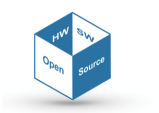
SEcube™
Open Security Platform

Introduction

Release: June 2020


HW
SW
Open
Source

Proprietary Notice

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

Authors

Matteo FORNERO *(Researcher, CINI Cybersecurity National Lab)* fornero.matteo@gmail.com
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

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, security is one of the biggest necessities of the world, because our lives resolve around computers present everywhere. 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 the most part of enterprises has their infrastructure fully computerized. All the transactions and important information are in personal computers or servers placed and interconnected everywhere around the world. Such a scenario provides endless opportunities to cyber-attackers, which have been dramatically increasing in number every day.
In order to protect the entire information chain, today more than ever, there is a large 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 too many times are underestimate or ignored when a security solution is deployed.
Although software gives easy and flexible protection, hardware is much faster and provides better immunity from contamination, malicious code infections or vulnerability.
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 the complexity of possible cyber-attacks grows up drastically and there are several hardware techniques to enforce the system, like 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 developers. Finally, there are single

[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

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 the 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 twenty-five 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 of the single cryptographic operation.
For example, the PKCS#11 interface[7] available since 1994 and still largely used in the public-private key based 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, each based on less than 10 APIs, 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 centered on **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).

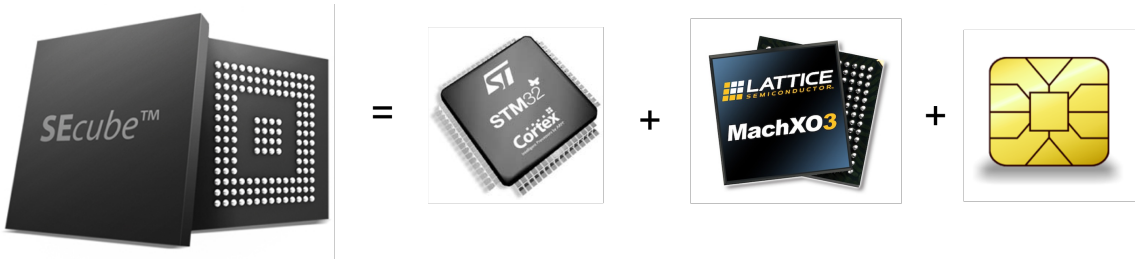


Figure 1: The 3 components of **SE*cube*™** .

[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

[9] www.filierasicura.it

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 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, 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 sub-modules (e.g., communication interface, file system, etc.) be easily identifiable.
All the software is released in source code under GPLv3 license[10].

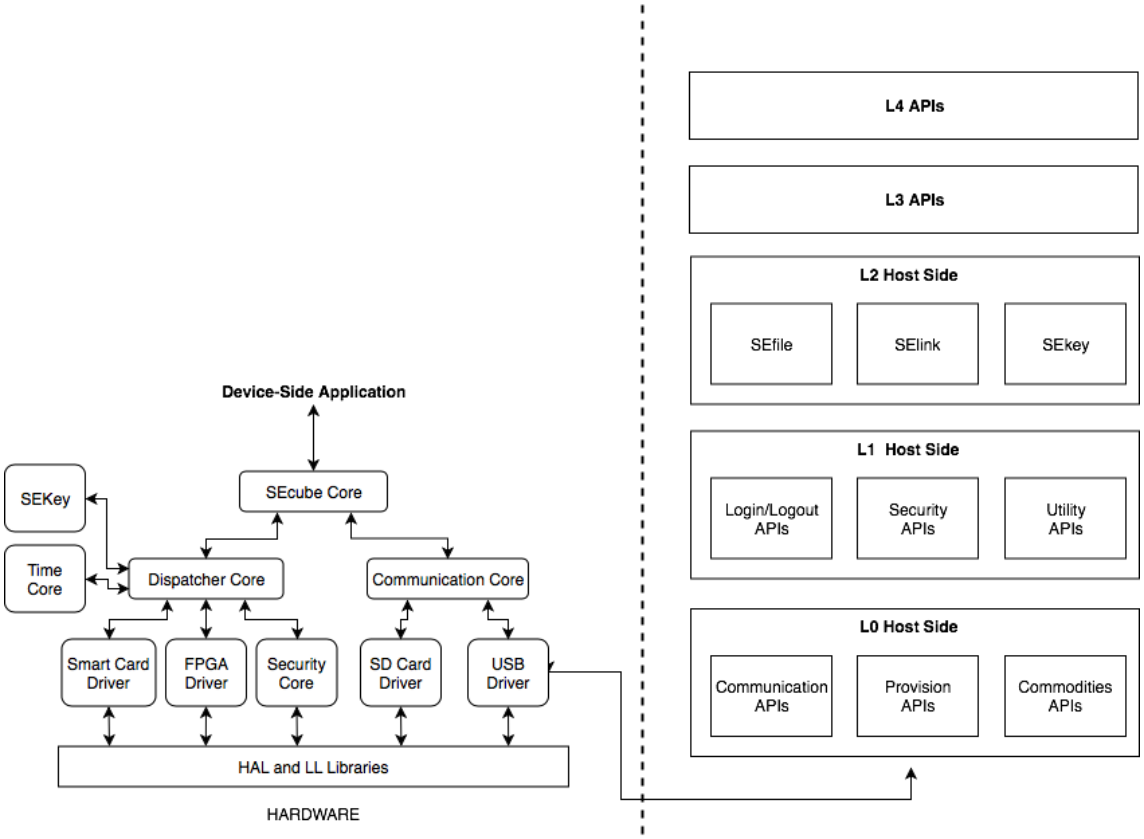


Figure 3: The **SE*cube*™** Software Architecture.

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.

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

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 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 using 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 just focused on the final security service.
Information can typically be in two major statuses: *at rest* and *in motion*. In both cases the most important and intuitive question to be considered when protecting data is the following: what recipient should information be protected for? 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 data bases, 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 group key and a set of security policies, both shared by ALL the members of the group.
The group key is used to protect both *static data* (data at rest) and *data transfers* (data in motion) among the members of the group. In particular, the key is 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

within the security infrastructure.
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. Of course, users are allowed to create sub-groups, within the security perimeter defined by the administrator.
In this way, at the application level it is not required to specify any kind of encryption method or cryptographic keys. As described in (cfr. Section 6) and (cfr. Section 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 detail is 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].

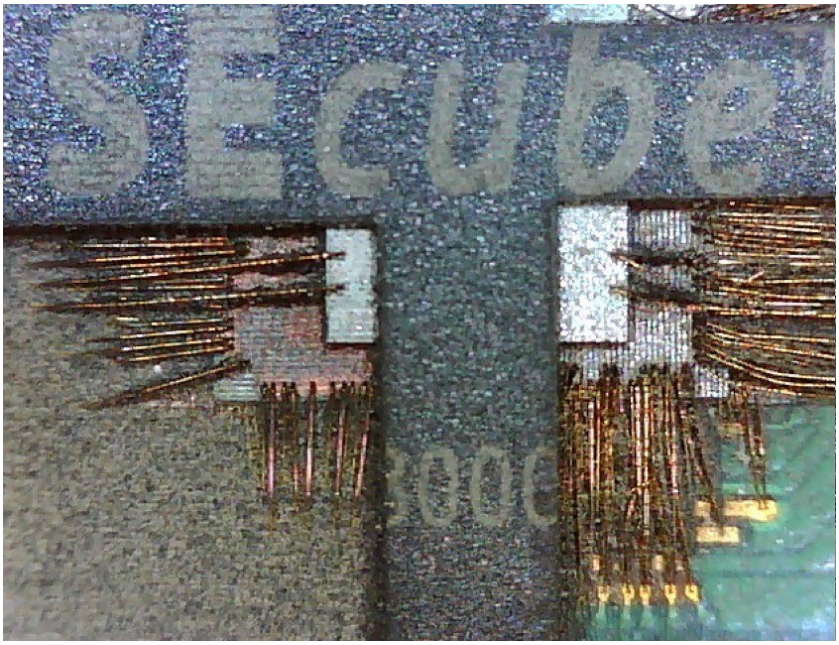


Figure 4: Detail of the die of the **SE*cube*™** Chip

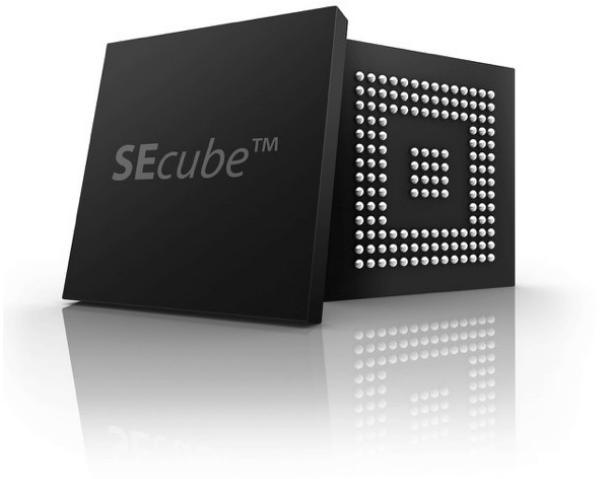


Figure 5: The **SE*cube*™** Chip

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 MiB[13] of Flash memory
- 256 KiB of SRAM
- 32-bit parallelism
- Operating frequency of 180 MHz
- Low power consumption.

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

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 Kib embedded block RAM
- 256 Kib 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 KiB = 210 Bytes
  - 1 MiB = 220 Bytes
  - 1 GiB = 230 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 Gib/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 Gib/s.
A CPU-FPGA clock line is provided to simplify the clock domains synchronization.
To limit the number of pins and the BGA package size, the FPGA JTAG is connected just to the embedded CPU, which manages both the debug and the programming operations. Therefore, the FPGA configuration can be implemented by means of customized, high-security techniques. For example, the programming bitstream can be encrypted and signed through dedicated algorithms. The CPU and/or the smartcard elements can then be used to decrypt and verify it before its injection into the FPGA.

3.1.3 The SmartCard

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

- ISO7816 interface
- JavaCard Platform, Global Platform 2.2
- 128 KiB Flash
- EC, ECDH up to 521 bit (HW accelerator)
- RSA up to 2 Kib (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] Infineon Chip Card & Security ICs Portfolio

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.

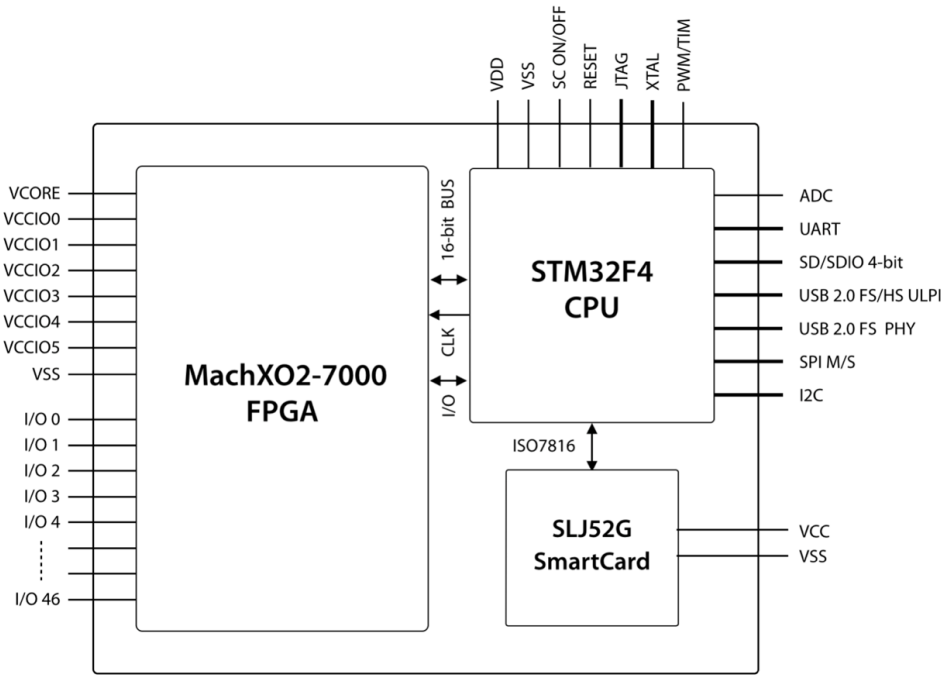


Figure 6: The **SE*cube*™** Hardware Architecture.

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.

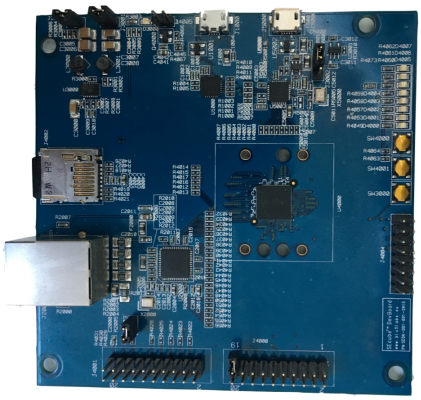


Figure 7: The **SE*cube*™** DevKit.

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 Led (SW4000, SW4001, SW3000, LED0, …)
- **SE*cube*™** embedded FPGA and CPU GPIOs (J4004, J4000)
- **SE*cube*™** embedded CPU JTAG (J4001).

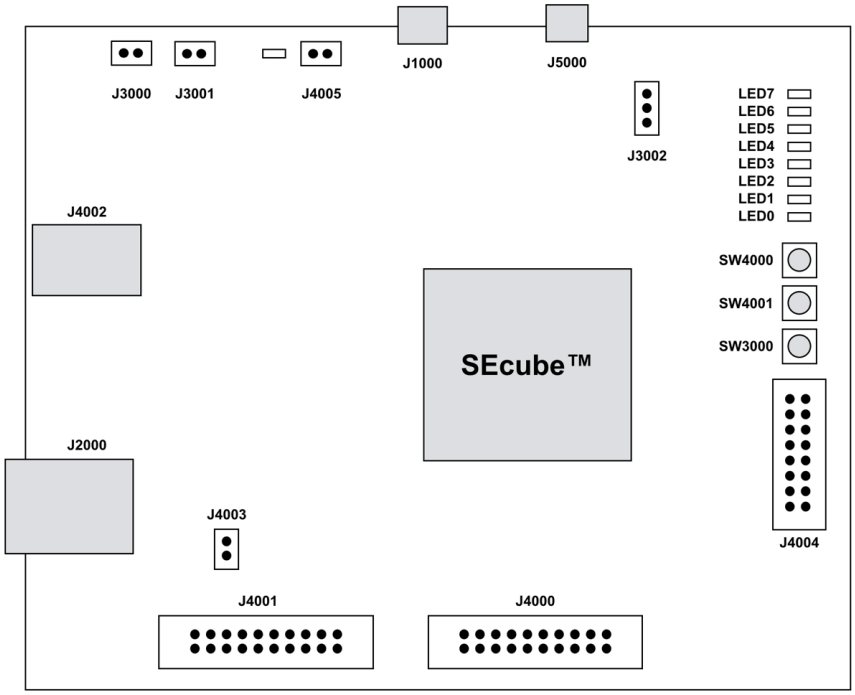


Figure 8: Floor planning view of the **SE*cube*™** DevKit.

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 34):

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

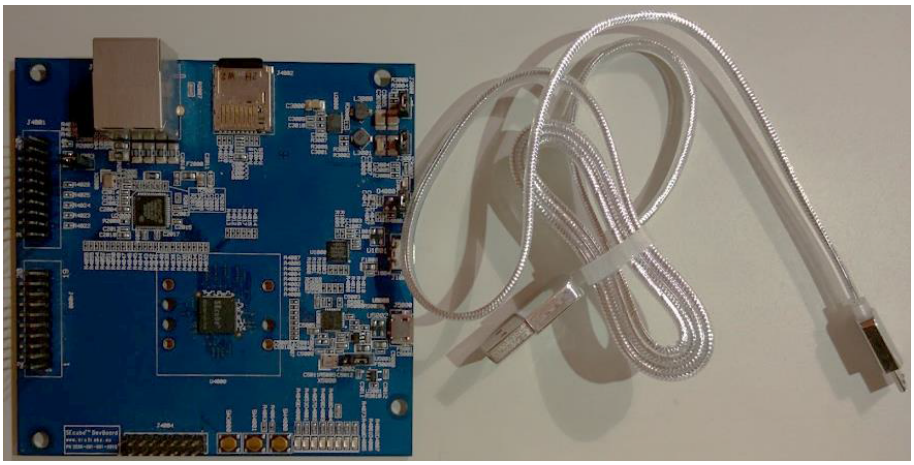


Figure 9: Components purchased with the **SE*cube*™** DevKit.

3.3 The USE*cube*™ Stick

As shown in Figure 10, the **USE*cube*™** Stick is an USB form factor based on the **SE*cube*™** Chip, which proves to be an amazing way to deploy the SEcube functionalities through a USB 2.0 High-Speed interface.

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.

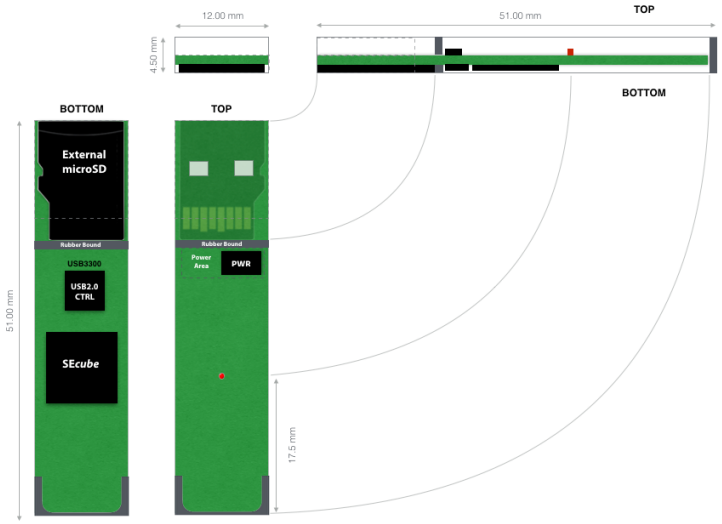


Figure 11: **USE*cube*™ Stick** internal structural details.

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 sub-modules (e.g., communication interface, file system, etc.) be easily identifiable. All the software is released in source code under GPLv3 license[18].

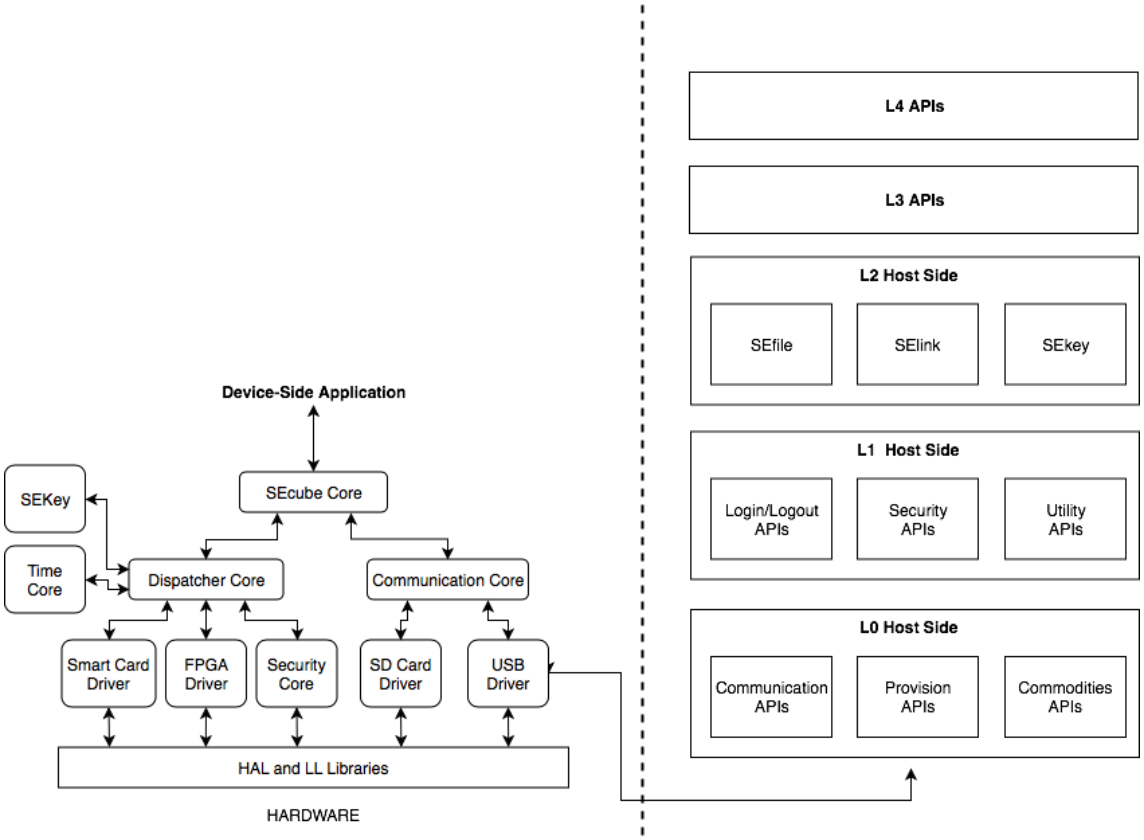


Figure 13: Software Architecture of **SE*cube*™** .

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

4.1 Device-Side Libraries

In this section are presented some of the principal Device-Side APIs, a more in depth explanation 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)
```

It is the device-side API to post back any data it receives.

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

It is the device-side API to initialize the device as well its serial number; the device remains locked unless the function answers correctly to a security challenge.

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, delete or update a key on the device.

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

Get a list of keys in the device.

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

Get a login challenge from the device.

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

Respond to challenge and complete the login.

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

Log our and release resources.

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)
```

This function is called when some request to the **SE*cube*™** arrive from the Host side.

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

This function is called when the **SE*cube*™** respond to the previous Host request.

4.1.4 Smart Card Driver

This set of functionalities shall provides all the APIs necessary for interacting whit the SmartCard. It is still to be implemented.

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

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.

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

Use a cryptographic context.

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

Set device time for key validity.

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

Get a list of available algorithms implemented on the device.

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 SEkey

This library contains the implementation of the APIs required to interact with **SE*key*™** for managing keys and groups. These APIs are still to be implemented.

4.1.9 Time Core

The set of APIs in this core implements the functionalities required for managing the time on the **SE*cube*™** device, such as reading the current time, using timers, etc.

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 uint8_t* serialno)
```

It is used to initialize the device as well as its serial number; the device remains locked unless it has been initialized. This service must be called before using the **SE*cube*™** device to set up its environment properly. This service can be called just once, as the device cannot be initialized more than once.

Communications APIs

```
void L0Open()
```

It opens the communication with the **SE*cube*™** device.

```
void L0Close()
```

It closes the communication with the **SE*cube*™** device.

```
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 send a packet of data containing a command and the relative parameters and then reads back the response written by the **SE*cube*™** in a shared data buffer.

```
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 to the list.

4.2.2 L1 Libraries

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

- login/logout
- security
  - key management
  - cryptography

In addition, L1 allows developers to manage several **SE*cube*™** devices concurrently, providing dedicated operation control flows (one command/response session per communication channel), which allow encoding and decoding commands for the individual **SE*cube*™** target.

Login/Logout APIs

```
void L1Login(const uint8_t* pin, uint16_t access, bool force)
```

It is used to initialize the procedures needed to login to the device. This service is used to let the user, or the admin, login on the **SE*cube*™** device. Before issuing any command to the **SE*cube*™** , apart from the Echo command, login is required. Please note that default user and admin PIN are set to all zeroes.

```
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()
```

It is used to initialize the procedures needed to logout from the device.

