Add them following the same procedure

3. In the project settings, if not automatically supported, add the imported files to the build path of the project



4. 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

5. 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

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

7. If a software library has been written by you for interacting with the FPGA, include it after "FPGA.h"

8. Also in "main.c", add a call to `B5_FPGA_Programming()` function

9. Add all the other calls that you may have written for the CPU-FPGA interaction

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

11. Connect the DevKit as described in previous section

12. Run the project following the IDE-specific steps that are described in Section 9.2.1

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.

APPENDIX A - SE*cube*™ Data Sheet

SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

General Description

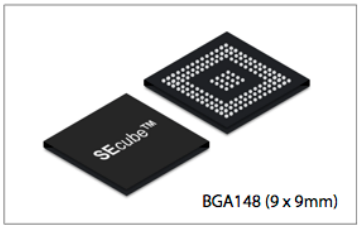
The SE*cube*™ (Secure Environment cube) is a powerful chip which integrates three key security elements in a single package. A fast floating-point Cortex-M4 CPU, a high-performance FPGA and an EAL5+ certified Security Controller (Smart Card).

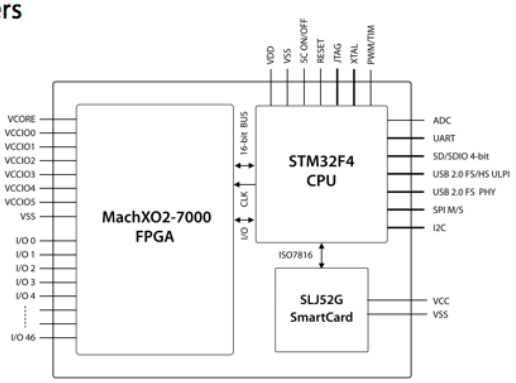
The result of this innovative combination gives an extremely versatile secure environment in a single SoC, in which developers can rapidly implement complex applications and appliances.

The SE*cube*™ chip has multiple embedded communication interfaces. In addition, the internal FPGA provides up to 47 fast I/O (100 MHz) for custom high-speed interface implementations.

This allows fast integration of the SE*cube*™ into any hardware design, while drastically reducing the final BOM.

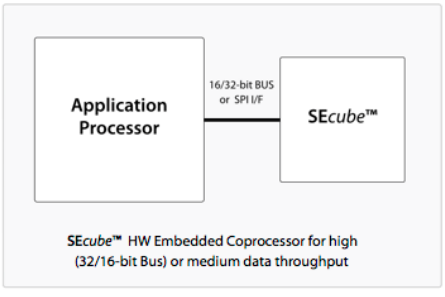
The SE*cube*™ is the ultimate solution for high-end design, delivering integration of a flexible, configurable and certified secure element.



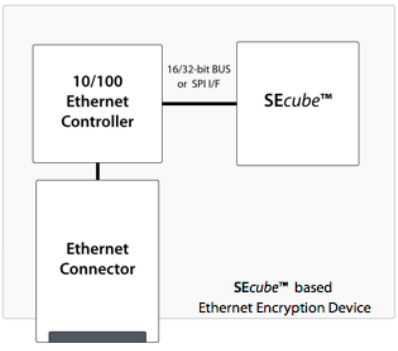


SE*cube*™ Block Diagram

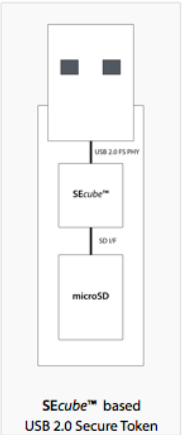
TYPICAL APPLICATION DIAGRAMS



SE*cube*™ HW Embedded Coprocessor for high (32/16-bit Bus) or medium data throughput



SE*cube*™ based Ethernet Encryption Device



SE*cube*™ based USB 2.0 Secure Token

SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

Main Features

Three powerful elements in one chip

■Embedded STM32F4 CPU

- Core: ARM® 32-bit Cortex®-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 180 MHz, MPU, 225 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories:
  - 2 MB of Flash memory organised into two banks allowing read-while-write
  - 256+4 KB of SRAM including 64-KB of CCM (core coupled memory) data RAM
- Clock, reset and supply management:
  - 1.7 V to 3.6 V application supply and I/Os POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC (1% accuracy)
  - Internal 32 kHz RC with calibration
- Low power
  - Sleep, Stop and Standby modes
  - VBAT supply for RTC, 20×32 bit backup registers + optional 4 KB backup SRAM
- JTAG interface
- 1×12-bit, 2.4 MSPS ADC, 7.2 MSPS in triple interleaved mode
- Up to 17 timers: up to twelve 16-bit and two 32- bit timers up to 180 MHz, 1 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
- 1xSPI (45 Mbits/s) Master/Slave configurable
- 1USART (11.25 Mbit/s, CTS,RTS RS232)
- 1 × I2C interface (SMBus/PMBus)
- 1 x SD/SDIO interface up to 48MHz (SD v4.2, SDIO v2.0), 1bit-4bit modes supported
- True random number generator
- CRC calculation unit
- RTC: sub-second accuracy, hardware calendar
- 96-bit unique ID
- USB Connectivity:
  - USB 2.0 full-speed device/host/OTG controller with on-chip PHY
  - USB 2.0 high-speed/full-speed device/host/OTG controller with dedicated DMA, on-chip full-speed PHY and ULPI
- Connections to SmartCard:
  - ISO7816 interface with Clock
  - 1 x GPIO to control external power supply
- Connections to FPGA:
  - 16-bit data, 6-bit address, 100MHz bus SRAM/PRAM mode, 2 x chip selects
  - Master Oscillator pin, up to 90 MHz
  - 5xGPIOs connected to the FGPA JTAG interface for bit-bending programming operations
  - 2xGPIOs for status/polling/interrupt signalling