```
void L1LogoutForced()
```

It allows to force the logout from the device.

Security APIs

```
void L1CryptoSetTime(uint32_t devTime)
```

It is used to set expiration date for a cryptographic context.
WARNING: This function must be invoked right after any login operation, otherwise all of the subsequent operations cyphering will fail.

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

It is used to initialize a cryptographic context.
WARNING: data length for input and output must be a multiple of the crypto block size. The length of the input buffer is computed with the macro ENC_SIZE(buffer_len). The output buffer is allocated with that length but the actual length is an output of the function.

```
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)
```

It is used to update a cryptographic context.
WARNING: data length for input and output must be a multiple of the crypto block size. The length of the input buffer is computed with the macro ENC_SIZE(buffer_len). The output buffer is allocated with that length but the actual length is an output of the function.

```
void L1Encrypt(size_t dataInLen, uint8_t* dataIn, size_t*
   dataOutLen, uint8_t* dataOut, uint16_t algorithm, uint16_t
   mode, uint32_t keyId)
```

It is used to encrypt a buffer of data given the algorithm, the encryption mode, and the buffer size; data encrypted are then retrieved and returned as outcome.

```
void L1Decrypt(size_t dataInLen, uint8_t* dataIn, size_t*
   dataOutLen, uint8_t* dataOut, uint16_t algorithm, uint16_t
   mode, uint32_t keyId)
```

It is used to decrypt a buffer of data given the algorithm, the decryption mode, and the buffer size; data encrypted are then retrieved and returned as outcome.

```
void L1Digest(size_t dataInLen, uint8_t* dataIn, size_t*
   dataOutLen, uint8_t* dataOut, uint16_t algorithm)
```

It is used to compute the digest of a data buffer given the algorithm, and the buffer size; the digest is used to sign the data buffer.

```
void L1GetAlgorithms(uint16_t maxAlgorithms, uint16_t skip,
   se3Algo* algorithmsArray, uint16_t* count)
```

It is used to retrieve the list of the security algorithms supported by the device.

```
void L1SetAdminPIN(uint8_t* pin)
```

It is used to change the current admin pin.

```
void L1SetUserPIN(uint8_t* pin)
```

It is used to change the current user pin.

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

It is used to update a security key inserted into the device and its cryptographic context, in terms of adding, editing or removing it.

```
void L1KeyList(uint16_t maxKeys, uint16_t skip, se3Key* keyArray
   , uint16_t* count)
```

It is used to retrieve the list of the security keys inserted into the device and its cryptographic context.

```
bool L1FindKey(uint32_t keyId)
```

It is used to find a security key if it is inserted into the device and its cryptographic context.

4.2.3 L2 Libraries

This level relies on L1 APIs 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 in details all the low-level hardware and security mechanisms. The provided APIs include a Key Management System (**SE*key*™**), a data-at-rest protection facility (**SE*file*™**), and a data-at-motion protection facility (**SE*link*™**), briefly outlined in the sequel. Conceptually, APIs belonging to the L2 abstraction layer expose functionalities needed for any user who wants to perform, by moving inside a *secure environment*, basic operations on regular files and network packages. The concept of a *secure environment* reflects the need of providing a simple mechanism to allow the user to customize the parameters of the secure session, by configuring the keys to be used, and the algorithms to be enforced through all the valid life of the session itself.

4.3 SE*file*™

Any OS provides an abstraction layer in its kernel space, used to separate file system generic operations from their 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 mounted locally. Data protection is provided at this level of abstraction, by means of a dedicated security engine, hereinafter referred to as *secure layer*. However, malicious user, or software, still may exploit existing flaws in the application accessing to the secure layer or even in the secure layer itself to steal and compromise private information and confidential data. A countermeasure to protect effectively 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*™** is a file system that exploits the hardware key management exposed from L1 libraries (see 4.2.2) and other functionalities from the **SE*cube*™** device. It has been developed having in mind the needs to ensure both simplicity of usage and security for data at rest: it allows secure storage, retrieve and usage of information that could not be trusted if stored elsewhere, e.g., any personal computer, or cloud service provider. **SE*file*™** is deeply analyzed in Chapter 6.

4.4 SE*link*™

SE*link*™ is a software application that uses the **SE*cube*™** open platform to secure the network traffic. It can encrypt network streams originating from any application, regardless of the application-level protocol. **SE*link*™** is a reference implementation of an application on top of the L1 APIs4.2.2 for **SE*cube*™** . By using **SE*link*™** it is possible to add a secure network layer to any software, without modifying its code base. In other words, the user can employ any of his/her favorite network-enabled software (e.g., web browser, remote desktop viewer) and entrust the customizable security features to the **SE*cube*™** platform, in a way that is completely transparent to the user, allowing

him to exploit the benefits of the security functionalities without having deep knowledge about security. The network software (i.e., browser, cloud service for data sharing, etc.) does not need to be aware of the presence of **SE*link*™** and will function as usual, because **SE*link*™** intercepts connections at a lower level. **SE*link*™** is deeply analyzed in Chapter 7.

4.5 SE*key*™

SE*key*™ is a library that is used to manage encryption keys stored on the **SE*cube*™** device. **SE*key*™** , therefore, is a simple Key Management System (KMS) that allows to handle the entire lifecycle of encryption key. The KMS offers a set of APIs that can be used to create and distribute encryption keys to several **SE*cube*™** devices, each one belonging to a user of the KMS itself.
The users of **SE*key*™** are organized in groups, so that access to sensitive information and to encryption keys can be granted with fine granularity, for example authorizing only a specific user or a specific group of users. **SE*key*™** is completely handled by a single Security Administrator, whose role is to manage the KMS generating new keys, adding and removing users, creating and modifying groups and so on.
SE*key*™ exploits other **SE*cube*™** libraries (i.e. L1 and **SE*file*™**) to keep the managed data as secure as possible, for example storing them into an encrypted SQL database. In conclusion, **SE*key*™** implements a simple Key Management System that allows to any user, even without a solid cybersecurity background, to handle the lifecycle of encryption key without the need of worrying about low level security details; this is done thanks to a broad range of simple APIs. **SE*key*™** is deeply analyzed in Chapter 8.

4.6 L3 Libraries

This level relies on L2 services to develop secure applications, such as:

- Secure Virtualized File System
- Secure Data Base
- Secure Real-time Messaging
- ...

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 configuration within the **SE*cube*™** treats the FPGA as an external memory device (PSRAM), leveraging the Flexible Memory Controller (FMC) available on the processor. The FMC is an interfacing peripheral of the microcontroller used to connect external memories such as NOR Flash, NAND Flash, SRAM, and PSRAM[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).

[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/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#page=1602&zoom=100,0,116

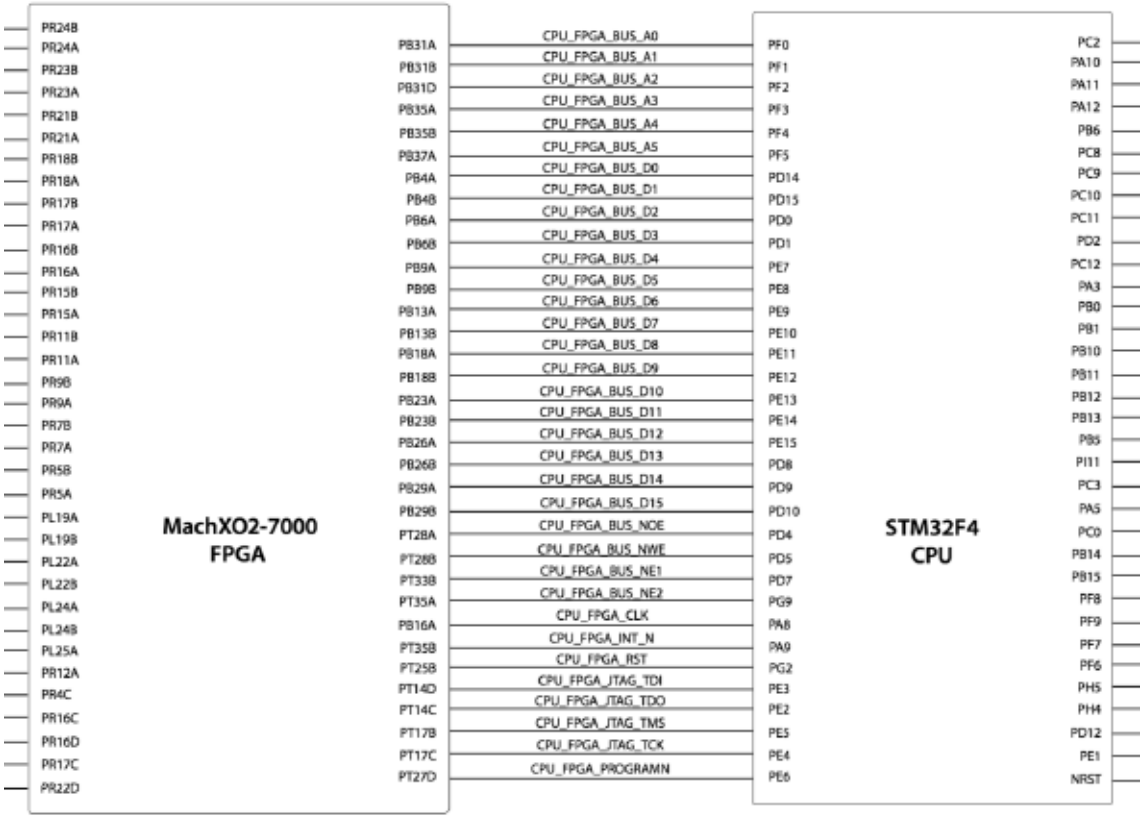


Figure 14: FPGA-CPU connections within **SE*cube*™**

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.

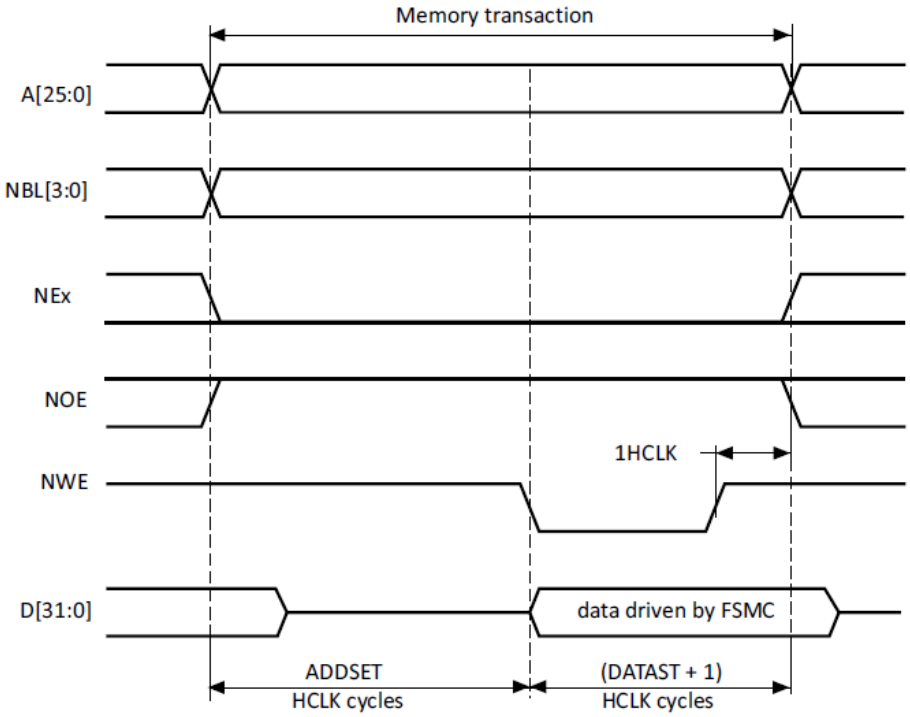


Figure 15: Access Mode A timing diagram for WRITE

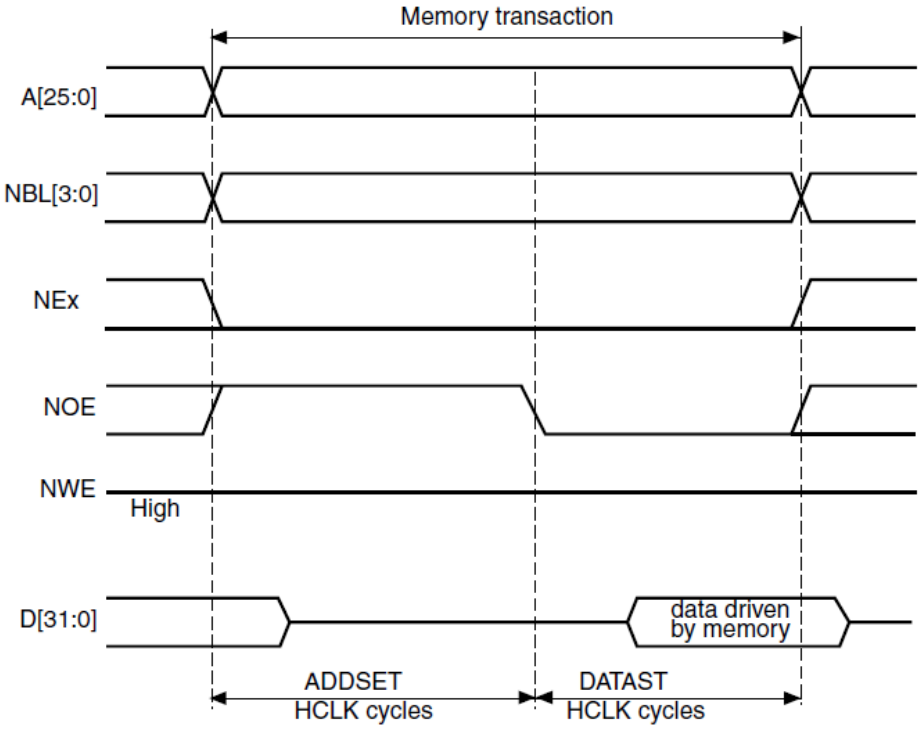


Figure 16: Access Mode A timing diagram for READ

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. This can be demanded to the GPIO initializer function `MX_GPIO_Init()`. Here, several calls to the HAL function `HAL_GPIO_Init()` consolidate in the FMC configuration registers the settings contained in a `GPIO_InitStruct` structure. Through this, it is possible 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 (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_FAST;
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_FAST;
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_FAST;
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_FAST;
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 SDK source files, under "/secube_sdk/libraries/Examples/TestFPGA", an example of bitstream file ("TEST_FPGA.h") can be found. The folder also contains 2 library files, "FPGA.h" and "FPGA.c", which must be included in a **SE*cube*™** device application which wants to exploit the FPGA. This little library contains the function `B5_FPGA_Programming()`, which must be called at the beginning of the application to program the FPGA. This 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 = GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 |
   GPIO_PIN_6;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);

/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = GPIO_PIN_2;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOE, &GPIO_InitStruct);
```

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 = GPIO_PIN_2;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
HAL_GPIO_Init(GPIOG, &GPIO_InitStruct);
```

and controlled in bit-banging when necessary through the dedicate HAL primitives:

```
HAL_GPIO_WritePin(GPIOG, GPIO_PIN_2, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOG, GPIO_PIN_2, 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. 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();
/* Enable clock for SYSCFG */
__HAL_RCC_SYSCFG_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/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#page=150&zoom=100,0,116

```
/* 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.

5.3.4 Interrupt signal

Pin PA9 of the interface has been allocated for hosting the interrupt signal coming from the FPGA (CPU_FPGA_INT_N). Such a signal can be very useful to handle a good CPU-FPGA cooperation, since time-consuming tasks can be processed by the FPGA in parallel without blocking the CPU activity, which is only interrupted at the end to retrieve the results.
The pin can be configured as an input pin attached to the External Interrupt/Event Controller (EXTI) and managed by the Nested Vectored Interrupt Controller (NVIC)[22]. The code extract below is an example of how the interrupt pin can be set to trigger the execution of the corresponding exception handler (`EXTI9_5_IRQHandler()`) at rising edges.

```
/* Set variables used */
EXTI_InitTypeDef EXTI_InitStruct;
NVIC_InitTypeDef NVIC_InitStruct;

/* Set pin PA9 as interrupt line from the FPGA */
GPIO_InitStruct.Pin = GPIO_PIN_9;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* Tell system that you will use PA9 for EXTI_Line9 */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource9);

/* PA9 is connected to EXTI_Line9 */
EXTI_InitStruct.EXTI_Line = EXTI_Line9;
/* Enable interrupt */
EXTI_InitStruct.EXTI_LineCmd = ENABLE;
```

[22] For additional details on EXTI and NVIC, please refer to STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs Reference Manual, Chapter 12: https://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf#page=371&zoom=100,0,116

```
/* Interrupt mode */
EXTI_InitStruct.EXTI_Mode = EXTI_Mode_Interrupt;
/* Triggers on rising edge */
EXTI_InitStruct.EXTI_Trigger = EXTI_Trigger_Rising;
/* Add to EXTI */
EXTI_Init(&EXTI_InitStruct);

/* Add IRQ vector to NVIC */
/* PA9 is connected to EXTI_Line9, which has EXTI9_5_IRQn vector
    */
NVIC_InitStruct.NVIC_IRQChannel = EXTI9_5_IRQn;
/* Set priority */
NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority = 0x00;
/* Set sub priority */
NVIC_InitStruct.NVIC_IRQChannelSubPriority = 0x01;
/* Enable interrupt */
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
/* Add to NVIC */
NVIC_Init(&NVIC_InitStruct);
```

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.
Thank to the usage of the FMC, the FPGA is seen by the processor as a normal peripheral, with a given set of shared locations starting from address `0x6000000` in the memory map. Thus, the communication with the architecture present onto the FPGA is basically composed of reads and writes, which in a high-level view form a master-slave protocol of queries and responses.
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., *enctypt, 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.

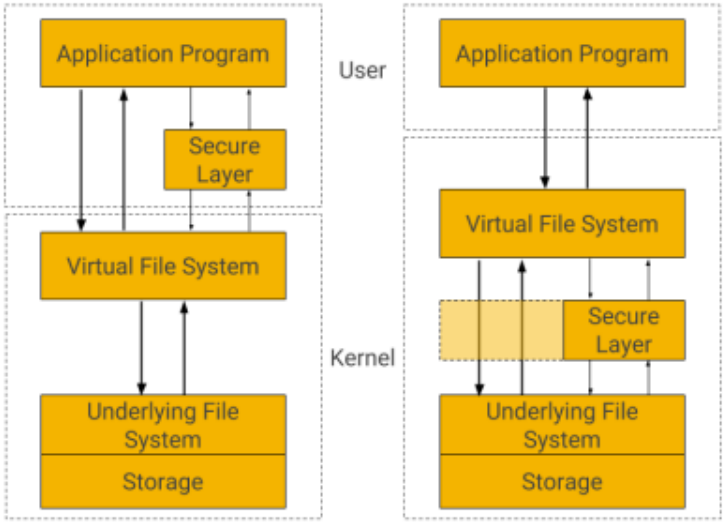


Figure 17: Secure Layer and Virtual File System: two different approaches.

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 could not be trusted

if stored elsewhere, e.g., any personal computer or cloud service provider.

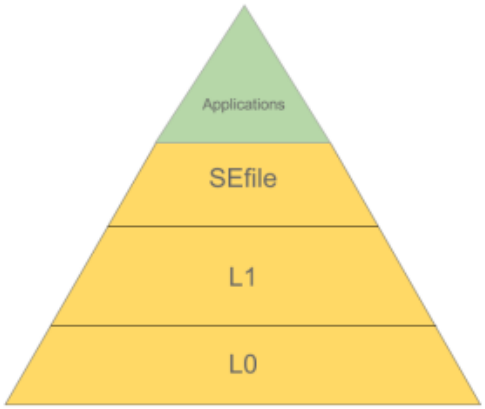


Figure 18: **SE*file*™** hierarchic organization.

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 authentication. The first sector is dedicated to the header, which provides information on 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. Is important to notice, however, 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.

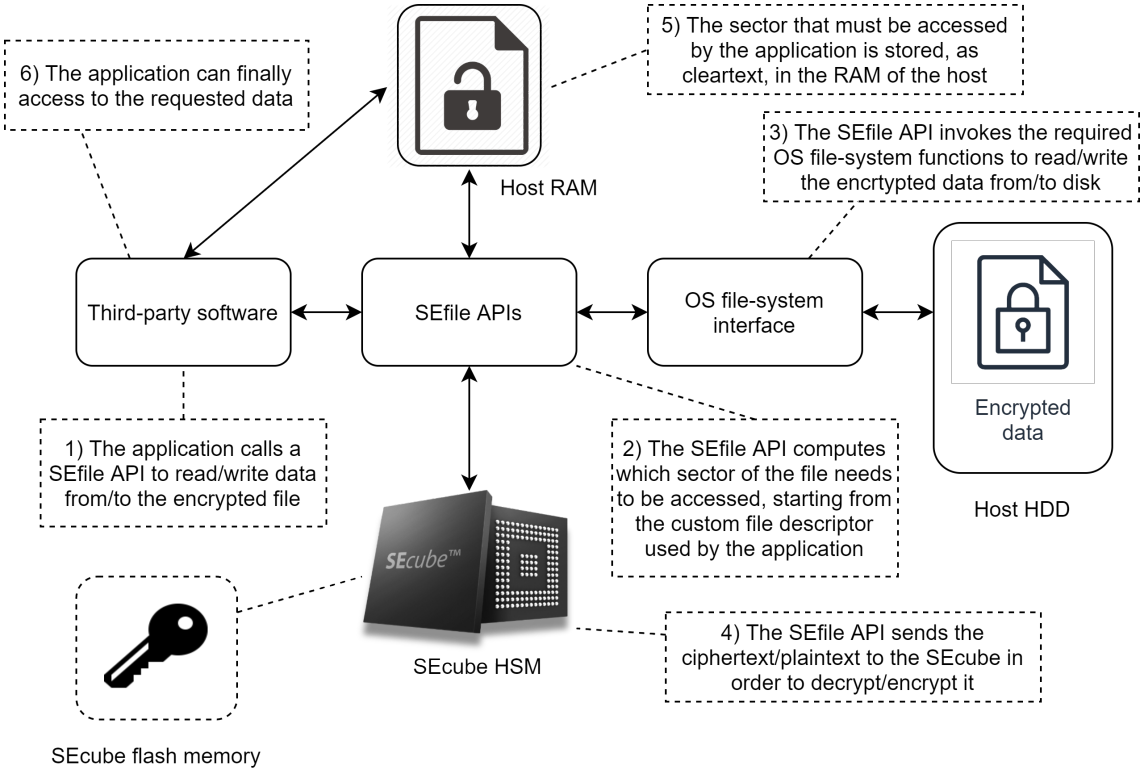


Figure 20: Simplified step-by-step **SE*file*™** overview.

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[23]. 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.

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

Most modes require a unique binary sequence, often called an 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 plaintext 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) supports only AES-256 as cipher algorithm. **SE*file*™** leverages it by using the Counter (CTR) mode of operation. The simplest of the encryption modes 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[24].
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 start from a randomly selected initialization vector, generated using ad-hoc functions provided from 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 (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

[24] Helger Lipmaa, Phillip Rogaway, and David Wagner. Comments to NIST concerning AES modes of operation: CTR-mode encryption. 2000

is ”squeezed” out[25]. 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 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;
```

[25] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. ”The Keccak sponge function family: Specifications summary”.

```
  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 exploiting 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*™ has been developed with the goal of being independent from the software working at the higher levels, but this is not always true. **SE*file*™** , in fact, uses a custom file structure (Figure 19) to grant security properties to the data; however, this structure may collide with other assumptions

that the application at the higher levels may have about the underlying file system.
This was the case of SQLite, a popular SQL database engine. The first release of **SE*file*™** provided a demo of the library applied to the SQLite Browser tool, however, **SE*file*™** was not working properly with SQLite.
Since SQLite was chosen as an important component to implement the **SE*key*™** library, a **SE*file*™** version specifically customized for SQLite has been developed. This custom **SE*file*™** version allows to implement an encrypted SQLite database with all the security properties of **SE*file*™** . The main difference between the standard **SE*file*™** library and the customized one is the usage of a slightly different file sector structure which is compatible with the assumptions of SQLite about the underlying file system. Because of this custom structure, you will also find custom APIs that must be used exclusively with SQLite encrypted databases; this APIs are the same of the standard version but they have been modified to work with the different sector structure.
In conclusion, if you simply want to try **SE*file*™** and/or **SE*key*™** and if you do not need to manage encrypted SQL databases, then you can simply ignore the APIs related to the encrypted 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.

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 Unix operating systems. 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 is 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 SEcube 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 exploit the constructors available for the SEfile object. Notice that 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 to 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, exluding 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 Premise

SE*link*™ is currently undergoing a significant revision. It will be totally rewritten, in order to be more coherent with the other libraries of the **SE*cube*™** such as **SE*file*™** and **SE*key*™** . The new version of **SE*link*™** is expected to be released in late Q3 or early Q4 of 2020. The documentation in this chapter is therefore to be considered outdated but it is kept here in case anyone needed further details about the first verison of **SE*link*™** .

7.2 Introduction

SE*link*™ is a software application that uses the **SE*cube*™** open platform to secure the network traffic. It can encrypt network streams originating from any application, regardless the application-level protocol.
SE*link*™ is a reference implementation of an application on top of the L1 API for **SE*cube*™** . By using **SE*link*™** it is possible to add a secure network layer to any software, without modifying its code. In other words, the user can employ any of his/her favorite network-enabled software (e.g., web browser, remote desktop viewer) and entrust the customizable security features to the **SE*cube*™** platform, in a way that is completely transparent to the user, allowing him to exploit the benefits of the security functionalities without having deep knowledge about security.
The software does not need to be aware of the presence of **SE*link*™** and will function as usual, because **SE*link*™** intercepts connections at a lower level.
SE*link*™ is made up of two macro-components (Figure 21):

- **Client-side software**: it is installed on the host that initiates the connection. It includes a driver, a background service and a graphical user interface. The driver intercepts outgoing connections and redirects them to the service. The service bridges the connection to the destination, applying the encryption layer
- **Server-side software**: it is installed on the host that accepts the connection. It includes a background service and a graphical configuration utility. The service is symmetrical to the client-side service, bridging the encrypted connection to its final destination.

The Client-side components are:

- **SE*link*™** driver
- **SE*link*™** service
- **SE*link*™** GUI.

The driver is needed to intercept all new TCP connections, system-wide, while not modifying any application (Figure 22). The driver redirects all connections to the service, which can decide whether each should be encrypted or not. The graphical user interface is a distinct application too, because Windows services are not intended to create GUI elements within the user's session.
The server-side components are:

- **SE*link*™** gateway
- **SE*link*™** gateway web UI.

On the server's side a "gateway" application, running as daemon, bridges the secure connections created by a **SE*link*™** client host to any server software (Figure 23). This daemon can by configured by directly editing its configuration file, or by using a graphical interface through any web browser.

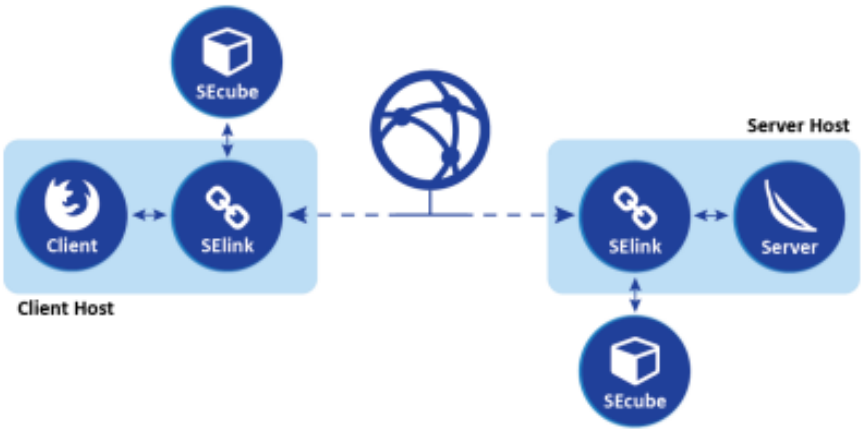


Figure 21: **SE*link*™** architecture.

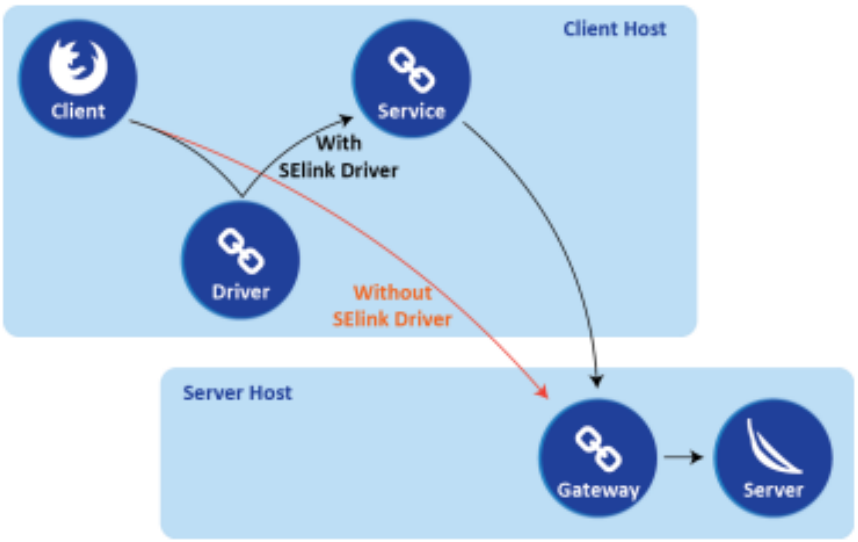


Figure 22: Connection establishment process: each directed arrow represents a TCP connection request.

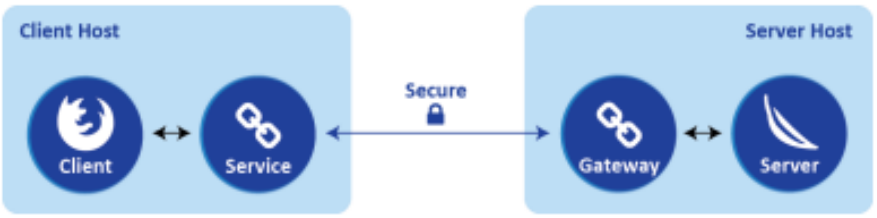


Figure 23: Final connections: each bidirectional arrow represents a TCP connection.

7.3 SE*link*™ driver

The choice of implementing a driver has been taken since it cannot be tampered from other user-mode applications, making it more secure and reliable; moreover, it leaves no traces within the applications' executable memory, so it goes undetected by most software protection frameworks and does not interfere with applications.
After some experiments with other frameworks, the driver was ultimately developed with the Windows Filtering Platform (WFP) API. The superseded Transport Driver Interface (TDI) API was discarded a priori, because of its deprecation. NDIS, despite having all the needed features, would have required more effort to achieve the same result.
The current WFP solution is very lean, consistent with the practice of performing the least needed amount of work within the kernel. In fact, any faulty code within the kernel can cause a system crash, hence the need to restrict kernel-mode code to the minimum necessary amount.
An important reason why the filtering logic has been implemented in user-space is that not all operations can be performed as easily within the kernel. To be more specific, routines within the kernel run at different Interrupt Request Levels (IRQL), depending on their priority. Each of the IRQLs has a mask for interrupts, with higher levels masking more interrupts. At higher IRQLs functions must complete as soon as possible, deferring any time-consuming work. Also, functions that can cause a page fault, such as operations on pageable memory, must only be performed at the lower levels that do not mask the specific interrupt. Specifically, WFP filter callback functions run at `IRQL = DISPATCH_LEVEL`, and cannot access pageable memory nor perform file I/O.
The **SE*cube*™** platform uses file I/O to communicate with the device; its host API would need to be completely redesigned to use it within a driver. The driver is written in C, as most drivers are. There is no official support to any other language for driver development with the Windows Driver Framework. As a matter of fact, most functions from the C standard library cannot be used either.
For network filtering the API Hooking approach was considered and discarded because of the disadvantages discussed in the previous chapter. LSPs have not been considered for the deprecation issue. Other user-space options involve either modifying applications to add support for **SE*link*™** or restrict the applicability of **SE*link*™** to applications supporting proxy protocol (e.g., HTTP/HTTPS proxy or SOCKS). Hence the decision to develop a kernel-mode filter.
The NDIS API was considered first, but considerable effort would have been required to keep track of TCP streams from the level at which NDIS filters operate. Windows Filtering Platform (WFP) has been chosen over NDIS because of its support for transport level filtering. Also, WFP is recommended by Microsoft itself as a possible alternative.
The driver has the main purpose of redirecting outgoing TCP connections from any application to a local proxy service.
The driver needs to exchange information with the service mainly for the following purposes:

- Sending information about the redirected connection, so that the service can bridge the connection to the intended destination
- Getting the process ID of the service.

Connection information is passed to the service through the redirect context, as stated before. As for the PID, the service sets a WFP Provider Context at startup, which can be read within the driver. Since the provider context cannot be read within the callback functions, because of the IRQL, a periodic timer task running at `IRQL = PASSIVE_LEVEL` polls the provider context at a fixed interval.

7.4 SElink service

The **SE*link*™** service is the main component of the client-side software: all the connections originating from the machine go through the service, which decides which connections should be encrypted and which should not.
The choice of the programming language, C++, is driven by multiple reasons:

- It is mandatory that the service performs well with many connections (hundreds to thousands) because it must handle most of the network traffic of the machine
- The service needs some data structures such as maps, sets, linked lists, which are not part of the C standard library
- The service must interface directly with the Windows API to be able to communicate with the driver
- The service must use a **SE*cube*™** device, whose API is only available for C.

C++ is a multi-paradigm language that seamlessly combines low-level and high-level features. It can directly include C code, perform raw virtual memory access, but is also suited to object-oriented and functional approaches. Plus, following the introduction of the C++11 standard, it has vast standard library that includes a common interface to some OS-specific features such as threading.
The boost library is also included for the following features:

- Command line parameters: used to parse the command line, generate a help message and useful error description messages
- JSON file parsing: for the configuration files
- Filesystem operations: to manipulate paths and access files through a OS-independent interface
- String formatting: used for some log messages
- Logging: to conveniently categorize log messages and possibly redirect them to a log file
- Hashing: for faster matching of an array/string against a set of arrays/strings
- Event-based socket I/O: used to accept connections and react to network events.

Since this application shares much of the logic with its cross-platform server-side counterpart, most of the classes are designed to be cross-platform, and are reused in **SE*link*™** gateway.
The service is intended to be a high performance and low footprint application; therefore, C++ was chosen for its unique combination of performance and high-level features. Boost further extends C++ with cross-platform implementations of additional features. Notably a high-performance event-driven socket I/O library is included in Boost.
The service oversees encrypting/decrypting and forwarding outgoing connections on the client side.
SE*link*™ service is configured by means of an external GUI application, which can send some commands to the service to perform operations or retrieve information.
The commands are hereby listed:

- `reload`: reload the filter rules file and update the filter rules
- `status`: query the device and service status

- `discover`: discover all connected devices, returning a list of their paths and serial numbers
- `set_device`: select a device by its serial number and pass its password
- `reset`: disconnect from the device and clear the device configuration.

The service creates a named pipe at startup, on which it listens for connections from the GUI. For each connection to the named pipe, a single request packet is processed and a response packet is returned, then the pipe is disconnected.
A thread is devoted to inter-process communication only. All the I/O operations for the named pipe are handled asynchronously with Windows overlapped I/O functions, and all wait operations, in addition to waiting on I/O completion with a timeout, also wait on a stop event. This makes it possible to immediately stop the inter-process communication thread when the service closes, without resorting to polling.
SE*link*™ service can register itself as a Windows service, and be managed through the Windows' services interface.
Running a process as a Windows service implies that:

- It runs as the SYSTEM user
- It can be configured to run at startup
- It can be enabled, disabled, started, stopped or deleted from a simple command line interface or from the Windows Management Console.

Summarizing, starting **SE*link*™** is as simple as issuing the following commands to the Windows command prompt:

```
sc start selinksvc
net start selinkflt
```

7.5 SE*link*™ GUI

An important part of the project is allowing users, with little prior knowledge of cryptography and programming, to use SElink for the practical purpose of encrypting network traffic.
Another point for the GUI is the necessity to provide an interactive dialog window to log into the **SE*link*™** , and avoid leaving the passphrase in the command line or configuration file.
The GUI could not be included within the service, because there is no conventional way of presenting a GUI to the user while running as the SYSTEM user. A simple GUI, here described briefly, has been developed for the cited reasons.
The Windows Presentation Framework (WPF) is a framework for Windows GUI applications, part of the .NET framework. It was chosen for the development of **SE*link*™** GUI because of its overall features and good integration with the operating system. The choice was not restricted to cross-platform framework, because the application is specifically made for the SElink client-side software, running on Windows.
Among the .NET and WPF features that were used there are:

- Windows tray icon functionalities
- JSON parser
- Customizable DataGrid GUI component.

When the application is run, it creates a clickable icon in the tray area, which is used to access the configuration dialogs. There are two main dialogs: the device selection dialog and the filter rules dialog, explained in the next Sections (Figure 24).

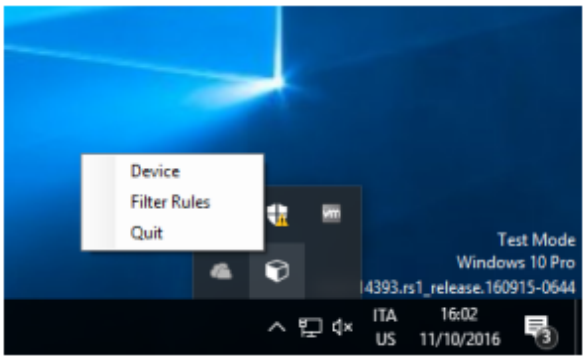


Figure 24: Tray icon interface.

For the service to connect and work properly, the user must select a suitable device (Figure 25).

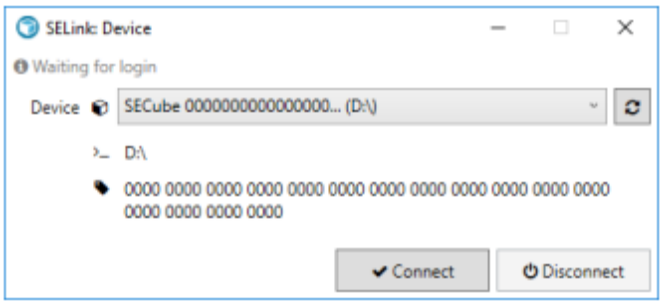


Figure 25: Device selection interface.

A passphrase is required to unlock the device (Figure 26).

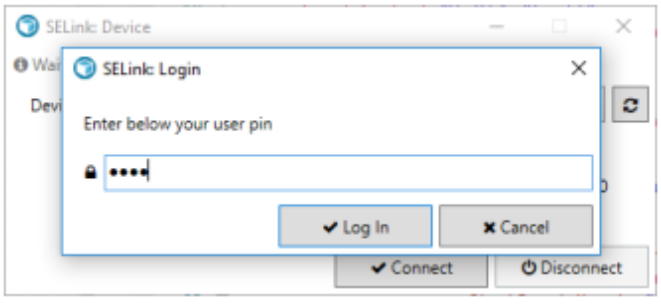


Figure 26: Unlocking the device.

After confirming the passphrase, the outcome of the operation is shown via balloon notification (Figure 27). The GUI does not directly perform any operation on the devices. Instead it uses the service by sending commands through a named pipe.

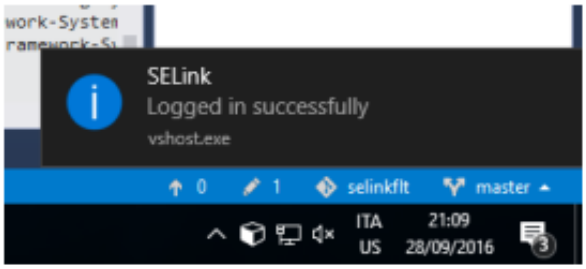


Figure 27: Login notification.

The filter rules window provides an easy way to edit the filter rules configuration file, minimizing the risk of making mistakes or producing an invalid configuration. Based on the DataGrid WPF component, it allows adding, removing and editing the filter rules. Rules are processed in order and only the first matching rule is considered.
Therefore, there is the possibility for a rule to cover one of the following rules, which means that any entity matching the second rule also matches the first rule, so the second rule will never be used. The insertion of a masking rule may or may not be intentional, therefore a warning is shown if there are any masked rules, and which is the first masking rule for each masked rule (Figure 28).

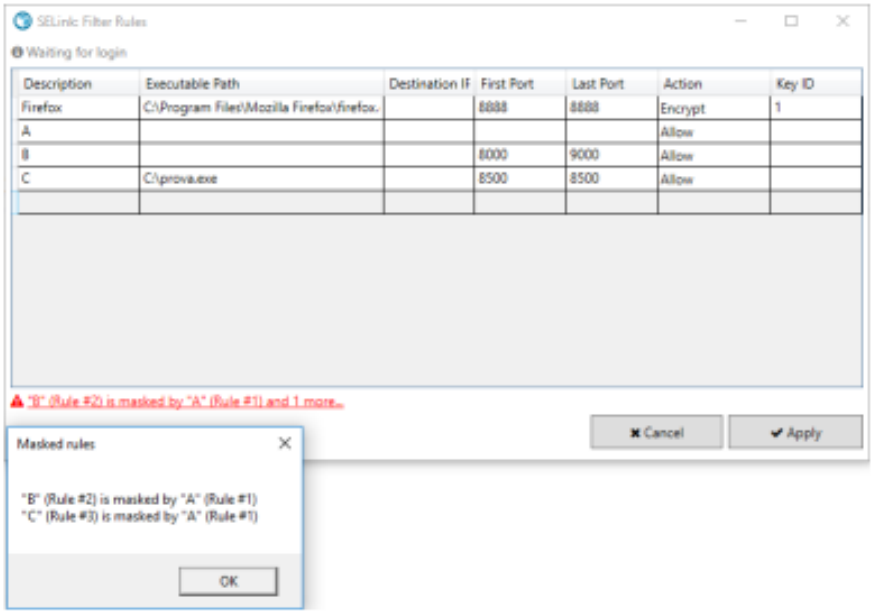


Figure 28: Filter rules editor.

All fields within each rule are validated, to prevent creating an invalid configuration. Each row of the grid shows a relevant error if it does not pass a validation rule (Figure 29).

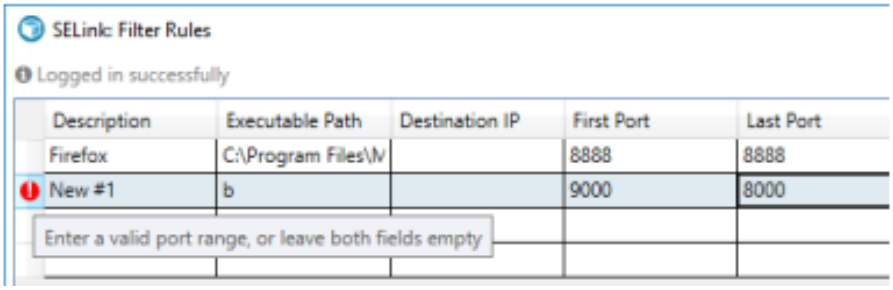


Figure 29: Filter rules editor highlights an invalid rule.

As soon as the rules are applied, the rules file is written and the service is signaled to reload the file.

7.6 Running the provided demo

The main goal of this demo is to show of a Windows software (gateway) that transparently intercepts and encrypts/decrypts TCP streams, and its server-side counterpart, a cross-platform proxy that intercepts and decrypts/encrypts the streams.
This software use the Layer1 API of the **SE*cube*™** open platform to perform the encryption step. The software can encrypt network streams originating from any application transparently, without the need to modify or recompile the application.
In addition, the final product is very be user-friendly and configurable by means of a graphical user interface, both client-side and server-side, and easy to install and capable of running in background silently, requiring minimum user attendance after setup.
To establish a secure communication, a properly configured host running the client software must initiate a connection towards a properly configured host running the server software.
In most cases the user will only need to install and configure the client software.

7.6.1 Requirements

Client requirements:

- 64-Bit Windows 10
- **SE*cube*™** device[26].

Server requirements:

- Any of:
  - Windows 7 or newer[27].
  - Linux
- **SE*cube*™** device.

7.6.2 The Client software

The only prerequisite is to install the client software is the Windows C++ Runtime library, freely accessible on-line[28].
There are three components to be installed for the software to operate properly:

[26] **SE*link*™** can be used without a **SE*cube*™** device, for testing purposes or custom configurations.
[27] Daemon mode not yet supported on Windows.
[28] https://www.microsoft.com/it-it/download/details.aspx?id=48145

- **SE*link*™** driver
- **SE*link*™** service
- **SE*link*™** GUI.

The provided setup executable installs all the components at once.
Since the driver is not signed, Windows 10 needs to run in Test Mode to install the driver.
To do so:

1. Open an administrator command prompt and execute the following command:
   `bcdedit /set TESTSIGNING ON`
   Warning: this command disables driver signature check enforcement on Windows.
2. Reboot

Finally, run **selink-setup.exe**.
After the setup procedure, the **SE*link*™** client software can be easily configured through the GUI application. You can access the device configuration window by left-clicking the tray icon and choosing Device. The status of the service and the device is shown on the top.
To connect a **SE*cube*™** device:

1. Select a suitable device from the drop-down menu. If a device does not show up, press the refresh button on the right.
2. Press the Connect button
3. Insert your user pin and click the Login button
4. The device configuration window should disappear, and a notification should appear shortly after.

To disconnect it, it is simply needed to press the Disconnect button.
You can access the filter rules configuration window by left-clicking the tray icon and choosing Filter rules.
Filter rules are used to manage outgoing connections, to decide which will be encrypted and with which key. A filter rule is made of:

- A condition to select connections based on some parameters:
  - **Executable path**: full path to the source application's executable file
  - **Destination IP**: destination IP address
  - **First port, Last port**: destination port range.
- An action to be taken with the matching connections:
  - **Action**: one of *Allow, Block* or *Encrypt*
  - **Key**: the key ID to be used when encrypting.

In order for a condition to match a connection, all the parameters must match. An **empty parameter** stands for **any value**.
Within the GUI you may define a list of filter rules, to select different actions for different connections. For example, the user might assign a different encryption key to each application you are going to use. Note that the order in which rules appear **is important**: the first matching rule is chosen, regardless of the following rules. If a connection does not match any rule, it is allowed (not encrypted) by default.
You can create a new rule by doing any of the following:

- Right-click on the grid and select Insert after or Insert before
- Fill the last row of the grid

The rules' fields must comply with the following constraints:

- **Description** must be shorter than **64 characters**
- **Destination IP** must be a valid **ipv4** or **ipv6** address
- **First port** and **last port** must describe a **valid port range**
  - first port, last port must be both natural numbers lesser than 65536, or both empty
  - if not empty, first port must be lesser than or equal to last port
- The **key** must be
  - An integer number if the action is *Encrypt*
  - Empty if the action is *Allow* or *Block*

Tip: you can override the default action by adding a rule with empty condition fields at the end of the list.
Drag and drop a row over the desired position to move the corresponding rule. Right-click on a row and select Delete to delete it.

7.6.3 Server side

To install the needed software on server side it is sufficient to:

1. Install g++ and the boost development libraries for your system[29]
2. Change directory into the **SE*link*™** source code directory and build the **SE*link*™** gateway:

   ```
   # make
   ```

3. Install
   - Default installation

     ```
     # make install
     ```

     The software and configuration files will be installed in "/opt/selink/"
     *It will be configured to run without a **SE**cube™ device, using the keys from "/opt/selink/keys.json"*
     The systemd unit will be installed in "/usr/lib/systemd/system/selinkgw.service"
   - Custom installation
     - Copy "bin/selinkgw" and any needed configuration file to a target directory
     - Customize the "system" unit in "example/selinkgw.service" and copy it to the appropriate location for the system.

The command to start the daemon is