■ Embedded MachXO2-7000 FPGA

- 6864 LUTs and 47 I/Os
- Ultra Low Power Device (65 nm process, 19 µW standby power, programmable low swing differential I/Os, Stand-by mode and other power saving options)
- Embedded and distributed memory
  - 240 Kbits SysMEM™ embedded blocks RAM
  - 54 Kbits distributed RAM
  - Dedicated FIFO control logic
- 256 Kbits On-Chip User Flash Memory
- Flexible I/O Buffers:
  - (LVCMOS 3.3/2.5/1.8/1.5/1.2, LVTTL, PCI, LVDS, Bus-LVDS, MLVDS, RSDS, LVPECL, SSTL 25/18, HSTL 18,Schmitt trigger input up to 0.5 V hysteresis, etc.)
  - On-chip differential terminations
- Wide Frequency range (10 MHz to 400 MHz)
- Non-Volatile infinitely reconfigurable
- In-field logic configuration while system operates

■ Embedded SLJ52G SECURITY CONTROLLER - SMART CARD

- JavaCard Platform, including ePassport and eSign applets
- ISO7816 Interface
- Supported standards: JC 3.0, GP 2.2, ICAO BAC, SAC, AA, BSI-TR03110 v1.11 EAC, ISO 18013 BAP, EAP config 1-4
- 128 KByte EEPROM
- DES, 3DES, AES up to 256-bit
- RSA up to 2048-bit, ECC up to 521-bit
- Certified Common Criteria CC EAL5+ high

SE*cube*™ Data Sheet Introduction

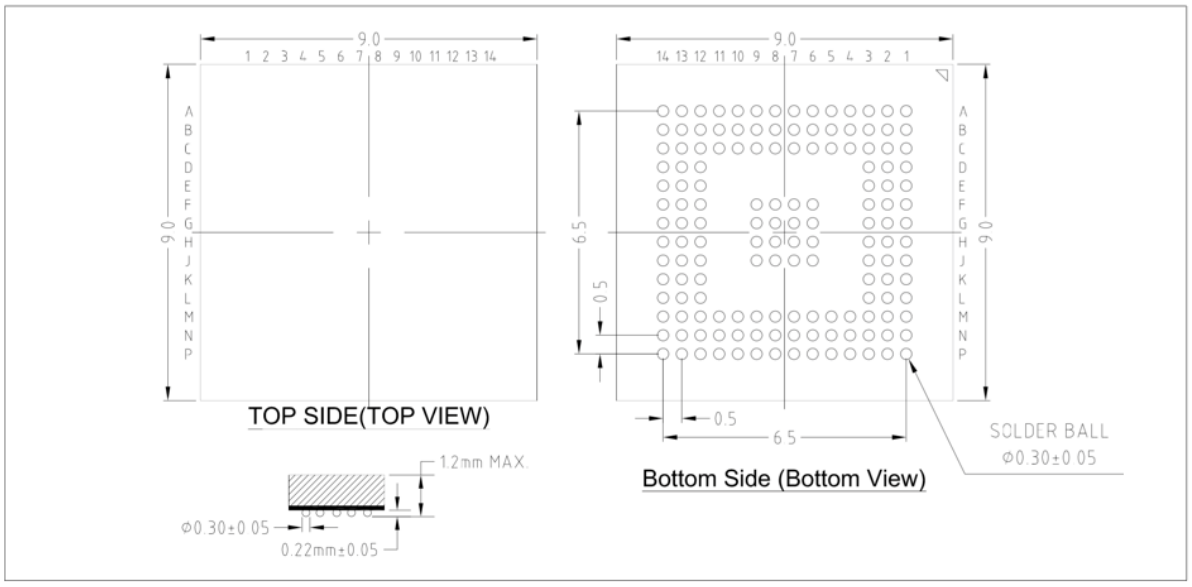
August 2015

Data Sheet DUI 15082DS

Pinout and Packaging

SE*cube*™ Pinout Table

A1	FPGA_IO_D12	B12	FPGA_IO_D4	E3	CPU_JTAG_TMS	H2	CPU_SDIO_D2	L1	FPGA_IO_CTRL1	N6	FPGA_VCORE
A2	FPGA_IO_CTRL4	B13	FPGA_IO_GP14	E12	CPU_USB_ULPI_D6	H3	CPU_SDIO_CLK	L2	CPU_SC_PWR	N7	CPU_SPI_SSN
A3	FPGA_IO_CTRL2	B14	FPGA_VCORE	E13	CPU_USB_ULPI_D5	H6	VSS	L3	CPU_UART_TX	N8	CPU_USB_ULPI_NXT
A4	FPGA_IO_D9	C1	FPGA_VCCIO0	E14	FPGA_IO_D0	H7	VSS	L12	CPU_USB_ULPI_CLK	N9	CPU_GP1
A5	FPGA_IO_D14	C2	CPU_VCAP2	F1	FPGA_IO_D15	H8	VSS	L13	FPGA_VCORE	N10	CPU_USB_ULPI_STP
A6	FPGA_IO_D7	C3	CPU_VDD	F2	CPU_JTAG_TDI	H9	VSS	L14	FPGA_VCORE	N11	CPU_I2C_SCL
A7	FPGA_IO_GP0	C4	CPU_VDD	F3	CPU_UART_CTS	H12	CPU_VDD	M1	FPGA_IO_CTRL3	N12	CPU_I2C_SDA
A8	FPGA_IO_D6	C5	CPU_UART_RX	F6	VSS	H13	FPGA_IO_GP13	M2	CPU_JTAG_TRST	N13	CPU_USB_ULPI_D0
A9	FPGA_IO_GP6	C6	FPGA_IO_GP3	F7	VSS	H14	FPGA_IO_D1	M3	VSS	N14	VSS
A10	FPGA_IO_GP5	C7	CPU_SDIO_D1	F8	VSS	J1	CPU_USB_ULPI_D7	M4	CPU_VDD	P1	FPGA_VCCIO0
A11	FPGA_IO_GP9	C8	CPU_SDIO_D0	F9	VSS	J2	CPU_VDD	M5	CPU_SPI_CLK	P2	VSS
A12	FPGA_IO_D5	C9	CPU_GP0	F12	CPU_VCAP1	J3	CPU_VDD	M6	CPU_XTAL_IN	P3	FPGA_VCCIO5
A13	FPGA_IO_GP15	C10	FPGA_VCCIO1	F13	CPU_USB_ULPI_D4	J6	VSS	M7	CPU_SPI_MOSI	P4	FPGA_IO_CTRL6
A14	FPGA_VCCIO1	C11	CPU_VDD	F14	FPGA_IO_GP11	J7	VSS	M8	CPU_RSTN	P5	CPU_USB_ULPI_DIR
B1	FPGA_VCCIO1	C12	CPU_VDD	G1	FPGA_VCCIO0	J8	VSS	M9	CPU_ADC	P6	FPGA_VCORE
B2	FPGA_IO_D10	C13	FPGA_VCORE	G2	CPU_JTAG_TCK	J9	VSS	M10	CPU_WKUP	P7	FPGA_VCCIO4
B3	FPGA_IO_CTRL0	C14	FPGA_VCCIO2	G3	CPU_SDIO_D3	J12	CPU_VDD	M11	CPU_VDD	P8	CPU_SPI_MISO
B4	FPGA_IO_D8	D1	FPGA_IO_CTRL13	G6	VSS	J13	FPGA_IO_D2	M12	VSS	P9	FPGA_IO_CTRL8
B5	FPGA_IO_D13	D2	FPGA_VCORE	G7	VSS	J14	FPGA_IO_D3	M13	VSS	P10	FPGA_IO_CTRL9
B6	FPGA_IO_D11	D3	FPGA_VCORE	G8	VSS	K1	FPGA_VCORE	M14	FPGA_VCCIO2	P11	FPGA_IO_CTRL10
B7	FPGA_IO_GP1	D12	CPU_USB_DM	G9	VSS	K2	FPGA_VCORE	N1	SC_VCC	P12	FPGA_IO_CTRL11
B8	FPGA_IO_GP2	D13	CPU_TIMER_PWM	G12	CPU_USB_DP	K3	CPU_JTAG_TDO	N2	FPGA_IO_CTRL5	P13	FPGA_IO_CTRL12
B9	FPGA_IO_GP4	D14	FPGA_IO_GP10	G13	CPU_USB_ULPI_D3	K12	CPU_USB_ULPI_D2	N3	VSS	P14	FPGA_VCCIO3
B10	FPGA_IO_GP8	E1	FPGA_IO_CTRL14	G14	FPGA_IO_GP12	K13	CPU_USB_ULPI_D1	N4	FPGA_IO_CTRL7		
B11	FPGA_IO_GP7	E2	CPU_UART_RTS	H1	CPU_SDIO_CMD	K14	FPGA_VCCIO2	N5	CPU_XTAL_OUT		



SE*cube*™ Packaging information

SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

Embedded Components Cross-Connections

Refer to the specific component's data sheets for further details.
Power supply signals are directly connected to the relating components.