```
# systemctl start selinkgw
```

To stop it launch

```
# systemctl stop selinkgw
```

[29] e.g., `pacman -S base-devel boost` on Arch Linux

The configuration is made up of a simple list of port mappings. Each entry contains:

- A description text
- The port on which encrypted connections will be accepted
- The host and port to which connections will be redirected, unencrypted
- The key id to use for encryption.

You may configure the SElink gateway in two ways:

- Using the web UI
- Using the configuration file.

To use the web UI:

1. Install python3, python3 modules bottle, and jsonschema on your system
2. cd into the gwconfig directory and run the Web UI as root (assumes a default installation)

```
# python3 gwconfig.py --use-token
```

3. Open the link on the terminal output with a web browser

Within the web UI you can add a new rule or delete an existing one.
To add a new rule, press Add and insert the rule. The mappings' fields must comply with the following constraints:

- Listen port and Redirect port must be natural numbers lesser than 65536
- Redirect host must be a valid ipv4 or ipv6 address.

To delete a rule, press the trash icon in the last column of the row to delete. After each modification, please remember to press the Apply button. The new configuration will be saved and applied immediately.
Vice versa, it is possible to edit directly the configuration file as follows:

1. Edit the configuration file with a text editor. Please refer to "example/selinkgw.json" for an example file
2. Signal the daemon:

```
# systemctl reload selinkgw
```

7.6.4 Advanced configuration

The driver can be configured to only filter connections with destination ports within a port range, and allow anything else, regardless of whether the service is running.
The driver configuration is stored in the following registry key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Service\selinkflt\Parameters`.

Name	Type	Description
`PortFirst`	`REG_DWORD`	First port of filtered port range
`PortLast`	`REG_DWORD`	Last port of filtered port range
`ServicePort`	`REG_DWORD`	Port on which the service is listening for redirected connections

As a convention on file paths, any path starting with ":" is relative to the executable's path. For example, if the executable is located in "C:/SElink/selinksvc.exe", then ":selinksvc.json" points to "C:/SElink/selinksvc.json".
Any relative path is relative to the current working directory, which depends on how the process was created.

INI files
Command line options for the **SE*link*™** service and **SE*link*™** gateway may be specified either by passing them as parameters on the command line, or setting them into dedicated configuration files:

- For **SE*link*™** service, create a file named "selinksvc.ini" in the same directory as the executable.
- For **SE*link*™** gateway, create a file named "selinkgw.ini" in the same directory as the executable.

An example of a valid ".ini" configuration file is:

```
provider=soft
keys=:keys.json
```

Please note that only long options (i.e., no short keys) are allowed in the configuration file.

SE*link*™ service options

Option	Value	Description
`--help, -h`	(none)	Show a help message
`--log, -l`	log file path	Set the location of the log file. Defaults to ":selinksvc.log" when running as service, or none when running in foreground.
`--config, -c`	configuration file path	Set the location of the filter rules configuration file. Defaults to ":selinksvc.json".
`--provider, -p`	provider type	Set the provider type. Can be `soft` or `secube`.
`--keys, -k`	path to key collection file	Set the keys. Only required if the provider is of type `soft`.
`--port, -w`	port	Set the driver connection redirection port.
`--foregrou`	(none)	Run in foreground instead of running as a service.

SE*link*™ gateway options

Option	Value	Description
`--help, -h`	(none)	Show a help message
`--log, -l`	log file path	Set the location of the log file. Defaults to ”/var/log/selinkgw.log” when running as service, or none when running in foreground.
`--config, -c`	configuration file path	Set the location of the filter rules configuration file. Defaults to “:selinkgw.json”.
`--provider, -p`	provider type	Set the provider type. Can be `soft` or `secube`.
`--keys, -k`	path to key collection file	Set the keys. Only required if the provider is of type soft.
`--serial-number, -s`	path to key collection file	Set the keys. Only required if the provider is of type `soft`.
`--pin, -z`	path to pin file	The user pin to log into the SE*cube*™ device will be read from the specified file. Only required if the provider is of type `secube`.
`--foreground, -f`	(none)	Run in foreground instead of running as a daemon.

SE*link*™ gateway web UI options

Option	Value	Description
`--help, -h`	(none)	Show a help message.
`--host`	host	Host on which the web UI server will listen.
`--port`	port	Port on which the web UI server will listen.
`--config, -c`	configuration path	Path to gateway configuration file.
`--pidfile`	pidfile path	Path to gateway pidfile.
`--use-token`	(none)	Generate a random token to restrict access to the web UI.
`--debug`	(none)	Enable debug mode.

7.6.5 The APIs commented

SE*link*™ uses the connection redirection feature in WFP, available for redirecting entire TCP streams to a different destination, possibly for filtering. Filtering in WFP is done by registering callback functions (named Callouts) intercepting the desired operations (e.g., connect, send, receive) at the desired layer.
Within the driver there are two callouts: one for the ipv4 connect redirection layer (`FWPM_LAYER_ALE_CONNECT_REDIRECT_V4`) and one for the ipv6 connect redirection layer (`FWPM_LAYER_ALE_CONNECT_REDIRECT_V6`), which means that all connection requests are intercepted. A filter specification is associated to the callout, restricting the intercepted requests to TCP connections on a user-defined port range.
Resources from the Microsoft website, such the Bind or Connect Redirection feature documentation and the sample driver projects, have been taken as reference for the implementation of the driver.
Any intercepted connection request is redirected to the local proxy, that will possibly encrypt the

connection and forward it to the intended destination. First, within the callout, a redirect context is filled with some data that will be useful for the user-space filter. The redirect context is a WFP feature to attach arbitrary data to the connection request, which can then be retrieved by a different application.
For later applying filter rules, the following parameters are added to the redirect context:

- Original source and destination address and port
- Process ID of the process which generated the request.

```
//Fill the redirect context
ConnectRequest->localRedirectContextSize = REDIRECT_CONTEXT_SIZE
    ;
RedirectContext = (SOCKADDR_STORAGE*)ConnectRequest->
    localRedirectContext;
RedirectContext[0] = ConnectRequest->localAddressAndPort;
RedirectContext[1] = ConnectRequest->remoteAddressAndPort;
RtlZeroMemory(&RedirectContext[2], sizeof(SOCKADDR_STORAGE));
ProcessId = (UINT64)InMetaValues->processId;
RtlCopyMemory(&RedirectContext[2], &ProcessId, sizeof(UINT64));
```

Then, the destination address and port are changed within the request headers, which effectively causes the socket to connect to a local service instead of its original destination. WFP also requires the target process ID to be set for local redirection, so the designated field is filled with the service's PID.

```
//Redirect to localhost
INETADDR_SETLOOPBACK((PSOCKADDR)&(ConnectRequest->
    remoteAddressAndPort));
INETASSR_SETPORT((PSOCKADDR)&(ConnectRequest->
    remoteAddressAndPort), RtkUshortByteSwap(Globals, ServicePort
    ));
ConnectRequest->localRedirectHandle = Globals.RedirectHandle;
ConnectRequest->localRedirectTargetPID = Globals.
    RedirectTargetPID;
FwpsApplyModifiedLayerData(ClassifyHandle, (PVOID)ConnectRequest
    , 0);
```

7.7 SE*link*™ APIs

This Section provides a brief overview about the **SE***link*™ APIs.
For more details about their implementation, please refer to the Doxygen-based documentation.

```
void SElink_buffer_init(SElink_buffer* buf)
```

This function initializes a buffer object to an empty buffer.

```
void SElink_buffer_ensure(SElink_buffer* buf, size_t newsize)
```

This function ensures the capacity for **SE***link*™ buffer.

```
void SElink_buffer_grow(SElink_buffer* buf, size_t required)
```

This function ensures capacity for **SE*link*™** buffer, which is expanded to the desired size (must be a power of 2).

```
void SElink_buffer_free(SElink_buffer* buf)
```

This function disposes of the **SE*link*™** buffer.

```
void SElink_raw_init(SElink_raw* sr)
```

This function initializes the **SE*link*™** raw context.

```
void SElink_raw_reserve_keys(SElink_raw* sr, size_t nkeys)
```

This function reserves the memory space for keys buffer.

```
void SElink_raw_reserve_algorithms(SElink_raw* sr, size_t
   nalgorithms)
```

This function reserves the memory space for algorithms buffer.

```
void SElink_raw_add_key(SElink_raw* sr, uint32_t id, const
   uint8_t* fingerprint)
```

This function adds a key to the **SE*link*™** raw context.

```
void SElink_raw_add_algorithm(SElink_raw* sr, uint32_t id, const
    uint8_t* fingerprint)
```

This function adds an algorithm to the **SE*link*™** raw context.

```
void SElink_raw_clear(SElink_raw* sr)
```

This function disposes of keys and algorithms buffers.

```
void SElink_raw_d1_write(SElink_raw* sr, size_t* len, uint8_t*
   data)
```

This function writes D1 packets.

```
bool SElink_raw_d1_get_size(SElink_raw* sr, size_t len, const
   uint8_t* data, size_t* size)
```

This function retrieves the size of D1 packets.

```
bool SElink_raw_d1_read(SElink_raw* sr, size_t len, const
   uint8_t* data)
```

This function reads D1 packets.

```
bool SElink_raw_d1_match(SElink_raw* sr, size_t len, const
   uint8_t* data)
```

This function finds a match in D1 packets.

```
void SElink_raw_d2_write(SElink_raw* sr, size_t* len, uint8_t*
   data)
```

This function writes D2 packets.

```
bool SElink_raw_d2_read(SElink_raw* sr, size_t len, const
   uint8_t* data)
```

This function reads D2 packets.

```
void SElink_raw_s1_write(SElink_raw* sr, size_t* len, uint8_t*
   data)
```

This function writes S1 packets.

```
bool SElink_raw_s1_read(SElink_raw* sr, size_t len, const
   uint8_t* data)
```

This function reads S1 packets.

```
bool SElink_raw_s1_match(SElink_raw* sr, size_t len, const
   uint8_t* data)
```

This function finds a match in S1 packets.

```
void SElink_raw_h_write(SElink_raw* sr, uint8_t* header, size_t
   data_size)
```

This function writes data packet headers.

```
void SElink_raw_h_read(SElink_raw* sr, const uint8_t* header,
   size_t* data_size)
```

This function reads data packet headers.

```
void SElink_raw_fingerprint(const uint8_t* salt, size_t len,
   const uint8_t* data, uint8_t* fingerprint)
```

This function generates raw fingerprints.

```
uint16_t SElink_raw_secube_import(SElink_raw* sr, se3_session* s
   , uint16_t key_size, uint16_t algo_type)
```

This function imports keys and algorithms from the device.

```
size_t SElink_raw_packet_size(size_t data_size)
```

This function retrieves the expected packet size.

```
void SElink_init(SElink* ctx, se3_session* s)
```

This function initializes the SElink context.

```
void SElink_destroy(SElink* ctx)
```

This function destroys the SElink context.

```
uint16_t SElink_duplex_write_request(SElink* ctx, SElink_buffer*
    buf)
```

This function writes duplex requests.

```
uint16_t SElink_duplex_get_request_size(SElink* ctx,
   SElink_buffer* buf, size_t* request_len)
```

This function retrieves the length of duplex request packets.

```
uint16_t SElink_duplex_reply(SElink* ctx, SElink_buffer* buf)
```

This function replies to duplex requests.

```
uint16_t SElink_duplex_read_response(SElink* ctx, SElink_buffer*
    buf)
```

This function reads duplex responses.

```
uint16_t SElink_simplex_write_invite(SElink* ctx, SElink_buffer*
    buf)
```

This function writes simplex invites.

```
uint16_t SElink_simplex_read_invite(SElink* ctx, SElink_buffer*
   buf)
```

This function reads simplex invites.

```
uint16_t SElink_write(SElink* ctx, size_t data_len, const
   uint8_t* data, SElink_buffer* buf)
```

This function writes data packets.

```
uint16_t SElink_read_header(SElink* ctx, size_t data_len, const
   uint8_t* data, size_t* remaining_bytes)
```

This function reads data packets' headers.

```
uint16_t SElink_read(SElink* ctx, size_t data_len, const uint8_t
   * data, SElink_buffer* buf)
```

This function reads data packets.

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 *encryption keys* to be used by applications based on the **SE*cube*™** device. **SE*key*™** supports the secure generation, distribution, and usage of encryption keys in domains characterized by multiple users who need to share sensitive data to be protected with the **SE*cube*™** device.

8.1 Key Management System

A Key Management System (KMS) is a purely software or hybrid hw-sw system with the aim of handling, administrating and protecting cryptographic keys. A KMS must provide several functionalities, such as the management of the lifecycle of keys, including their *generation, usage, storage* and *deletion*. Other essential features of a KMS are the distribution of the keys to authorized entities and the protection of the keys while they are stored on physical devices.
According to the National Institute of Standards and Technology (NIST)[30]: "*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.*"[31].
Encryption keys must be protected, otherwise they become useless: encryption keys are only as good as the security used to protect them. In order to implement a well-designed KMS, a set of policies, procedures and components (both hardware and software) must be selected.

8.2 Encryption Keys

Encryption keys used within the **SE*cube*™** *Open Security Platform* are always dedicated to *symmetric* encryption algorithms, such as AES-256. Symmetric encryption algorithms use the same key both for encryption and decryption, they are fast and resilient against quantum cryptopgraphy attacks but they require both parties to know the same encryption key.
SE*key*™ is able to manage any type of encryption key, independently from its length, algorithm and usage. However, since the open source SDK of the **SE*cube*™** supports only the AES-256 algorithm, the APIs of the KMS have been written and tested having in mind the usage of 256 bit symmetric keys.
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);

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

- **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 30 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.

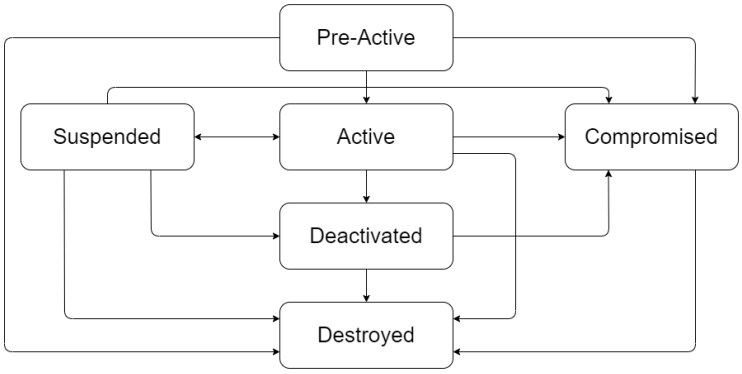


Figure 30: Key State diagram.

- 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*™

SE*key*™ is distributed among multiple **SE*cube*™** devices: it is composed by one Admin's **SE*cube*™** , and one **SE*cube*™** for each user of the system. The Admin's **SE*cube*™** can be seen as a *server* and the other **SE*cube*™** devices as *clients*. However, **SE*key*™** is not a centralized KMS where a server handles requests coming from the users. **SE*key*™** reverses the traditional client/server approach, the administrator in fact automatically pushes data to a predefined location while the users fetch these data according to their needs.

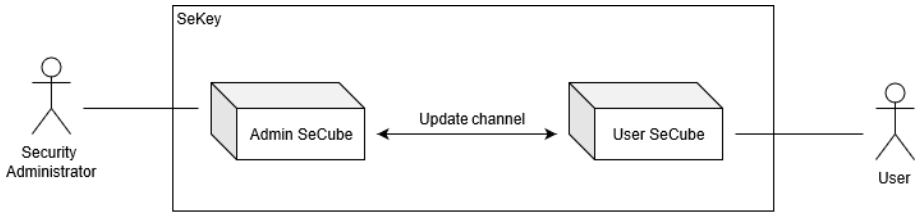


Figure 31: **SE*key*™** context diagram.

Therefore, **SE*key*™** is distributed also in terms of data. The KMS is focused not only on the encryption keys but also on other metadata that carry 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 replicated (with the push and fecth mechanism mentioned above) on the devices of the users. The data distribution mechanism is totally transparent to the administrator and to the users, the greatest advantage is that the users are automatically provided with the data they need. Therefore, the users can exploit the KMS locally and 'offline', resorting exclusively to the information they have on their personal **SE*cube*™** , without the need of contacting an external server. Notice that the administrator provides the data to the users according to the 'Least Privilege' principle [32].
Each **SE*cube*™** involved in **SE*key*™** adopts the following approach to store the data belonging to the KMS:

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

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

This data partitioning scheme is valid both for the administrator and for the users. The only difference among them is that the **SE*cube*™** of the administrator stores everything that is managed by **SE*key*™** (every key and every 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 on his **SE*cube*™**).

8.4 Administrator and users of SE*key*™

There are only two types of actors in **SE*key*™** : administrator and users. There is only one administrator, his job is to manage the KMS using specific APIs, for example he can add/delete users, add/delete/modify groups, add/delete keys and so on. On the other hand, the users simply fetch the changes applied by the administrator to the system and store them in their KMS configuration; then they can use a reduced set of APIs to perform operations such as retrieving a key, retrieving the list of the users belonging to a certain group and so on.
Notice that the roles of the administrator and of the users cannot overlap, meaning that the administrator is not a user of **SE*key*™** and a user cannot be the administrator. However, it is clear that the role of the administrator must be carried on by a real person who manages the KMS according to specific needs (i.e. the IT security manager of a company); notice that this person cannot act as administrator and user at the same time using the same **SE*cube*™** device. If the administrator of **SE*key*™** also needs to be a user of the KMS, then he/she must possess two different **SE*cube*™** devices and, ideally, two different host computers. The first pair **SE*cube*™** - host is used to work as administrator, the second pair is used to work as a user of the system.
One of the most important tasks performed by the administrator is the physical initialization of the **SE*cube*™** devices of the users. Each **SE*cube*™** can be initialized using an API provided by **SE*key*™** , so the administrator sets two PIN codes that must be unique to that device and that are used to access to it with admin privilege or user privilege. Notice that the administrator of **SE*key*™** will be able to store and see all the PINs of all **SE*cube*™** devices, on the other hand each user must receive only the PIN that allows him to access to his **SE*cube*™** with the lowest privilege. Depending on the PIN that is used to 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).

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 encryption keys between users who have a personal **SE*cube*™** device and belong to a certain environment. Within this context, users can be split in several *groups*, a specific security policy is associated to each group. 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 rigid security policy with respect to another one);
- the default cryptoperiod of the keys of the group (which is adopted if a specific key does not have a specific cryptoperiod or if its cryptoperiod is longer than the default one).

The concept of the group is strictly related to the one of *encryption keys*. **SE*key*™** , in fact, is designed such 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:

- the groups are the owners of the keys
- a user can use a key only if he/she is a member of the group that owns that key
- users who are member of the same group share all the keys of that group
- each group may own zero, one or multiple keys
- each group may own as many keys as the corresponding parameter specified by its security policy
- each key is owned by only one group
- a key cannot be owned by two or more groups, not even in different moments
- the ownership of a key cannot be transferred from a group to another group
- if a group is destroyed, all the keys owned by that group become 'zombie' keys (they become unusable and their owner is set to 'zombie' but they are not destroyed because they may still be required to decrypt old 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 encryption keys. A user can access only to the keys associated to the groups to which he/she 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.).

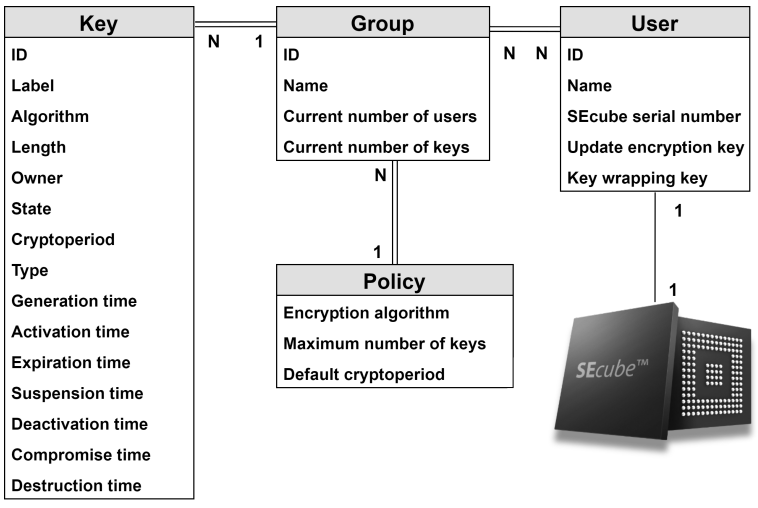


Figure 32: **SE*key*™** logical overview.

8.6 Use Cases

8.6.1 Security Admin's use cases

Among all the management use cases, the ones related to the creation of new objects (groups, users, keys) are the most complete. For this reason, they are described here, while the ones related to the modification and elimination of objects, that are similar and simpler, are not shown.

Use case UC1: Add Group
Application: **SE*key*™**
Actor: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in.
Post-conditions: A new group is added to **SE*key*™** .
Main scenario:

1. Security Admin 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: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in
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. Security Admin 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: Security Admin
Pre-conditions: Security Admin is already authenticated and logged in
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. Security Admin 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 *encryption 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's Use Cases

The purpose of the user is to exploit group keys in order to encrypt and decrypt data to be exchanged with other users. A user can decide to encrypt some data by choosing as recipient another user, an entire group to which he belongs or a set of users.

Use case UC4: Encrypt Data
Application: **SE*key*™**
Actor: User
Pre-conditions: User is already authenticated and logged in
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: User is already authenticated and logged in
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 library of SE*key*™ offers many APIs, however, inside the source code you will also find a lot of other functions that are used for internal purposes. The functions listed in this section are what you need to build and exploit the KMS, do not use other functions unless you are perfectly sure that you have understood the documentation and the meaning of what is done in the source code. For more details about the implementation of the SE*key*™ APIs, please refer to the Doxygen-based documentation.

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

These APIs are used respectively to start and stop SE*key*™ . The first function takes as input parameters the pointers to the L0 and L1 object that are necessary to exploit the SE*cube*™ device. Notice that those objects should be created by the higher levels, i.e. the 'main' of your application. These functions must be called when opening and closing the application that needs to work with SE*key*™ .

```
int sekey_admin_init(L1& l1, vector<string>& pin, string&
   userpin, string& 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 to build the KMS (i.e. adding users, groups and keys). Notice that this function can be executed exclusively by the administrator of SE*key*™ .

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

This function is used to physically initialize a **SE*cube*™** device dedicated to the user identified by the ID specified as first parameter. The second and third parameter are the two PINs to be set on the **SE*cube*™** , in particular, only the user PIN should be disclosed to the owner of the **SE*cube*™** . The last parameter is a list of PINs to be used to login to the **SE*cube*™** of the user before the initialization (the latest **SE*cube*™** firmware uses a default PIN which is 32 bytes long, each byte is equal to 0, so a list with a single PIN of all zeros should be enough).
Notice that the initialization of the **SE*cube*™** device of the user must be done after calling the `sekey_add_user()` function. However, you do not need to call this function immediately after the `sekey_add_user()`; you can delay it as long as you want, the only thing that matters is that you call this function before physically giving the **SE*cube*™** to the user.
Notice that this function can be executed exclusively by the administrator of **SE*key*™** .
For Linux users: when you connect the **SE*cube*™** of the user to the host computer, remember to mount the **SE*cube*™** otherwise the initialization might fail.