SE*cube*™ Internal Connections

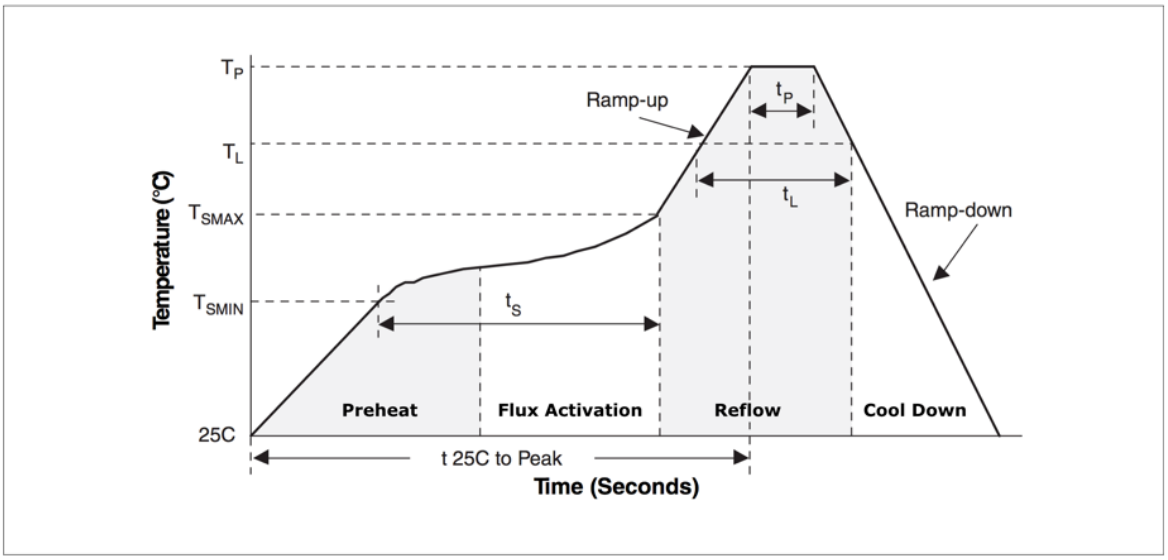
SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

Reflow Profiles

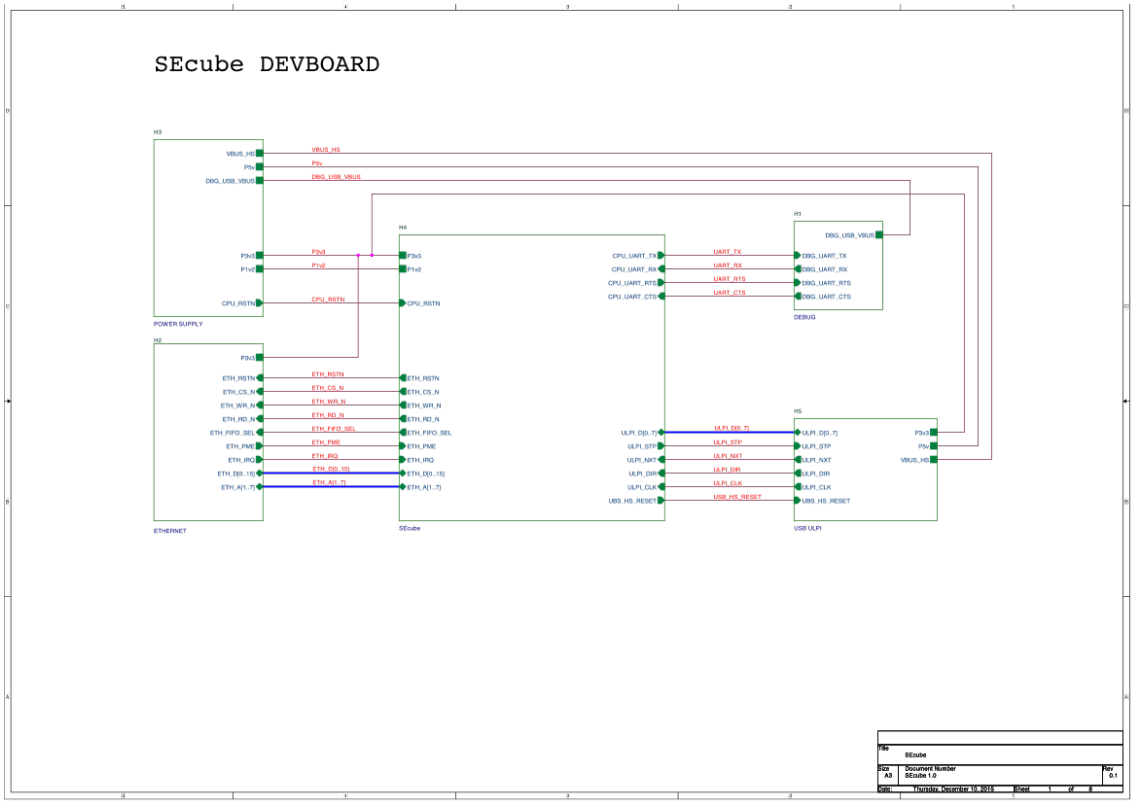
SE*cube*™ Reflow Table

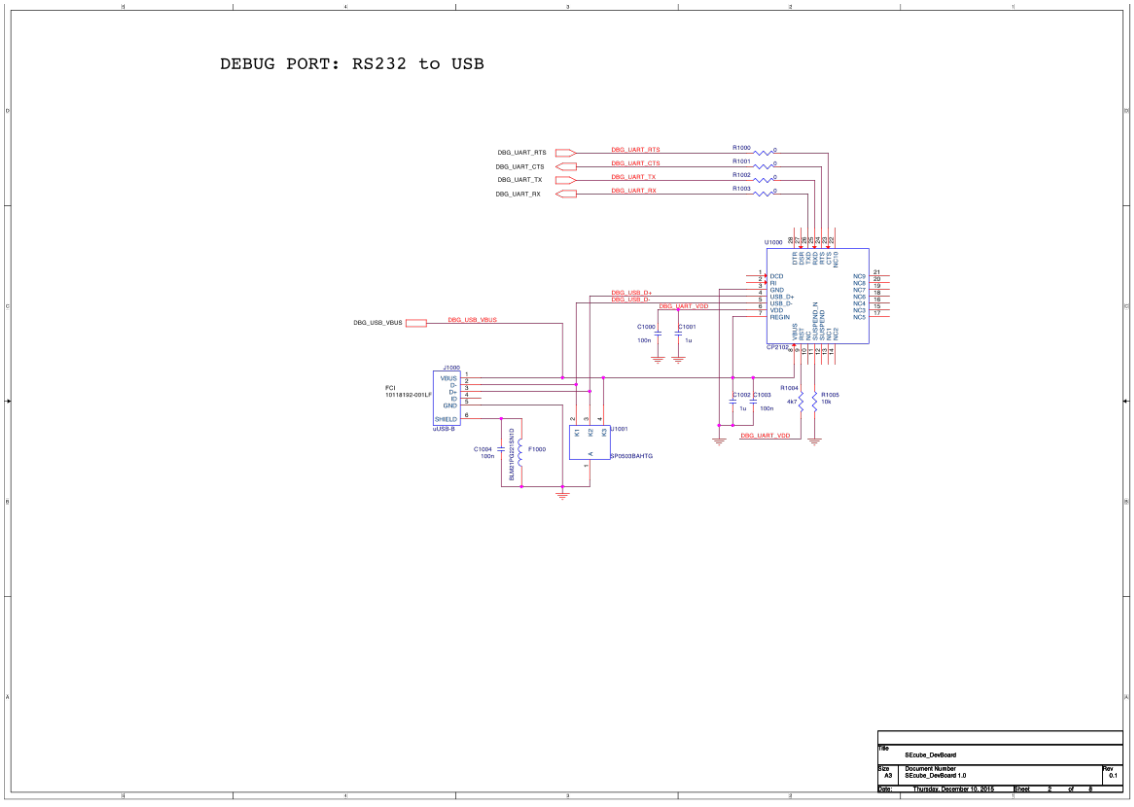
Parameter	Description	Pb-Free and Halogen-Free Packages
Ramp-Up	Average Ramp-Up Rate (T_{SMAX} to T_P)	3 °C/second max.
T_{SMIN}	Preheat Peak Min. Temperature	150 °C
T_{SMAX}	Preheat Peak Max. Temperature	200 °C
t_S	Time between T_{SMIN} and T_{SMAX}	60 seconds–120 seconds
T_L	Solder Melting Point	217 °C
t_L	Time Maintained above T_L	60 seconds–150 seconds
t_P	Time within 5 °C of Peak Temperature	30 seconds
Ramp-Down	Ramp-Down Rate	6 °C/second max.
t 25 °C to T_P	Time from 25 °C to Peak Temperature	8 minutes max.