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

This function adds a new user to **SE*key*™** , using the required ID and username. If the ID is not available, an error is returned. The user, when is added, does not belong to any group by default. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

This function deletes the user specified by the ID passed as parameter from **SE*key*™** . The user will be completely deleted from the system (i.e. from the list of users and from any group to which the user may belong); moreover, the **SE*cube*™** of the deleted user will receive a special instruction that will result in the complete deletion of any information related to **SE*key*™** from the **SE*cube*™** of the user (all the keys will be erased from the internal flash memory and the database containing the metadata of the KMS will be emptied). Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

This function adds the given user to the given group. When the user is added, the other members of the group are notified of the presence of a new user (the administrator automatically 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 encryption keys associated to that group. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

This function deletes the given user from the given group; however, the user is still part of **SE*key*™** . This means that all the keys of that group will be deleted from the **SE*cube*™** of the user, moreover, the user will loose contact with other users whose only common group was the one specified as parameter. Notice that this function can be executed only by the administrator of **SE*key*™** .

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

This simple function allows to change the username of the user specified by the ID passed as parameter. This function can be executed exclusively by the administrator of **SE*key*™** .

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

These functions retrieve the details about a specific user (first one) or about all users (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_user`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any user of the system but only about the users that he knows, meaning the user that have at least a common group with him.
Notice that these functions can be used by the administrator and by the users.

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

This function adds a new group to **SE*key*™** , using the required ID, name, and security policy. If the ID is not available, an error is returned. Notice that the new group, by default, is empty and does not own any key. This function can be executed exclusively by the administrator of **SE*key*™** .

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

This function deletes the given group from **SE*key*™** . The users belonging to the group will simply be removed from the group and will not have access to its keys anymore. The keys associated to the group will not be deleted, they will be deactivated and their owner will be set to 'zombie' (i.e. like with processes in an operating system). The keys will still be usable, by the administrator, to decrypt old data; the administrator may also decide to delete all the keys of the group if they are not needed anymore. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

Simply change the name of an existing group. Notice that the other attributes of the group (i.e. ID and involved users) are not changed. This function can be executed exclusively by the administrator of **SE*key*™** .

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

Simply change the maximum number of encryption keys that can be associated to an existing group. Notice that the other attributes of the group (i.e. ID and involved users) are not changed. If the specified number of keys is lower than the current number of keys, the change does not take place. This function can be executed exclusively by the administrator of **SE*key*™** .

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

Change the default cryptoperiod used for the encryption keys of the specified group. Notice that the change is applied only for keys that will be activated after this change, the encryption keys that were activated in the past are not affected by this change therefore their expiration date will stay the same as before. This function can be executed exclusively by the administrator of **SE*key*™** .

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

These functions retrieve the details about a specific group (first one) or about all groups (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_group`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any group of the system but only about the groups to which he belongs. These functions can be used by the administrator and by the users.

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

This function adds a new key to **SE*key*™** , using the required parameters. If the ID is not available, an error is returned. Notice that the caller must provide also the owner of the key, meaning the ID of the group that is associated to the key (i.e. the key 'K11' is associated to the group 'G7'). The *cryptoperiod* is optional, the caller can insert '0' to use the default cryptoperiod of the group owner of the key. Finally, notice that the key type must be always '*symmetric_data_encryption*'. This function can be executed exclusively by the administrator of **SE*key*™** .

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

This function is used to activate the key specified by the ID passed as parameter. Upon activation, the expiration time is set depending on the cryptoperiod of the key. Notice that a key can be activated only if its status is 'pre-active' or 'suspended' (in this last case it is a re-activation). Once a key is activated, it can be used to encrypt data. This function can be executed exclusively by the administrator of **SE*key*™** .

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

This API is used to change the status of an encryption key, according to Figure 30. Notice that to change the status to 'active', there is a dedicated function. The status of a key must be changed by the administrator according to the conditions of the environment, for example if the administrator suspects that a key has been found by an unauthorized entity, then the key must be set to the 'compromised' state. Similarly, to destroy a key its status can simply be set to 'destroyed' and it will be erased from all **SE*cube*™** devices that have it in their flash memory. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

This function is simply used to change the label of an encryption key. The other attributes of the key remain unchanged. Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

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

These functions retrieve the details about a specific key (first one) or about all keys (second one), filling the object passed as second parameter. For more information about the attributes of the object of class `se_key`, check the Doxygen documentation. Notice that, when a user calls this function, he cannot retrieve the details about any key of the system but only about the keys that

are stored on his personal **SE*cube*™** , meaning the keys associated to the groups to which he belongs. Notice that these functions can be used by the administrator and by the users.

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

These APIs allow 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). Notice that in the first and in the third function, the most important rule to find the most secure key is to find an active key (i.e. usable for encryption) 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, for example two users may have only a group in common, which includes 3 other users; therefore also these users will have access to the chosen key and will be 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. Notice that these functions can be used only by the users of **SE*key*™** .