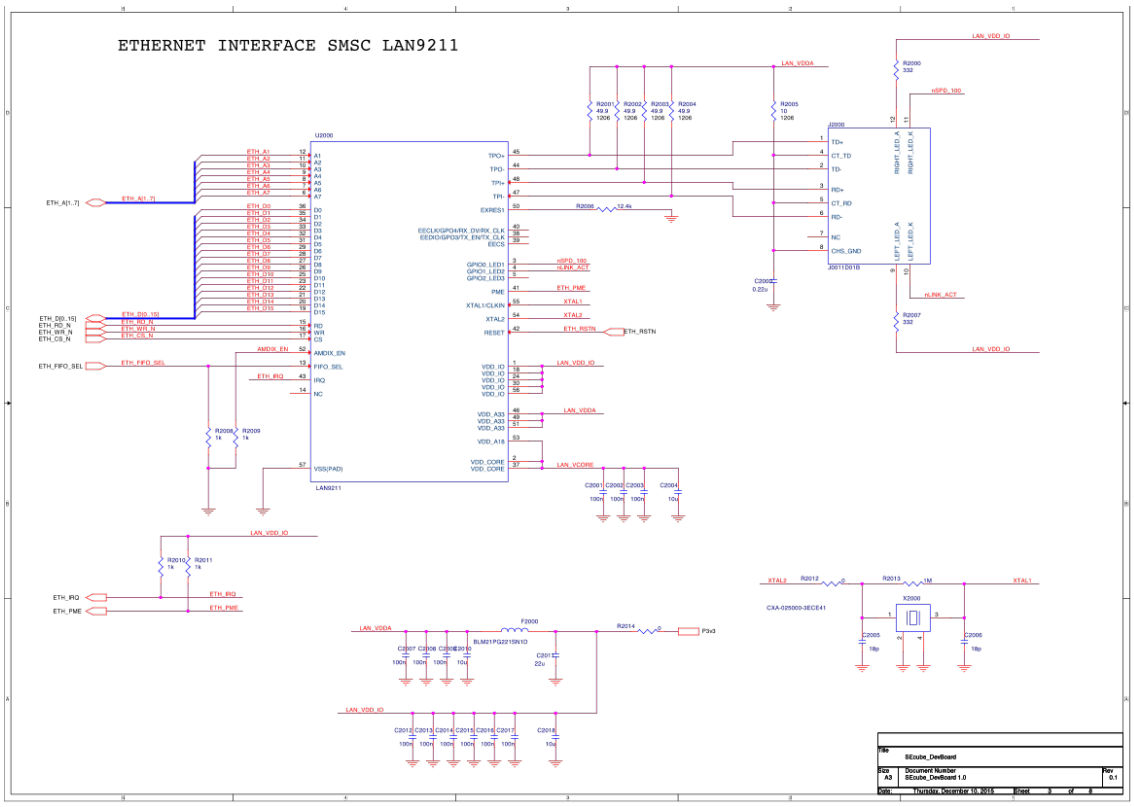


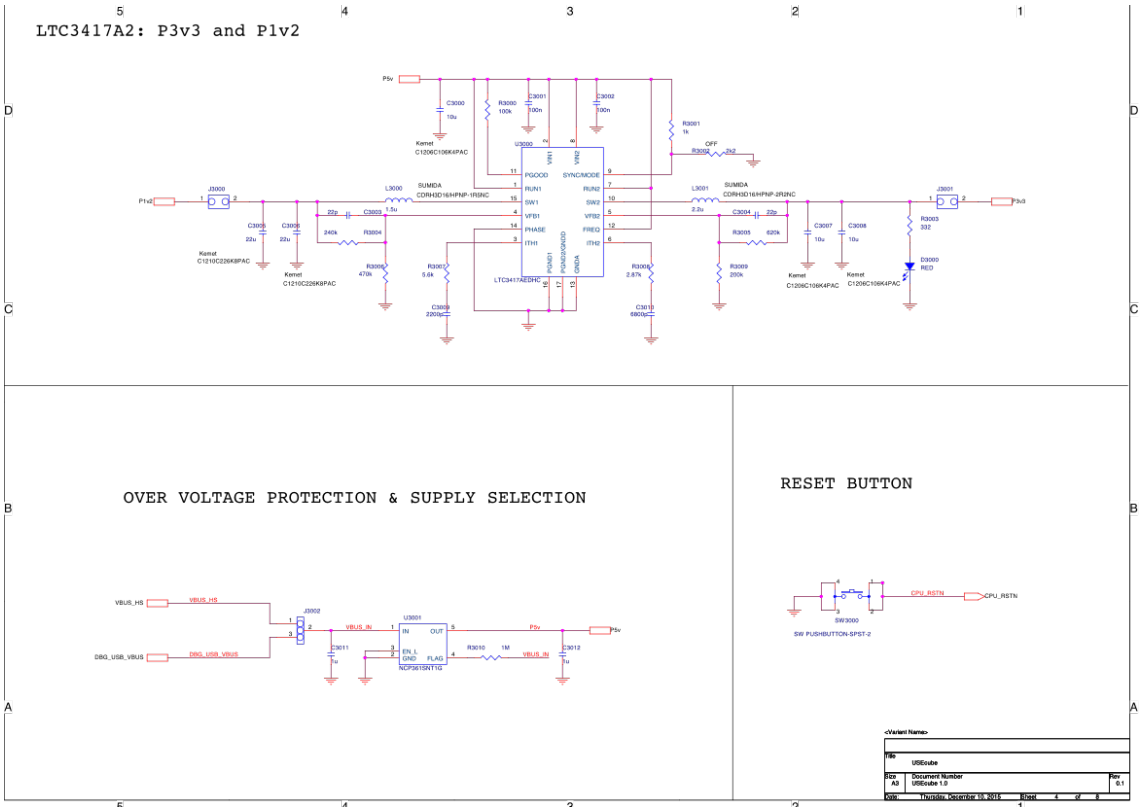
SE*cube*™ Termal Reflow Profile

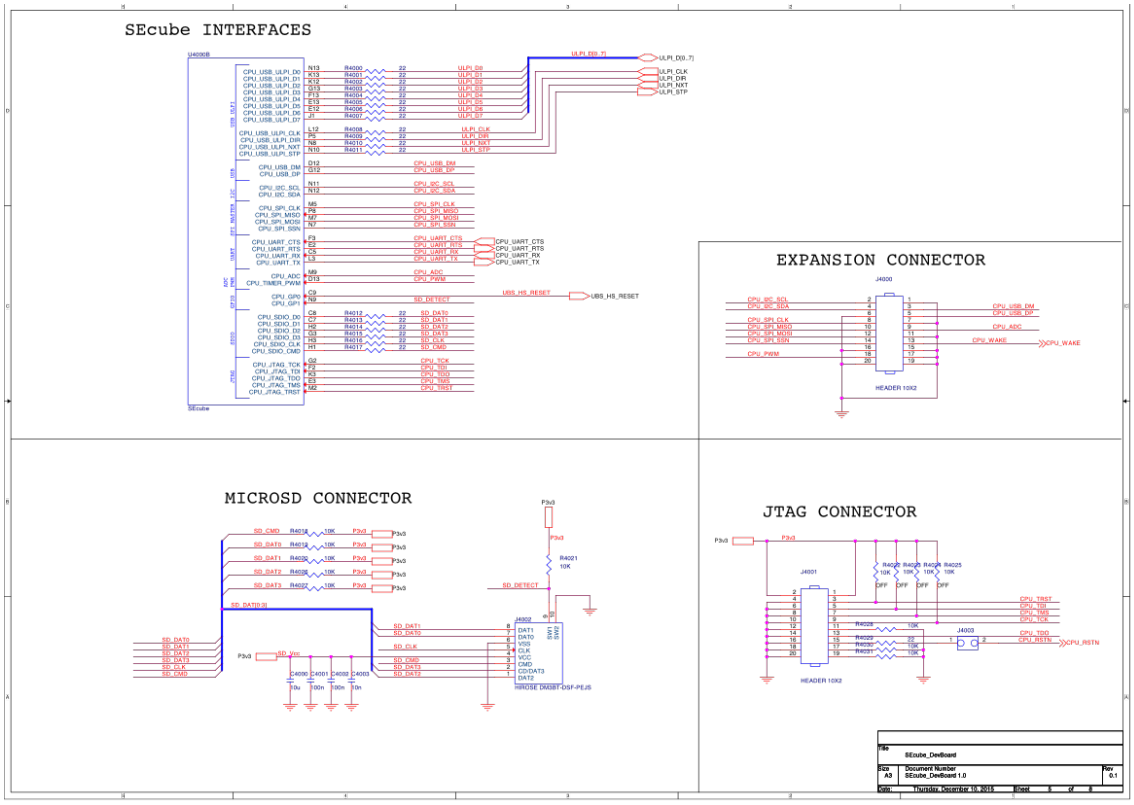
APPENDIX B - SE*cube*™ DevKit Schematics