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

This function opens the log file generated by the **SE*cube*™** device with the serial number passed as first parameter and copies its content, as cleartext, into the string passed as second parameter; the string then 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. Notice that 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 to recover the integrity of the data of **SE*key*™** concerning a specific user. Let us imagine that a user has deleted the encrypted file used by **SE*key*™** from the MicroSD of his **SE*cube*™** or, for some reason, **SE*key*™** is not working properly on the **SE*cube*™** of the user. The administrator can try to solve the problem calling this function that, given the ID of the user and the serial number of his **SE*cube*™** , sends again to the user all the data that he is allowed to access (keys, groups, users, etc.). Notice that this function can be executed exclusively by the administrator of **SE*key*™** .

```
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 the APIs of **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 called automatically by **SE*key*™** , but the higher levels may safely call it from time to time (but it is not strictly necessary). Its purpose is to check if the administrator has sent any new data related to **SE*key*™** , concerning the current user. This function can be called exclusively by the users of **SE*key*™** , it cannot be called by the administrator.

9 Getting Started

9.1 The SE*cube*™ System Setup

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

9.1.1 Hardware resources

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

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

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

- 2+ GiB[33] 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[34],
- 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[35], at a price of 19.19 €. Your purchase should comprise the following items (Figure 33):

- 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)

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

Figure 33: Components purchased with the ST-Link/v2 programmer

The ST Discovery and ST Nucleo boards represent an affordable and flexible way for users to build project with a microcontroller from the STM32 family, choosing from the various combinations of performance, power consumption and features.
These boards do not require any separate probe as they both integrate a ST-Link/V2 programmer/debugger.
The STM32 Nucleo board comes with the STM32 comprehensive software HAL library together with various packaged software examples, as well as direct access to online resources. We suggest getting the boards through RS. It is important to clarify that you do not need to buy them both: you can buy only one board, and your purchase will in any case represent a valid alternative to the ST-Link/v2 programmer.
The recommended Discovery[36] and Nucleo[37] 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[38].
Your purchase should comprise the following items, depicted in Figure 34:

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

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

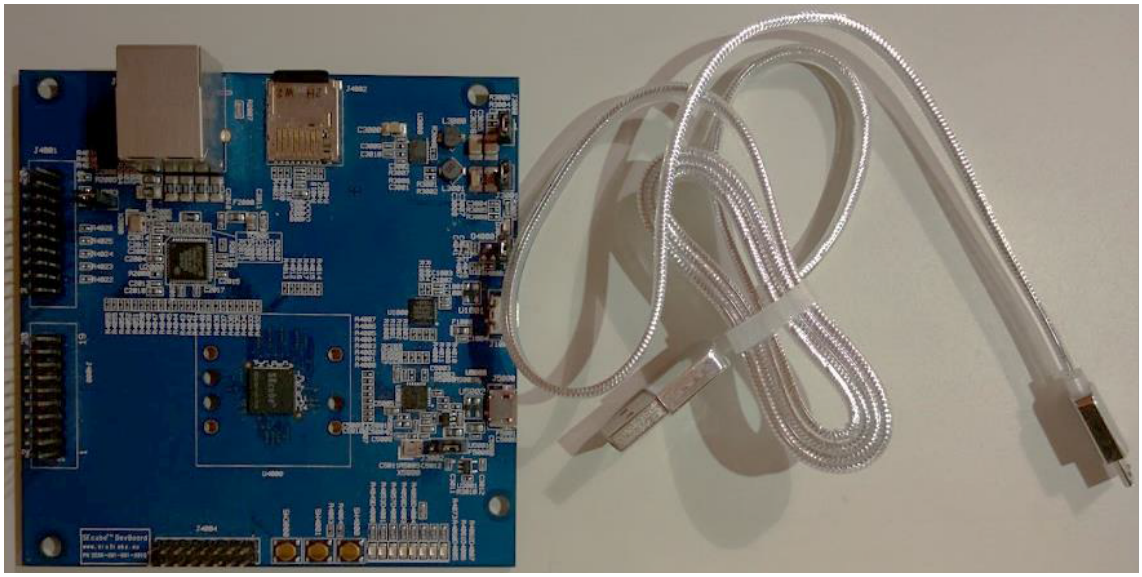


Figure 34: Components purchased with the **SE*cube*™** DevKit

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

Two **Operating Systems** are currently supported:

- Windows 7 (or later)[39]
- Linux with Kernel 2.6 (or later)[40]

[39] This procedure has been tested with Windows 7 Professional x64
[40] This procedure has been tested with both Linux Chakra kernel 4.5.7-1 x64 and Linux Ubuntu 14.04LTS x64

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

Version Required

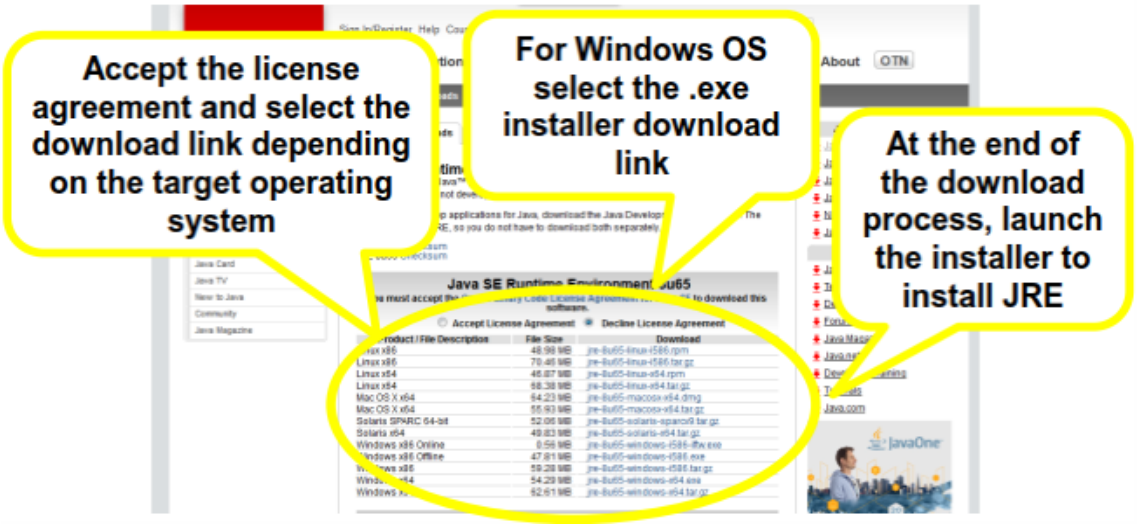
Version 8u111 (or later)

How to get it

The program is available free of charge from the Oracle website[41].

Installation hints

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



What is going to be used for

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

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

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

Version required

Version 4.6 Neon (or later)

How to get it

You need to download the Eclipse IDE for C/C++ Developers[42].

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”.

[41] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[42] https://www.eclipse.org/downloads/

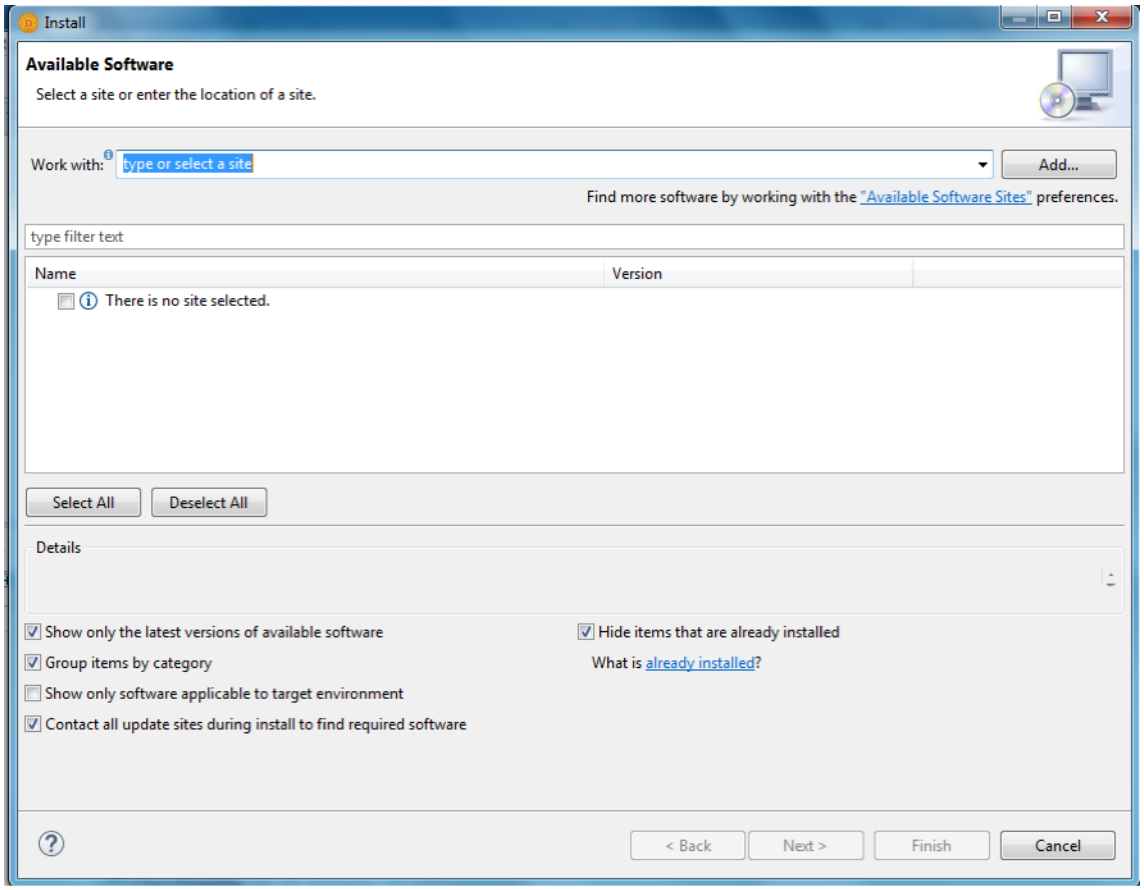
What it's going to be used for
Eclipse is the recommended IDE to develop applications that will run on the STM32F429 processor of the **SE*cube*™ DevKit**.

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

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

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

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



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[43].
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

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

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

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

Lattice Diamond® software is the leading-edge software design environment for cost-sensitive, low-power Lattice FPGA architectures. Lattice Diamond's integrated tool environment provides a modern, comprehensive user interface for controlling the Lattice Semiconductor FPGA implementation process.
Version required
Version 3.5 (or later)
How to get it
The software is downloadable free of charge online[44].
Installation hints
When downloading the software, it is possible to choose the free license.
What it's going to be used for
The software will be used for controlling the implementation process of the Lattice Semiconductor FPGA comprised within the **SE*cube*™** DevKit.

ST-Link v2 drivers provide support for the ST-Link/v2 programmer.
Version required
Version 4.0.0 (or later)
How to get it
The software is downloadable free of charge online[45].
Installation hints
During the installation procedure, it is possible to receive warnings from the Operating System if drivers are not properly signed; however, the installation procedure should not be interrupted.
What it's going to be used for
To allow the usage of the ST-Link/v2 programmer.

The **STM32 ST-LINK Utility** software facilitates fast in-system programming of the STM32 microcontroller families in development environments via the ST-LINK and ST-LINK/V2 tools.
Version required
Version 4.0.0 (or later)
How to get it
The software is downloadable free of charge online for Windows users[46].
Linux user, instead, can resort to the Open On-Chip Debugger (OpenOCD); this software is downloadable free of charge online[47].
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

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

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

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 35.
At the end of this Section you will have acquired a clear overview of the procedures to follow to start using the environment.

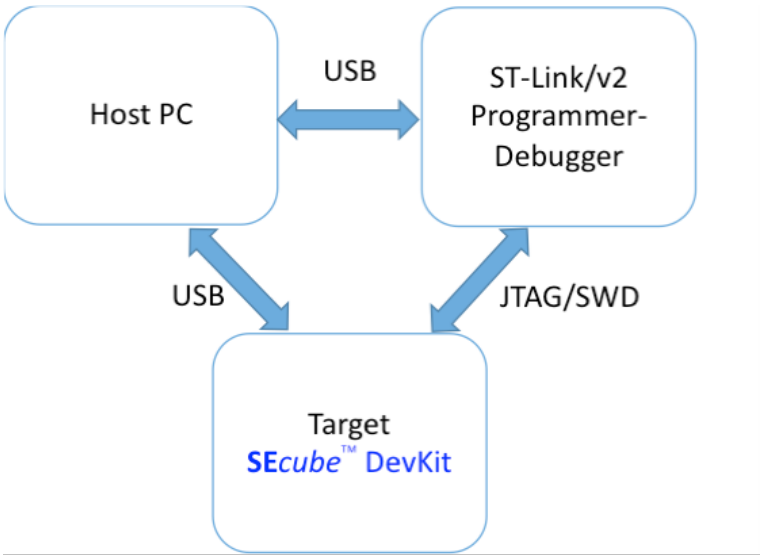


Figure 35: System Architecture

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 36, while a close-up on the JTAG connection is in Figure 37.

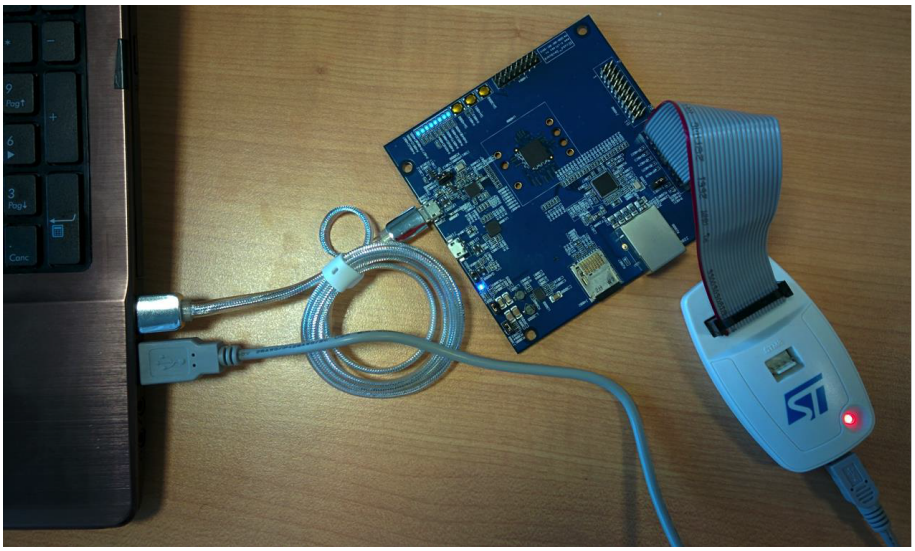


Figure 36: Connection between the STLink/v2 programmer and the **SE*cube*™** DevKit

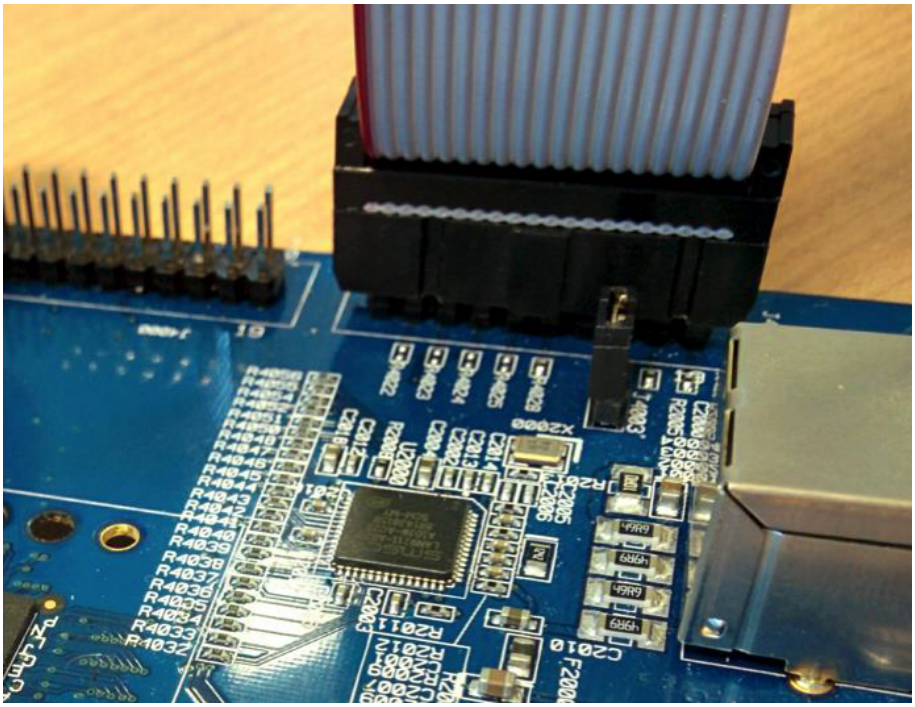


Figure 37: Connection between the STLink/v2 programmer and the **SE*cube*™** DevKit, close-up (highlighted in red) on the JTAG connector orientation

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 39 and described in the STM32 Nucleo-64 boards user manual[48]

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

2. Connect the **SE*cube*™ DevKit** with the programmer by means of the JTAG/SWD cable: the cable should be inserted on the JTAG docks of the DevKit (you must pay attention in inserting the plug with its protrusion oriented towards the inner side of the DevKit) and on the SWD dock of the board, accordingly to the schema in Figure 38

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

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

Figure 38: Programmer SWD pins schema

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

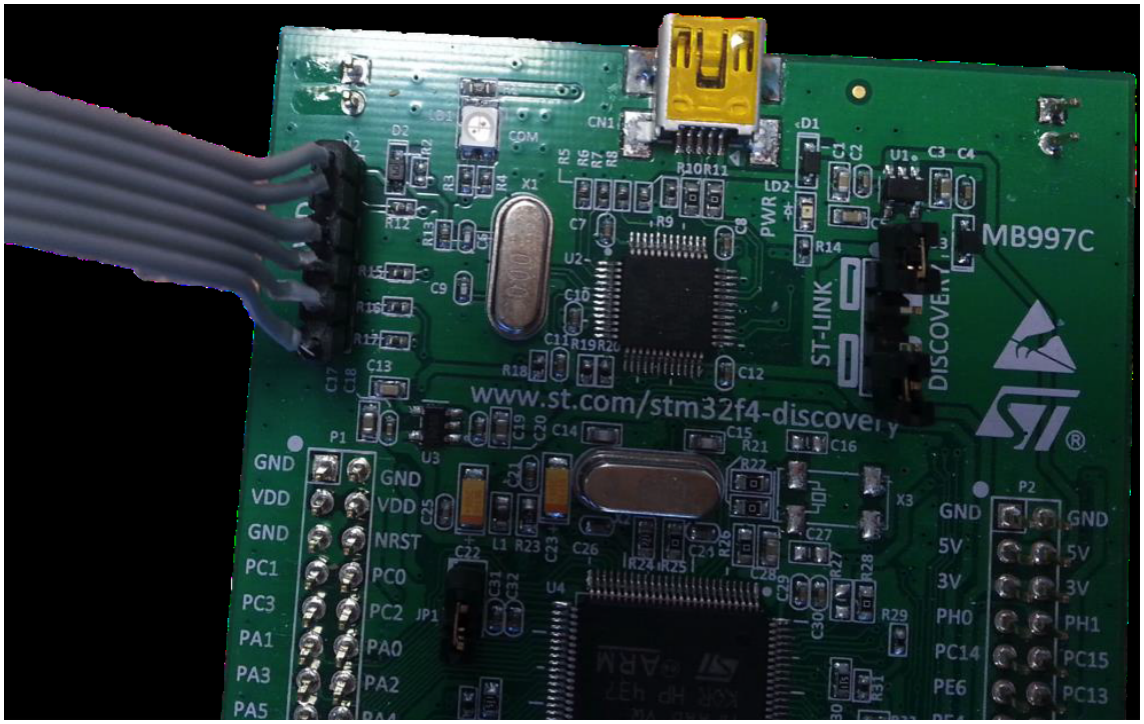


Figure 39: Jumpers configuration to isolate the ST-Link programmer on a Discovery board (highlighted in red, the same applies to Nucleo boards)

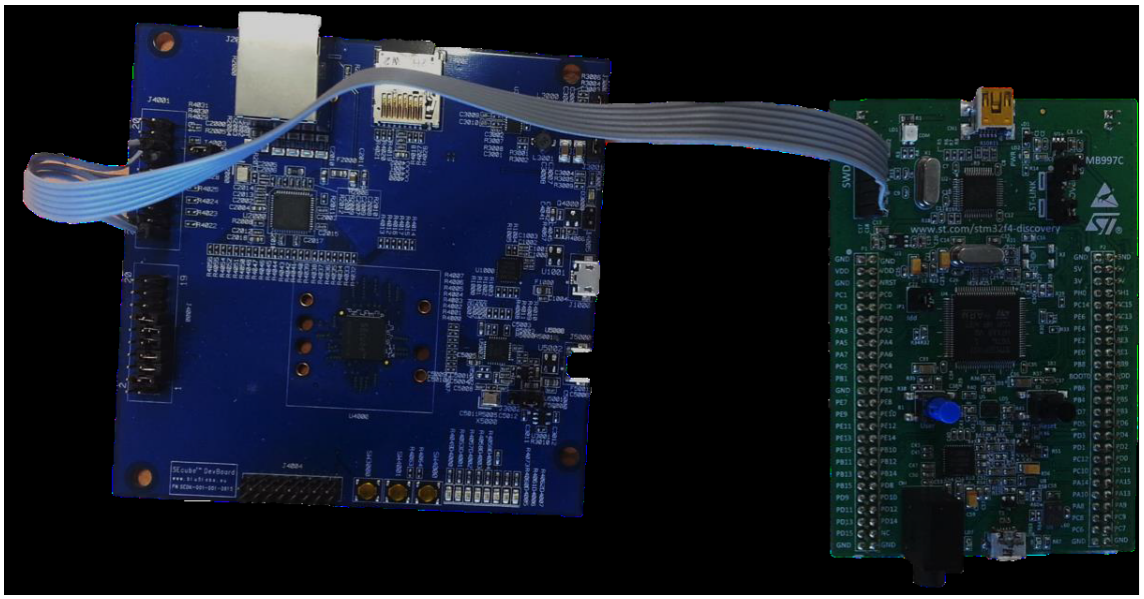


Figure 40: Connection between the Discovery board and the DevKit (the same applies to Nucleo boards)

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.

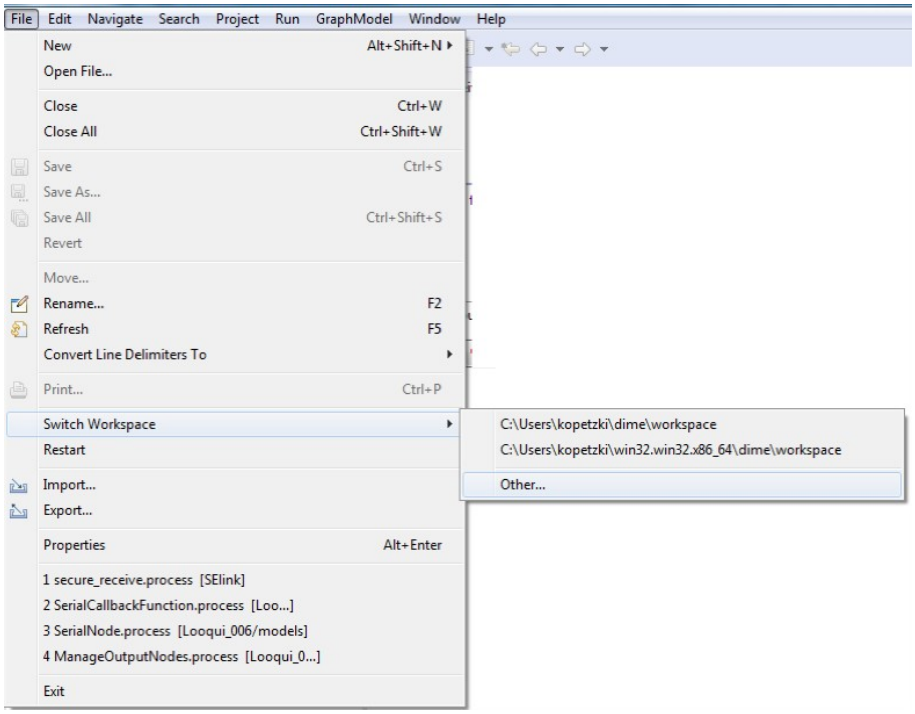
9.2 Installing the SE*cube*™ OpenSource Software Libraries

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.

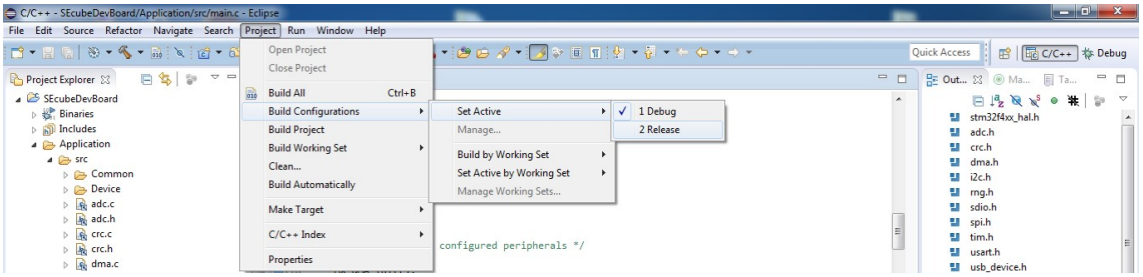
9.2.1 SE*cube*™ Open Source Software Libraries - Device Side

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

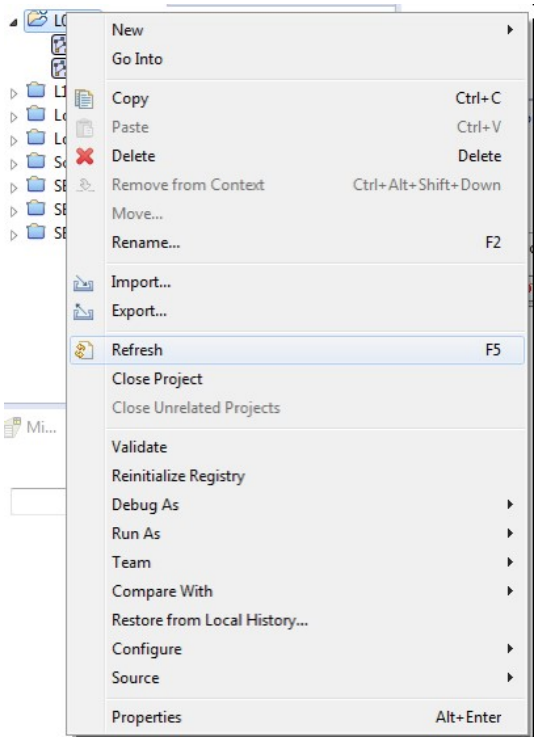
1. Download the SDK package containing the project from https://www.secube.eu/resources/open-sources-sdk/
2. Extract the .zip file to a known location
3. Open the folder ”SEcube SDK_v1.4_1” and extract the content of the archive ”SECubeSDK.tar.gz”
4. From your location, go to ”SECubeSDK/SDK device side/code”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



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

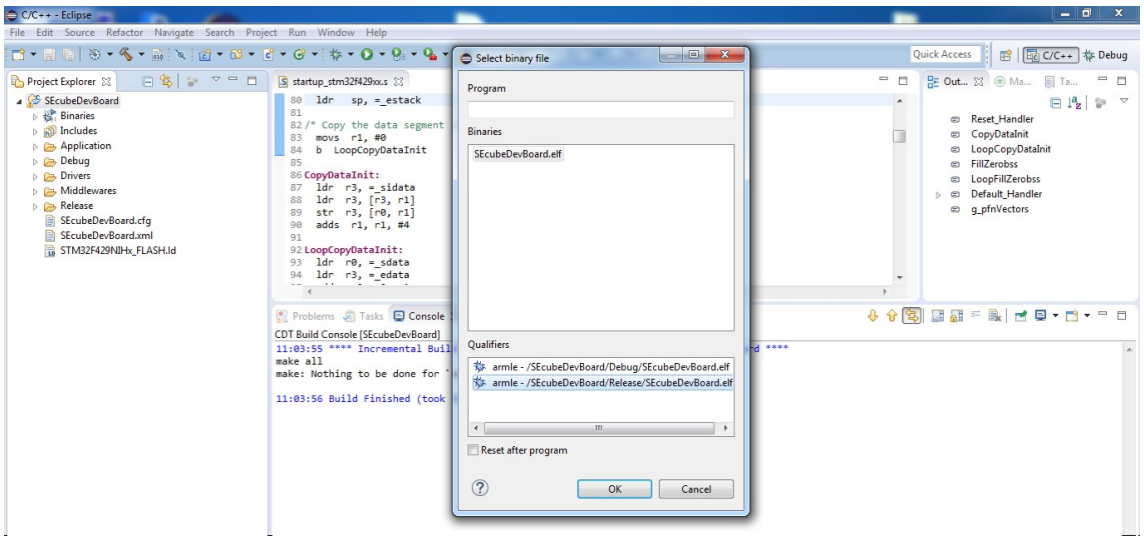


11. Build the project in Debug mode from «Project » Build All»
12. Set the Release configuration from «Project » Build Configuration » Set Active»
13. Build the project in Release mode from «Project » Build All»
14. Right-click on the project in the Project Explorer and select «Refresh»

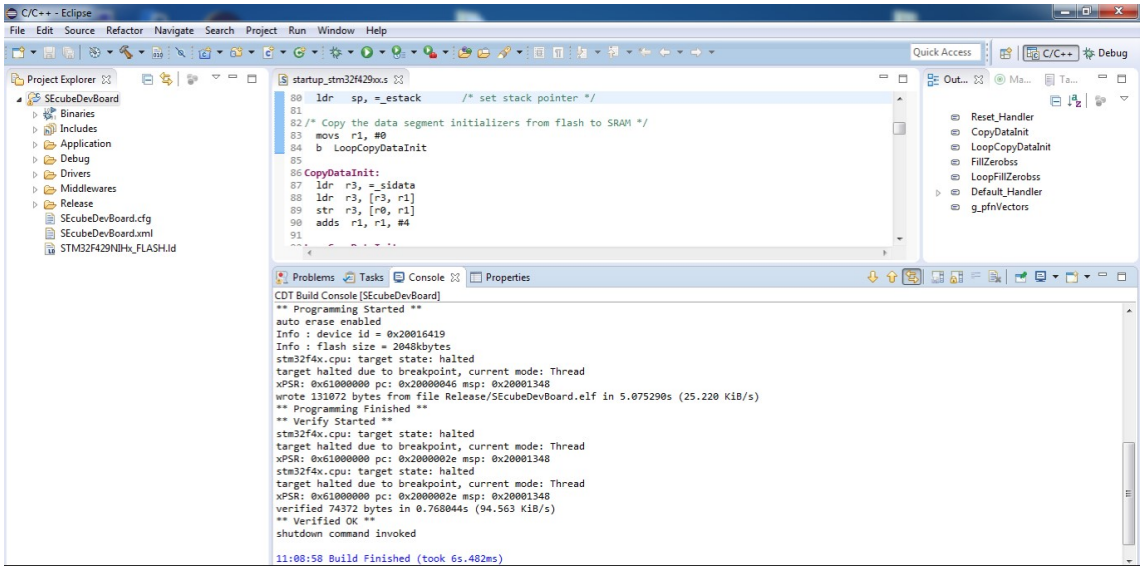


15. Connect the DevKit as described in previous section

16. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under «Target » Program Chip» (i.e., select the label containing the string “/Release”)



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

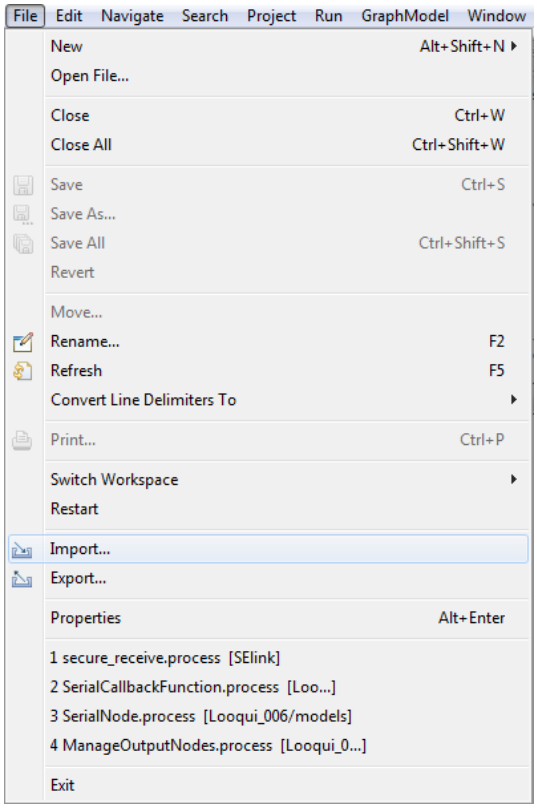


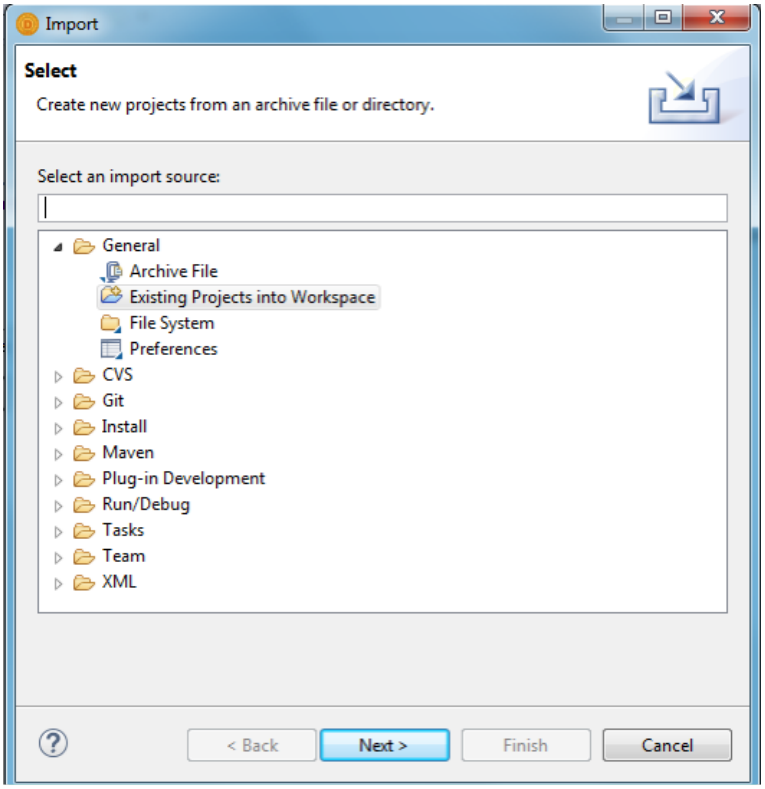
Now your DevKit is fully configured and the STM32 microprocessor is ready to be used to develop your security applications.

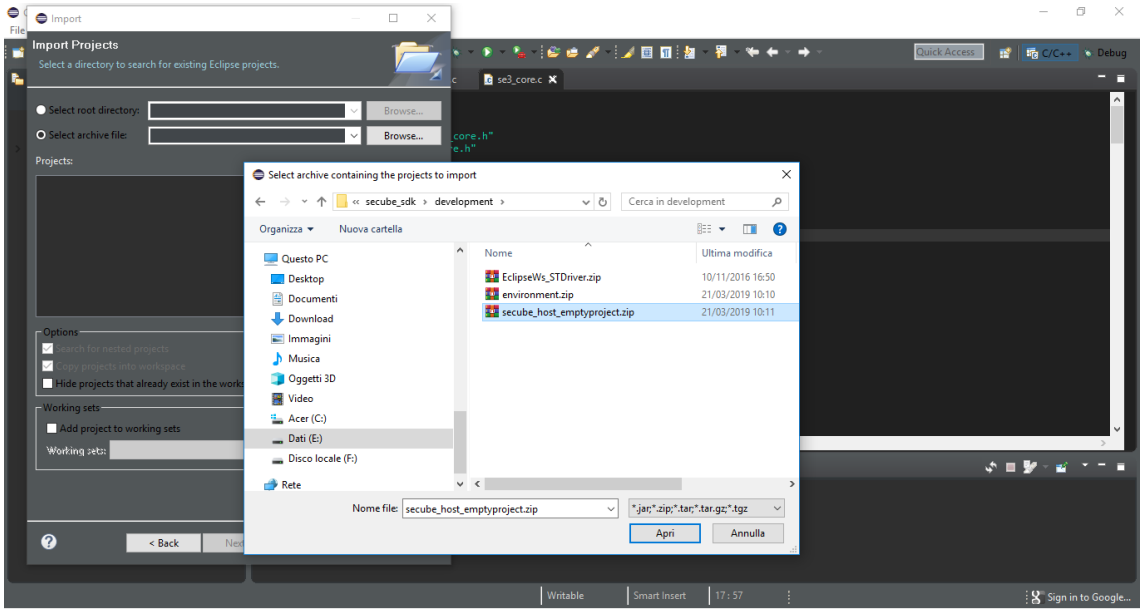
9.2.2 SE*cube*™ Open Source Software Libraries – Host Side

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 5
2. Launch Eclipse
3. Change Eclipse perspective to «C/C++ selecting Window » Perspective » Open Perspective » Other… » C/C++»
4. Import the project «File » Import…», select “Existing file into workspace” and press “Next”, Select archive file named ”secube_host_emptyproject.zip”, browse to the file downloaded and press “Finish”


File Edit Navigate Search Project Run GraphModel Window
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
1 secure_receive.process [SElink]
2 SerialCallbackFunction.process [Loo...]
3 SerialNode.process [Looqui_006/models]
4 ManageOutputNodes.process [Looqui_0...]
Exit


Import
Select
Create new projects from an archive file or directory.
Select an import source:
General
Archive File
Existing Projects into Workspace
File System
Preferences
CVS
Git
Install
Maven
Plug-in Development
Run/Debug
Tasks
Team
XML
< Back
Next >
Finish
Cancel



5. Set your preferred configuration from «Project » Build Configuration » Set Active»
6. Build the project from «Project » Build All»
7. Connect the **SE*cube*™** DevKit with the USB cable
8. Now you are ready to write your code within the “main.c” file in the Project Explorer, compile it and run it

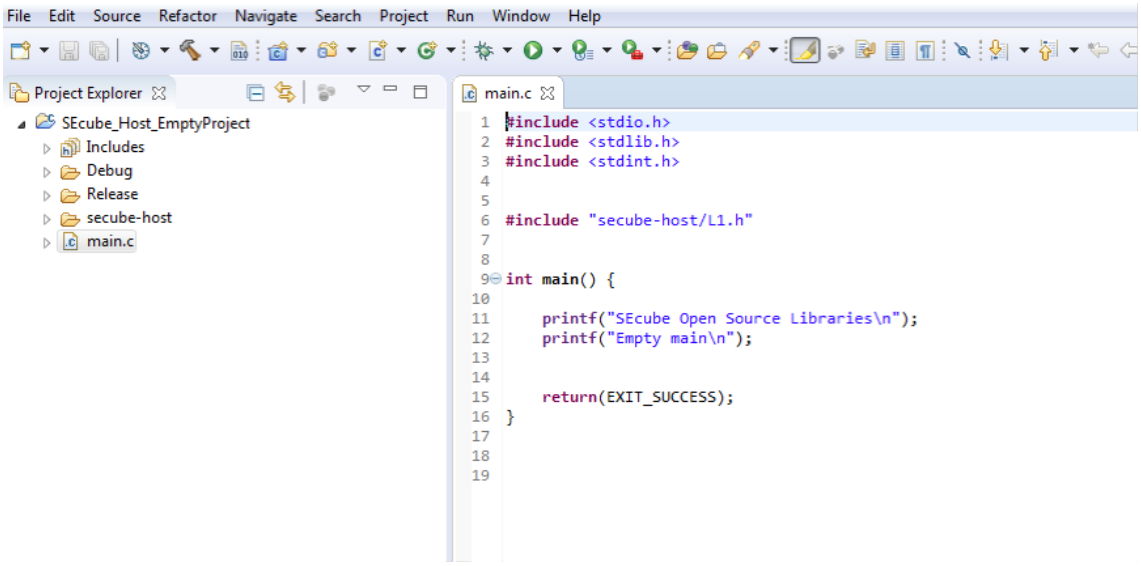
9.3 Running your first programs

This Section guides you to set-up, edit, and use the basic examples and tutorials provided as simple entry points to the environment, leveraging on the capabilities provided by the microcontroller.

9.3.1 Hello World (host-side)

Typically, the first program you run in a new environment is a basic one (e.g., one printing a string such as “Hello World” on the screen) intended to let you test that the environment is properly configured and to familiarize with the configurations needed to make the program code executable. The code shown in Appendix C is a first example of how to use the Open Source Libraries; it resorts only on the STM32 microprocessor as it tries to log in and out from your **SE*cube*™** DevKit.
Hereby we list a step-by-step guide to run your first program on the **SE*cube*™** DevKit.
Note: to run this software you must have already initialized your device.
To use it follow this step-by-step guide:

1. Launch Eclipse
2. Import the project following section 9.2.2
3. Open the file named “main.c”
4. Replace its content with the code shown in Appendix C (it can also be found in the SDK under the folder “SEcube_SDK/Libraries/Examples/HelloWorld/”)



5. Connect the DevKit with USB cable
6. Set the Release configuration from «Project » Build Configuration » Set Active»
7. Build the project from «Project » Build All»
8. Run the project

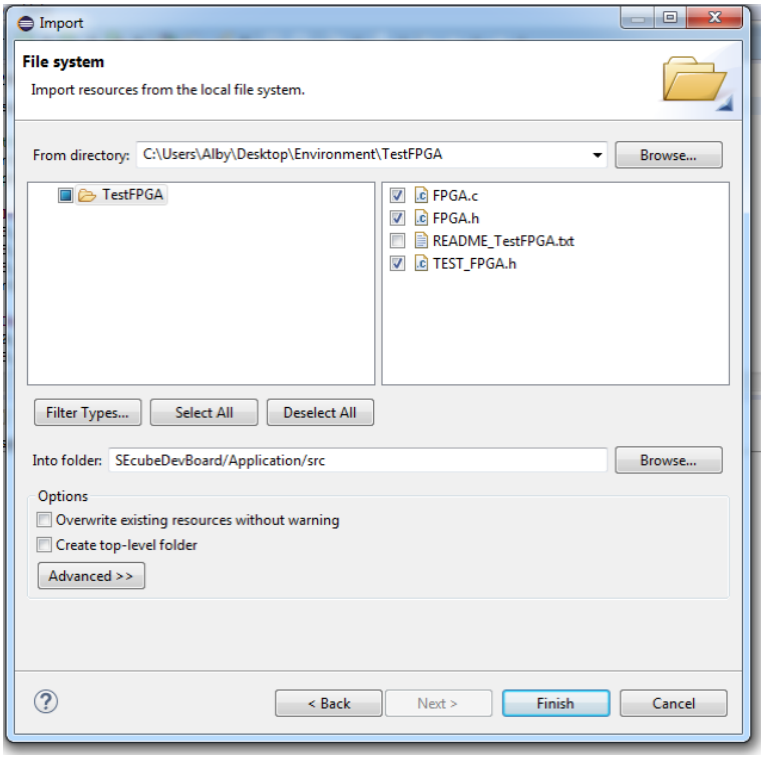
When setting up a project with the provided files for the API, a directory structure is implicitly assumed. As SEfile includes “se3/L0.h”, “se3/L1.h” etc., a folder “se3” must be present to contain the expected parts of the API. The tool for setting up the environment could serve as such an example project[49].

9.3.2 FPGA_LED (device-side)

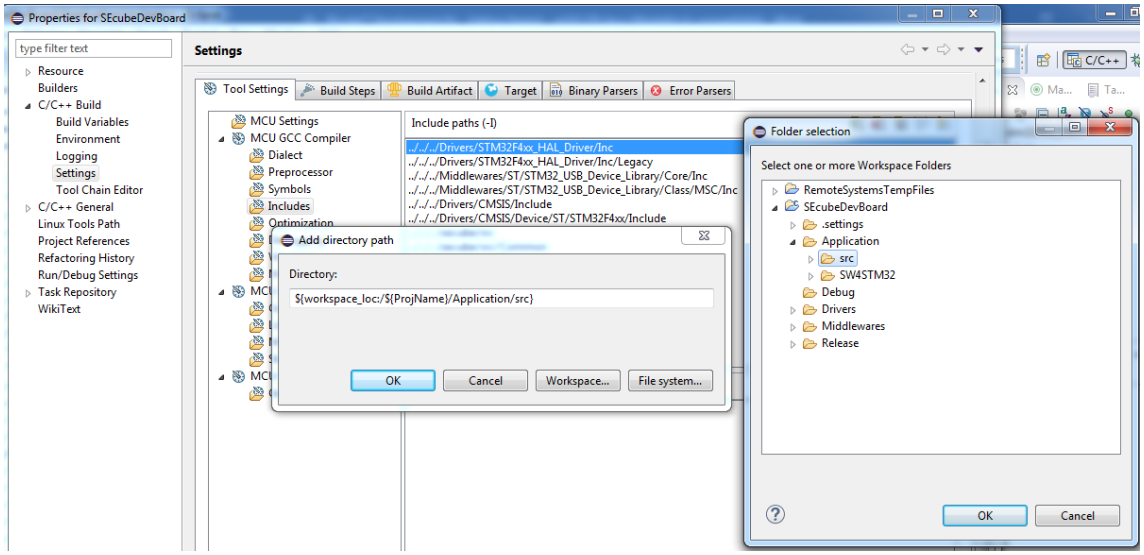
The procedure shown in this paragraph guides you to a first example of how to use the Open Source Libraries with the FPGA; it programs the FPGA embedded in the **SE*cube*™** chip to make the led blink.
Hereby we list a step-by-step guide to run this program:

1. Import the project as described in section 9.2.1
2. Import the necessary «File » Import…», select “Filesystem” and press “Next”
3. Browse to the directory where the SDK has been downloaded and then to the path “SE-cube_SDK/Libraries/Examples/TestFPGA/”
4. Select the files in that folder (FPGA.c, FPGA.h and TEST_FPGA.h); you might want to set also “Destination Folder” to “SEcubeDevBoard/Application/src” and then press “Finish”

[49] Freely available on-line (https://bitbucket.org/ede1/secube-command-line-tools.git)



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



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

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

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