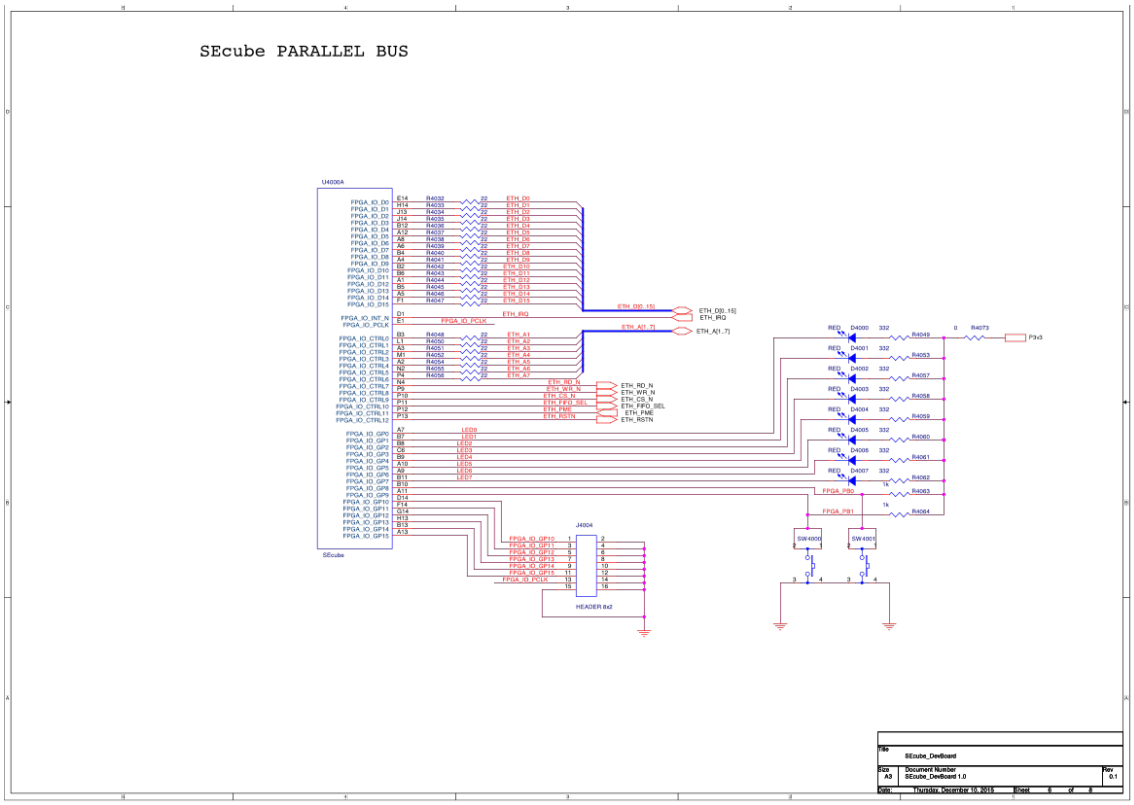


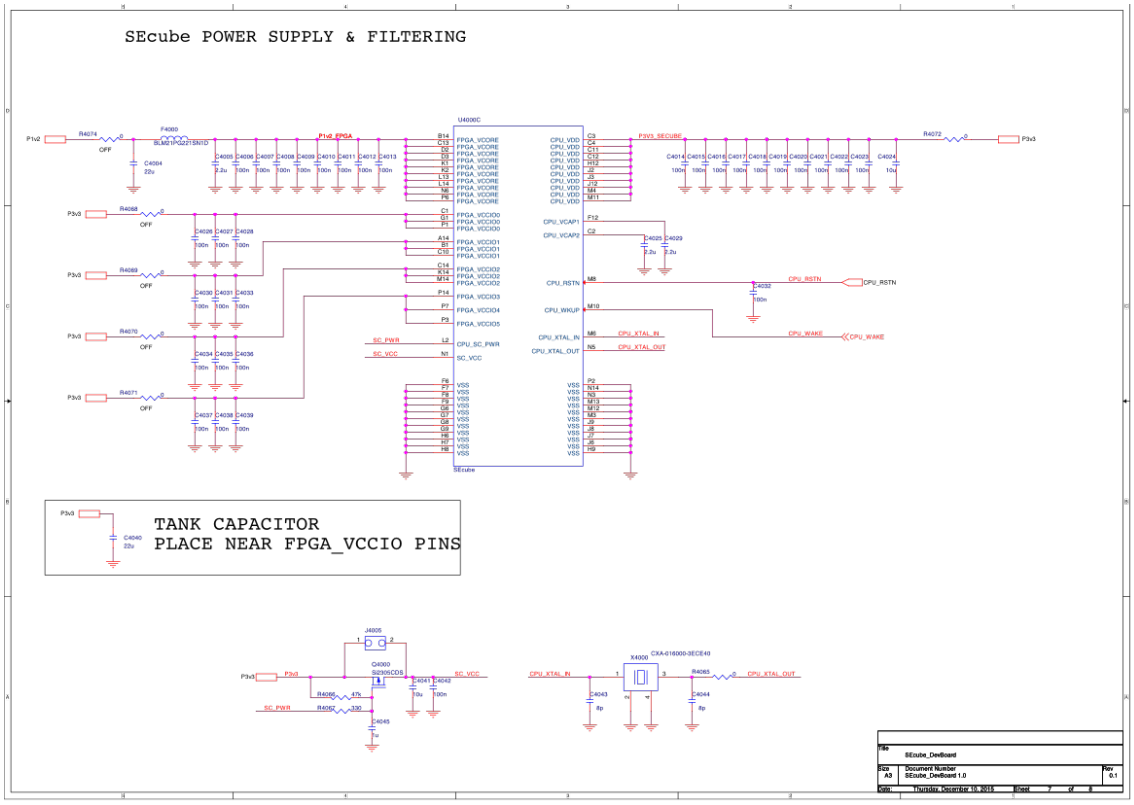


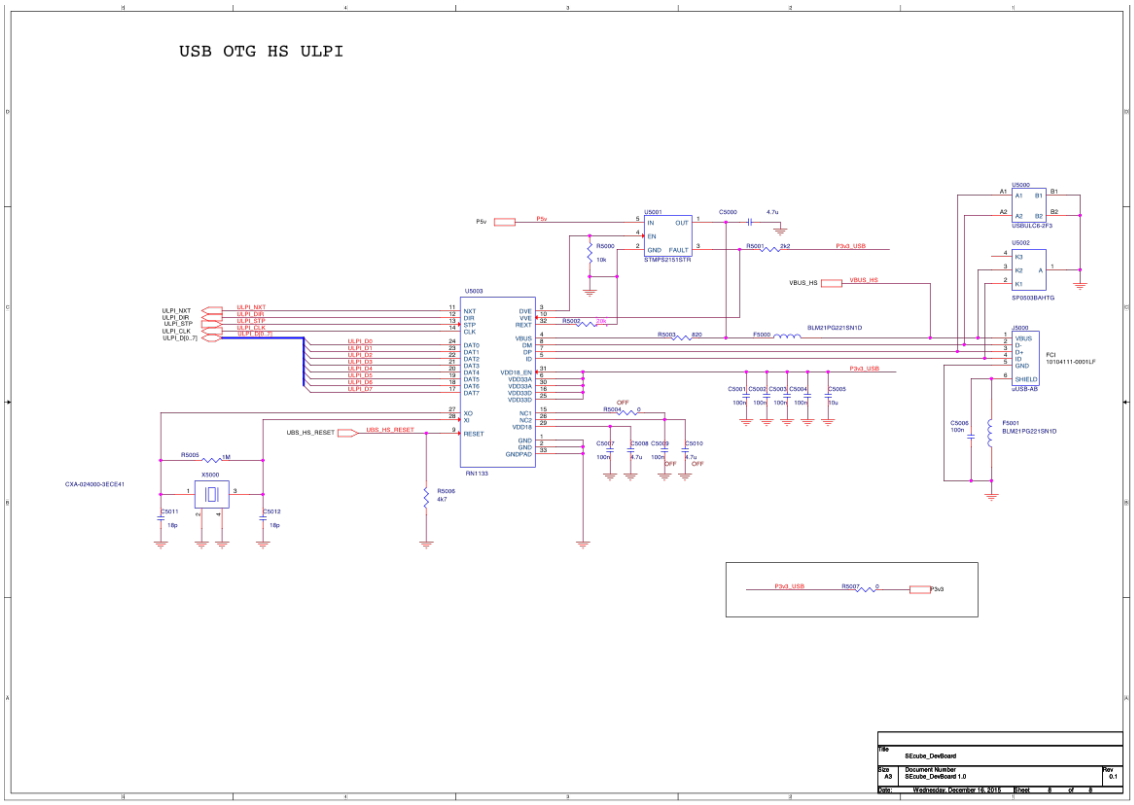

ETHERNET INTERFACE SMSC LAN9211
LAN9211
SEcube_DevBoard
SEcube_DevBoard 1.0


LTC3417A2: P3v3 and P1v2
OVER VOLTAGE PROTECTION & SUPPLY SELECTION
RESET BUTTON
USEcube
USEcube 1.0


SEcube INTERFACES
EXPANSION CONNECTOR
MICROSD CONNECTOR
JTAG CONNECTOR


SEcube PARALLEL BUS


SEcube POWER SUPPLY & FILTERING
TANK CAPACITOR
PLACE NEAR FPGA_VCCIO PINS


USB OTG HS ULPI