```
/*Configure GPIO pin : PE2 */
GPIO_InitStruct.Pin = GPIO_PIN_2;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
```

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

9. Save the changes to all files and build the project
10. Connect the DevKit as described in previous section
11. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under «Target » Program Chip» (i.e., select the label containing the string "/Release")

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

9.3.3 A Functional Test

In this Section, we list the instructions to follow to run the Functional Test provided with the **SE*cube*™** DevKit.
At the end of this Section you will have acquired a clear overview of the procedures to follow to run programs on the environment.
Notice that to run this program your **SE*cube*™** DevKit must be already initialized.
In the sequel you can find a step-by-step guide to run the provided functional test:

1. Import the project following section 9.2.2
2. Locate the project in the folder "secube_sdk/Examples/FunctionalTest"
3. Locate the "funct_test.c" file in the Project Explorer
4. Right-click on it and select "Delete"
5. Import the new "main.c" file from «File » Import…», select "File System" and press "Next", browse the file "secube_sdkK/Examples/FunctionalTest" folder within the folder previously extracted, select the "funct_test.c" file and press "Finish"
6. Set the Debug configuration from «Project » Build Configuration » Set Active»
7. Build the project in Debug mode from «Project » Build All»
8. Set the Release configuration from «Project » Build Configuration » Set Active»
9. Build the project in Release mode from «Project » Build All»
10. Right-click on the project in the Project Explorer and select «Refresh»
11. Connect the **SE*cube*™** DevKit with USB cable
12. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under «Run as » Local C/C++ Application»

The provided functional test exploits capabilities offered by the Open Source SDK to show the basic common operations typically used with a **SE*cube*™** device; to provide an in-depth understanding of it, its code is hereby decomposed and commented.
The functional test can be found in the SDK in under the folder "secube_sdk/Libraries/Examples/FunctionalTest/". Starting from the includes instructions

```
#include <stdint.h>
#include <stdbool.h>
#include "../secube-host/L1.h"
```

it is possible to see how the SDK level "L1", offering high-level functions, is being used. The global declarations

```
static uint8_t pin[32] = {
  'a','b','c','d',  0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
};
static uint8_t pin0[32] = {
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
};
```

contain the PIN used to access the device. Each PIN is a generic value of 32-byte. Note that the value of `pin0` is used to login the first time just after the device is initialized.
Skipping to the main, there is a common snippet used to list all the **SE*cube*™** device connected to the host:

```
L0_discover_init(&it);
while (L0_discover_next(&it)) {
  printf("SEcube found!\nInfo:\n");
  printf("Path:\t%ls\n", it.device_info.path);
  printf("Serial Number: ");
  print_sn(it.device_info.serialno);
  printf("\n\n");
}
```

Here it is shown how you can retrieve information about the device (i.e., its path and its serial number) from the structure `it.device_info`.
Next step allows to open the device (if it is correctly found).

```
L0_discover_init(&it);
if (!L0_discover_next(&it)) {
  return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT);
}
if (return_value != SE3_OK) {
  printf("Error opening device\nAbort Tests");
  return(0);
}
else {
  printf("Open Device success\n");
}
```

Without knowing which **SE*cube*™** device is of interest, here the first available device is opened with `L0_open()` function from level "L0". It should be noted that every function in the SDK returns a value stating whether the operations has been completed successfully (`SE3_OK`) or not

(in this case the value returned is bigger than 1). So, it is good practice to always check for the expected return value after calling a function of the SDK.
When a device is opened, it does not offer any facilities but the echo service. This service can be used to test if the link between the host and the device side is working and the peripherals can communicate.

```
if (test_echo(&dev)) {
  printf("Echo success\n");
}
else {
  printf("Error Echo\n");
}
```

The function `test_echo()` does not belong directly to the SDK; it requires as only parameter a pointer to the device where to test the echo service. In this function, the buffer of random data to be sent is generated and the one to be received is allocated. The test will send by default 1 MiB of data.

```
enum {
  TEST_SIZE = 1 * 1024 * 1024
};
...
test_randbuf(TEST_SIZE, &sendbuf);
recvbuf = (uint8_t*)malloc(TEST_SIZE);
```

The test is repeated for NRUN times. The data sent can be at most SE3_REQ_MAX_DATA bytes.

```
for (rep = 0; rep < NRUN; rep++) {
  printf("Echo - Run %d... ", rep);
  sp = sendbuf;
  rp = recvbuf;
  n = TEST_SIZE / SE3_REQ_MAX_DATA;
  for (i = 0; i < n; i++) {
    r = L0_echo(dev, sp, SE3_REQ_MAX_DATA, rp);
    if (SE3_OK != r) goto cleanup;
    sp += SE3_REQ_MAX_DATA;
    rp += SE3_REQ_MAX_DATA;
  }
}
```

Finally, the data received in `recvbuff` is compared with the one sent `sendbuf`:

```
if (memcmp(sendbuf, recvbuf, TEST_SIZE)) goto cleanup;
printf(" -> OK\n");
```

To use any service offered by the **SE*cube*™** device other than the echo service, the login procedure is required. The function `test_login()` shows how to perform log in operations with the device. Several login operations are completed with different PINs:

- Log in as User with default pin (all zeroes) - It should fail when initialized

```
return_value = L1_login(&s, dev, init_pin, SE3_ACCESS_USER
    );
```

- Log in as User with correct PIN

```
return_value = L1_login(&s, dev, pin, SE3_ACCESS_USER);
```

- Log in as Admin with correct PIN

```
return_value = L1_login(&s, dev, pin, SE3_ACCESS_ADMIN);
```

- Log in as User with wrong PIN

```
return_value = L1_login(&s, dev, pin_aaaa, SE3_ACCESS_USER
    );
```

To pass this test, both User and Admin PIN must be set to “test” (otherwise you should change accordingly the content of the pin arrays).
The last operation in the function `test_algorithm()` shows how to get the list of encryption algorithms available. Also for accessing to this service, the login is required.
First a suitable array is created:

```
#define MAX_ALG 10
...
se3_algo alg_array[MAX_ALG];
```

Then, the “L1” function requesting the algorithm list is called

```
printf("Retrieving algorithm list...\n");
return_value = L1_get_algorithms(&s, 0, MAX_ALG, alg_array, &
   count);
if (return_value != SE3_OK) {
  printf("Error retrieving algorithm list");
  return(!SE3_OK);
}
```

Finally, for each algorithm received (i.e., available on the device), its characterizing information (e.g., its name) are printed out with the following code:

```
for (i = 0; i < count; i++) {
  memcpy(buff_name, alg_array[i].name,
     SE3_CMD1_CRYPTO_ALGOINFO_NAME_SIZE);
  printf("Algorithm name: %s\n", buff_name);
  printf("Algorithm type: %u\n", (unsigned int)alg_array[i].type
     );
  printf("Algorithm block size: %u\n",(unsigned int)alg_array[i
     ].block_size);
  printf("Algorithm key size: %u\n\n\n", (unsigned int)alg_array
     [i].key_size);
}
```

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

[50] www.secube.eu

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
  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 41, the injection can be executed through the USB Loader Tool, which comes together with the **USE*cube*™ Stick**.

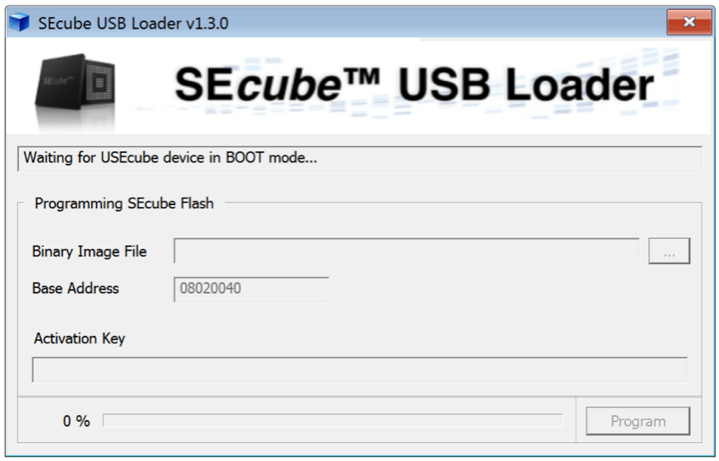


Figure 41: **SE*cube*™** USB Loader.

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

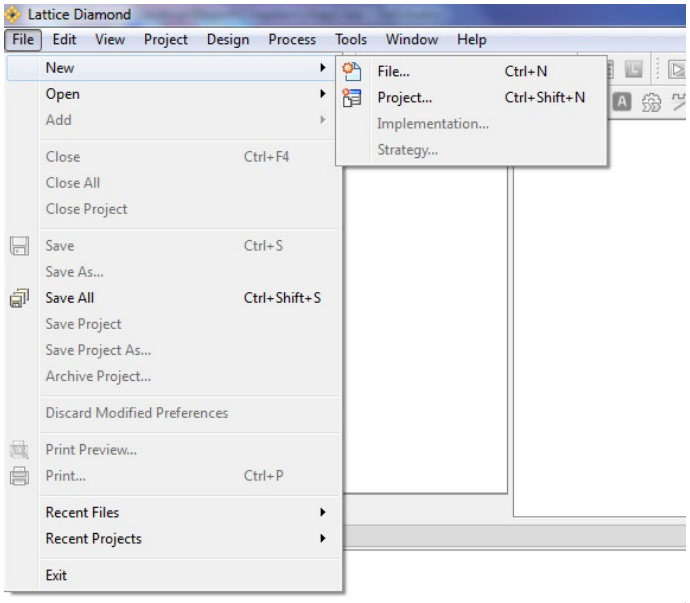
9.5.1 How to import your own project

By reading first Chapter 5 and then the example listed in 9.3.2, you should have understood how the CPU-FPGA communication system should work. You need the programming library for the FPGA composed by the three files included in the aforementioned example before, you need to set the GPIOs of the JTAG port of the FPGA as indicated by the piece of code listed there, and you certainly need a call to the FPGA programming function in the `main()`. What is to be substituted are the two huge byte arrays present in the file "TEST_FPGA.h" with the bitstream containing the information about the pin interface and for programming internally the FPGA through the JTAG. Such file is generated automatically from your own HDL description of the FPGA by the Lattice Diamond® Deployment Tool after the synthesis steps. What you should do is nothing else than replace within the file these two arrays with the ones generated by this tool, but this will be explained in detail the following.

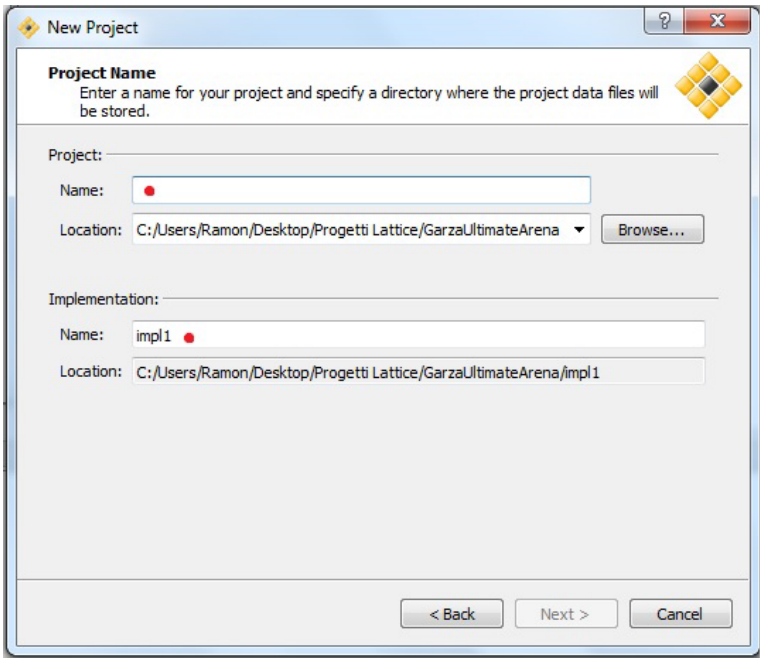
9.5.2 How to create a Lattice Project

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.

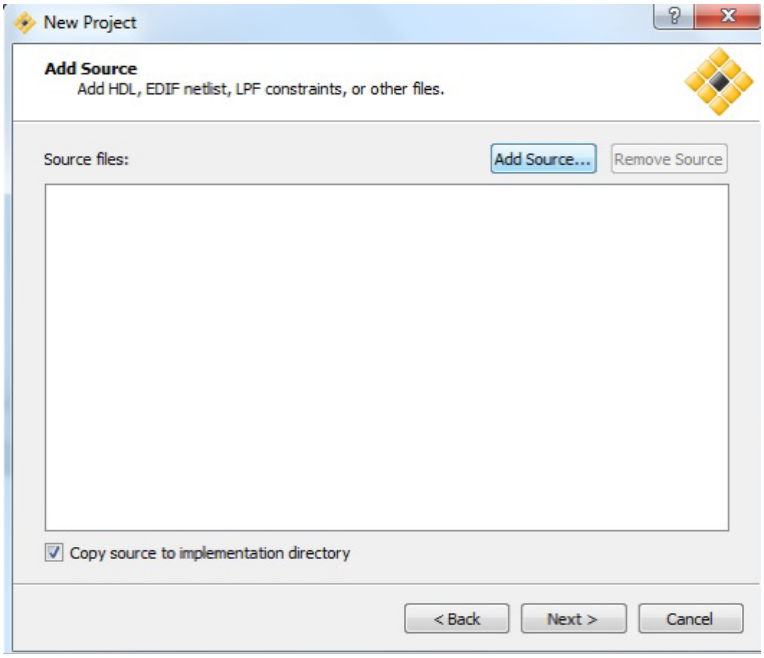
1. Create a new project



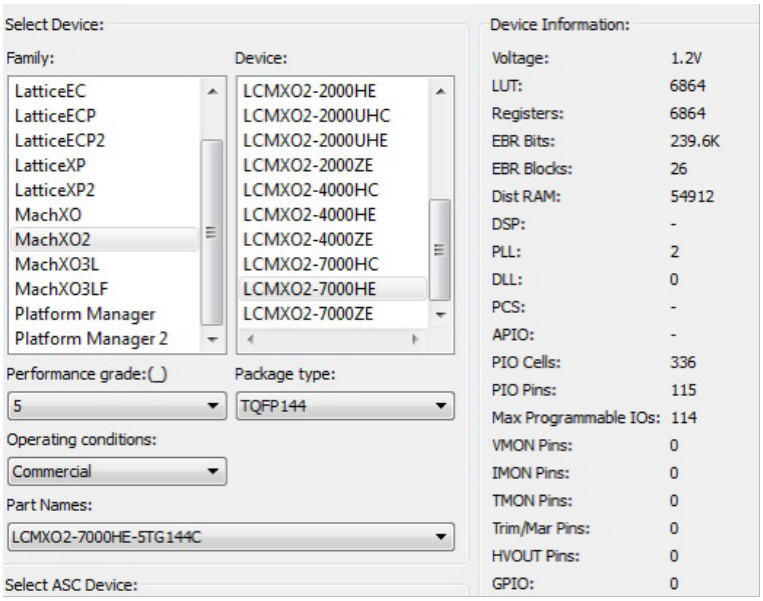
2. Browse to the location of your HDL project folder and insert an implementation name



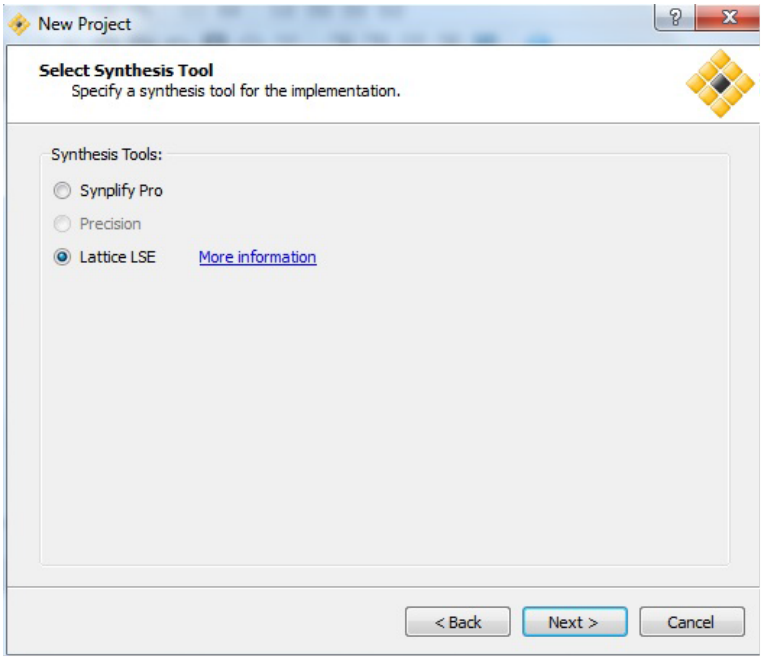
3. Add VHDL source files in this step or inside the project by right clicking on the input files folder



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

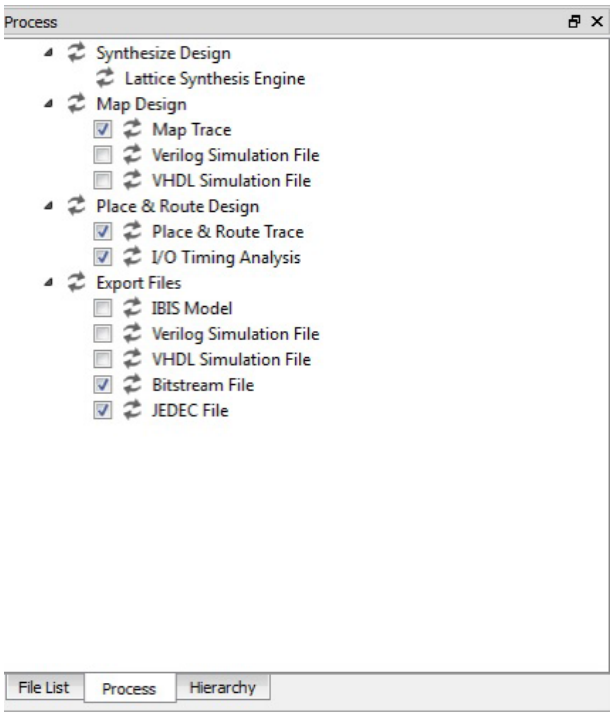
```
LOCATE COMP "<name_of_the_port[bit_number]>" SITE "<pin_name
    >"
```

   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

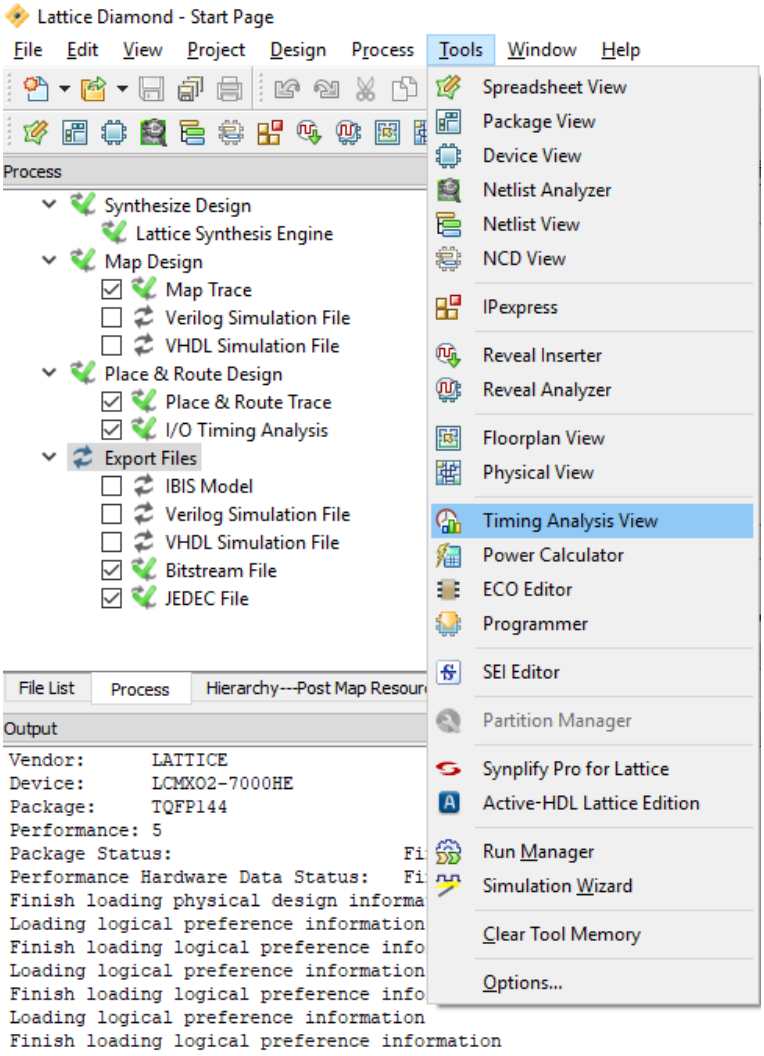
9.5.3 Synthesis Procedure

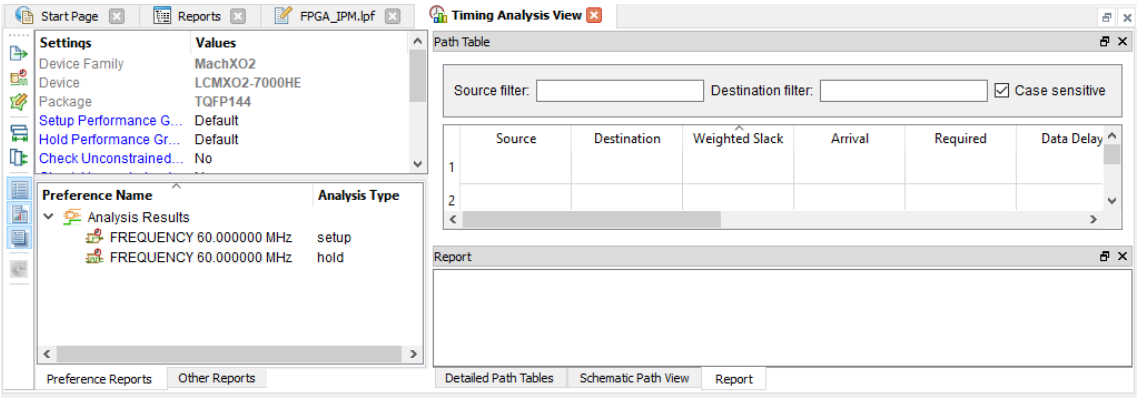
1. Go to «Process» tab of the Process Window and select the check marks as in the following.



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

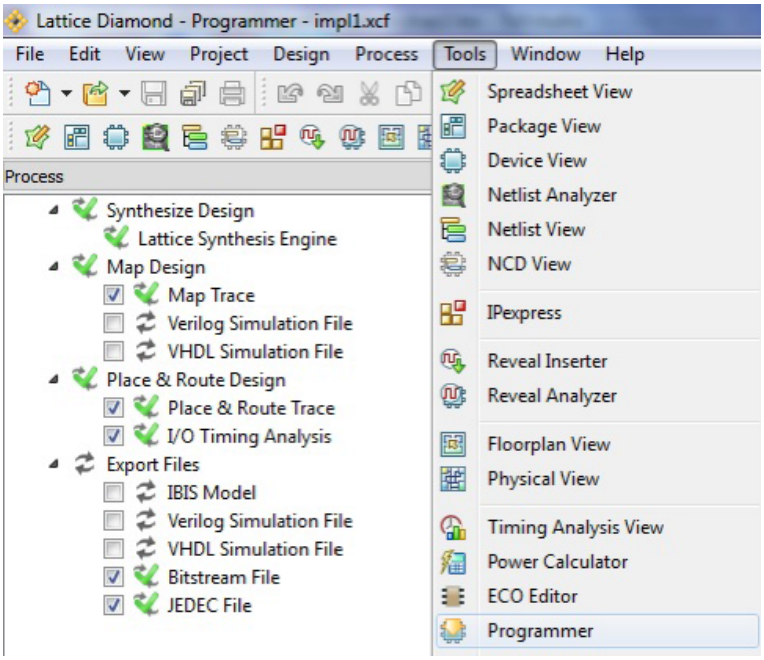
3. At the end of all synthesis steps, go to «Tools » Timing Analysis View» and check if there are no violation for setup and hold times



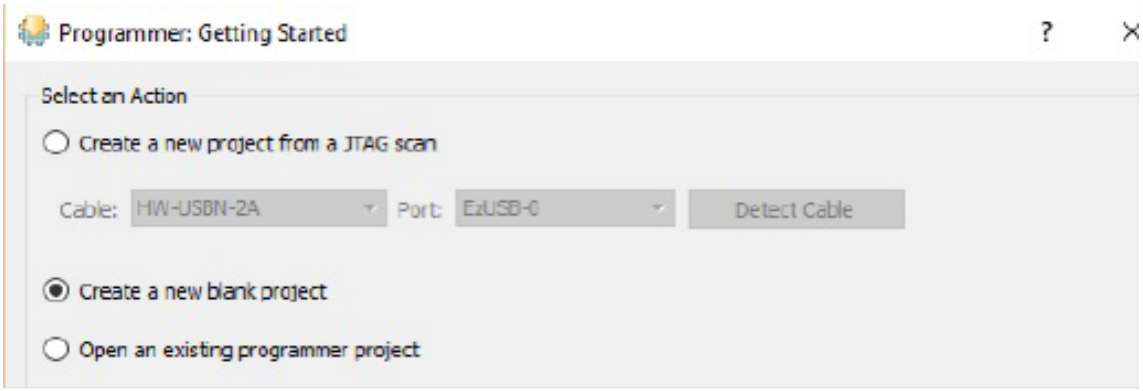


If times are respected, constraints are written in black. Otherwise, they are written in red. By clicking on the constraint, on the right it is possible to see details about the violating path

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

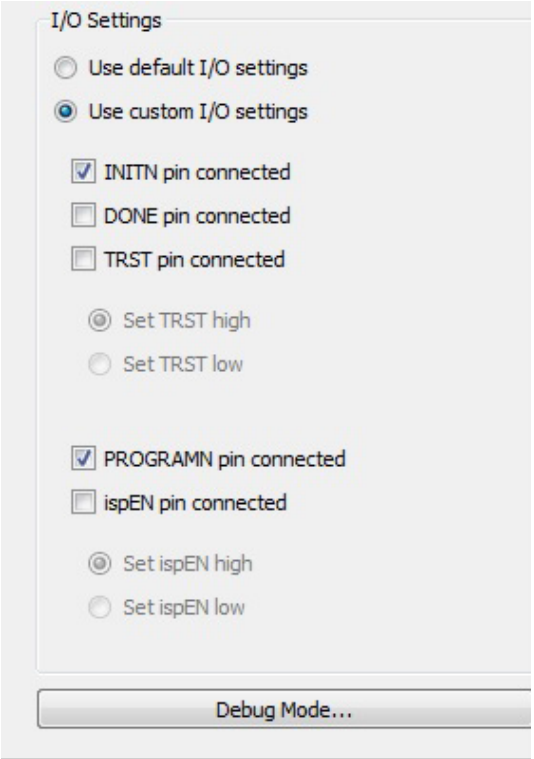


5. Select “Create a new blank project” on the appearing window



6. Verify the device family and the XCF file name

7. Check “I/O Settings” as in figure

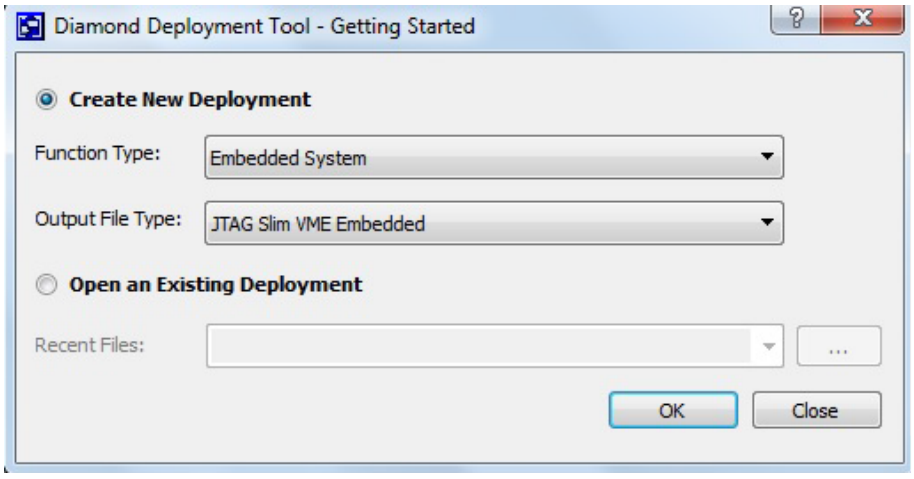


8. Save the changes applied. This file is necessary for Deployment Tool

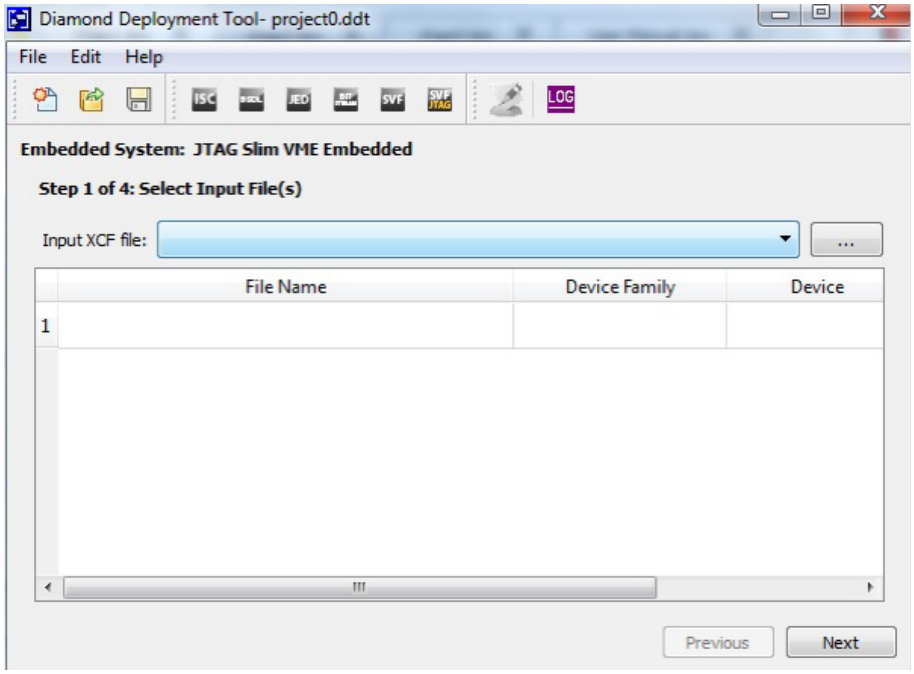
9.5.4 Deployment Tool usage

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

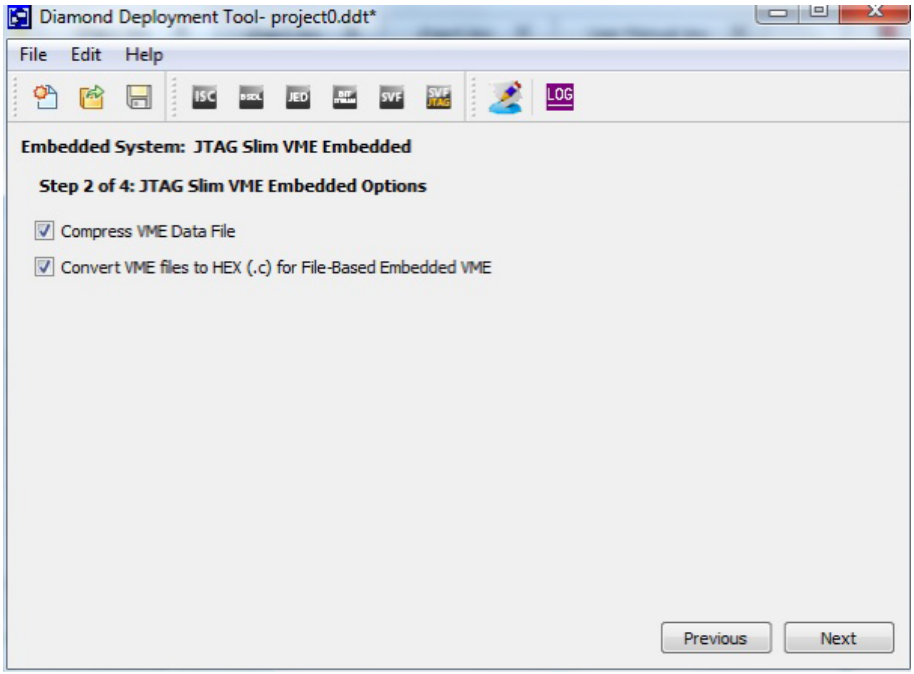
1. First, open the Deployment Tool and create a new project like in the following



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. Import the project as described in section 9.2.1

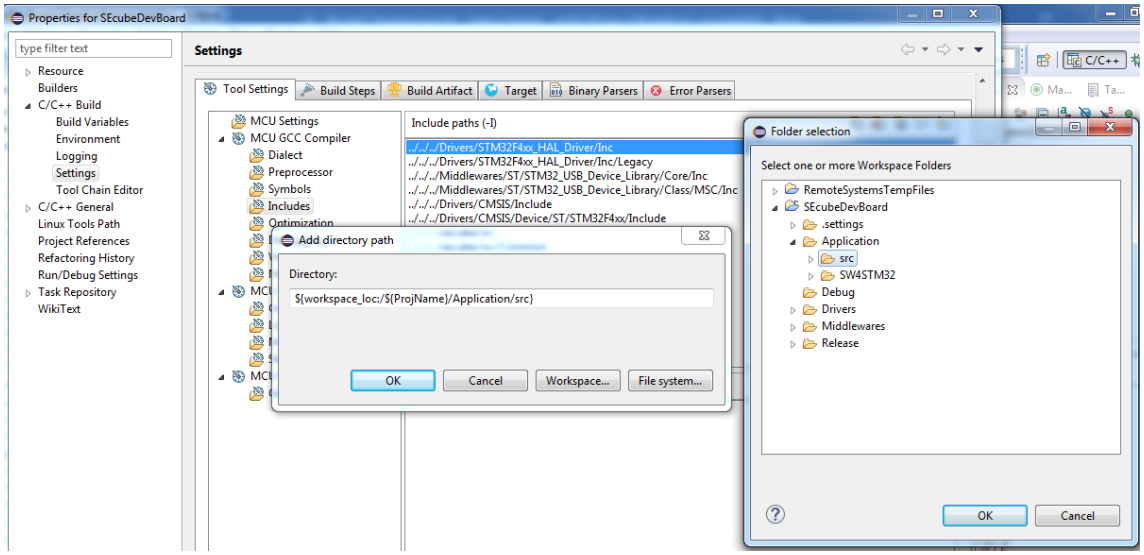
2. Import the necessary «File » Import…», select “Filesystem” and press “Next”

3. Browse to the directory where the SDK has been downloaded and then to the path “SEcube_SDK/Libraries/Examples/TestFPGA/”

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

5. 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

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



7. 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

8. 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

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

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

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

10. Now edit the code in “main.c” file including the header file “FPGA.h”
11. Also in “main.c” add a call to `B5_FPGA_Programming()` function
12. Save the changes to all files and build the project
13. Connect the DevKit as described in previous section
14. Run the project by right-clicking on it in the Project Explorer and selecting the Release binary under «Target » Program Chip» (i.e., select the label containing the string “/Release”)

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

APPENDIX A - SE*cube*™ Data Sheet

SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

General Description

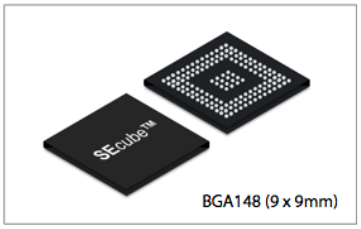
The SE*cube*™ (Secure Environment cube) is a powerful chip which integrates three key security elements in a single package. A fast floating-point Cortex-M4 CPU, a high-performance FPGA and an EAL5+ certified Security Controller (Smart Card).

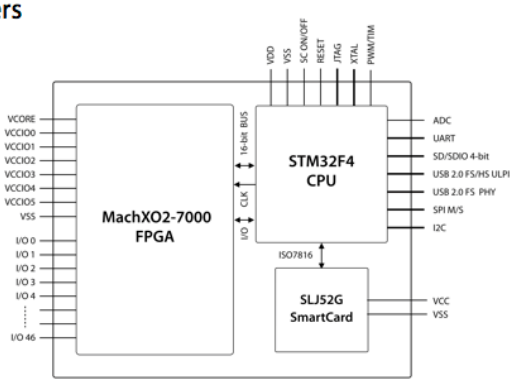
The result of this innovative combination gives an extremely versatile secure environment in a single SoC, in which developers can rapidly implement complex applications and appliances.

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.

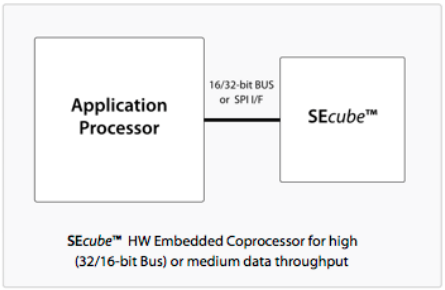
The SE*cube*™ is the ultimate solution for high-end design, delivering integration of a flexible, configurable and certified secure element.



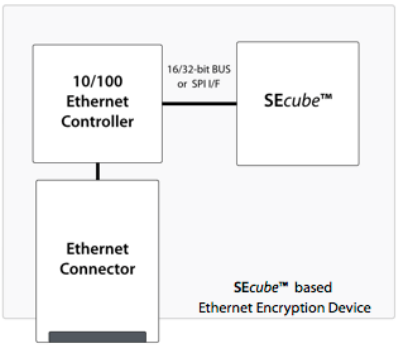


SE*cube*™ Block Diagram

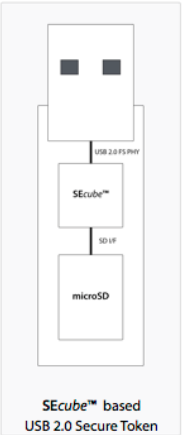
TYPICAL APPLICATION DIAGRAMS



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

SEcube™ is a Blu5 trademark. All other brand or product names are trademarks or registered trademarks of their respective holders.
The specifications and information herein are subject to change without notice.

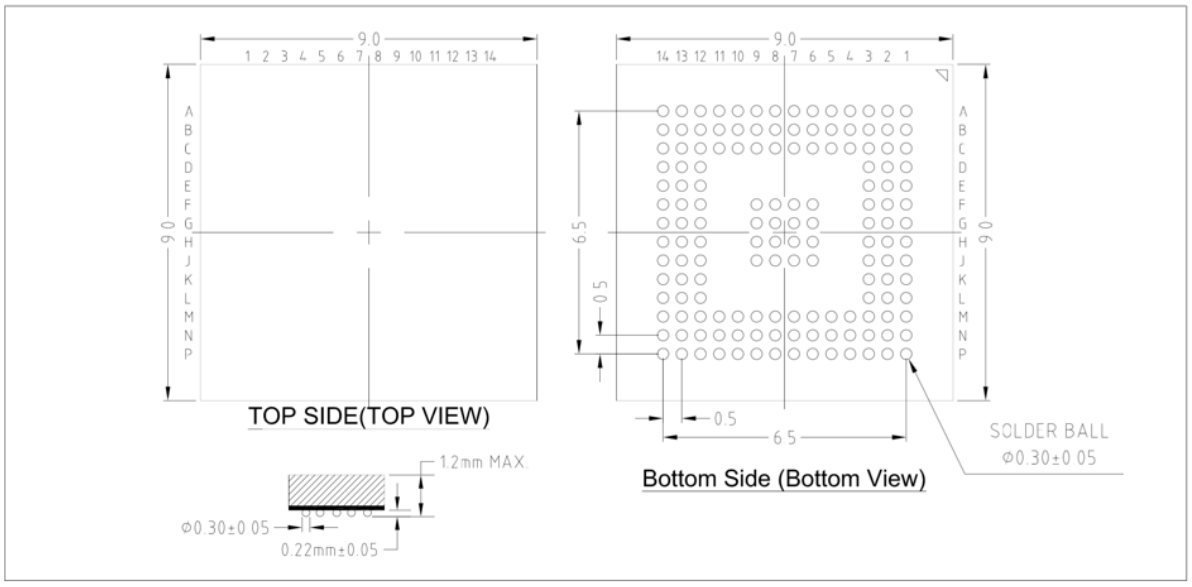
SE*cube*™ Data Sheet Introduction

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

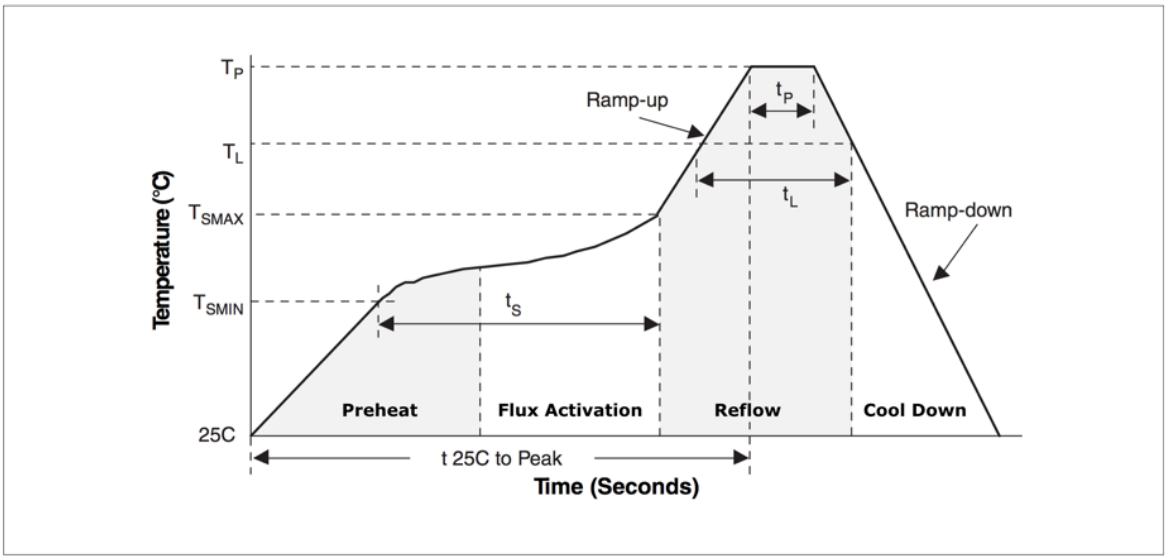
SE*cube*™ Data Sheet Introduction

August 2015 Data Sheet DUI 15082DS

Reflow Profiles

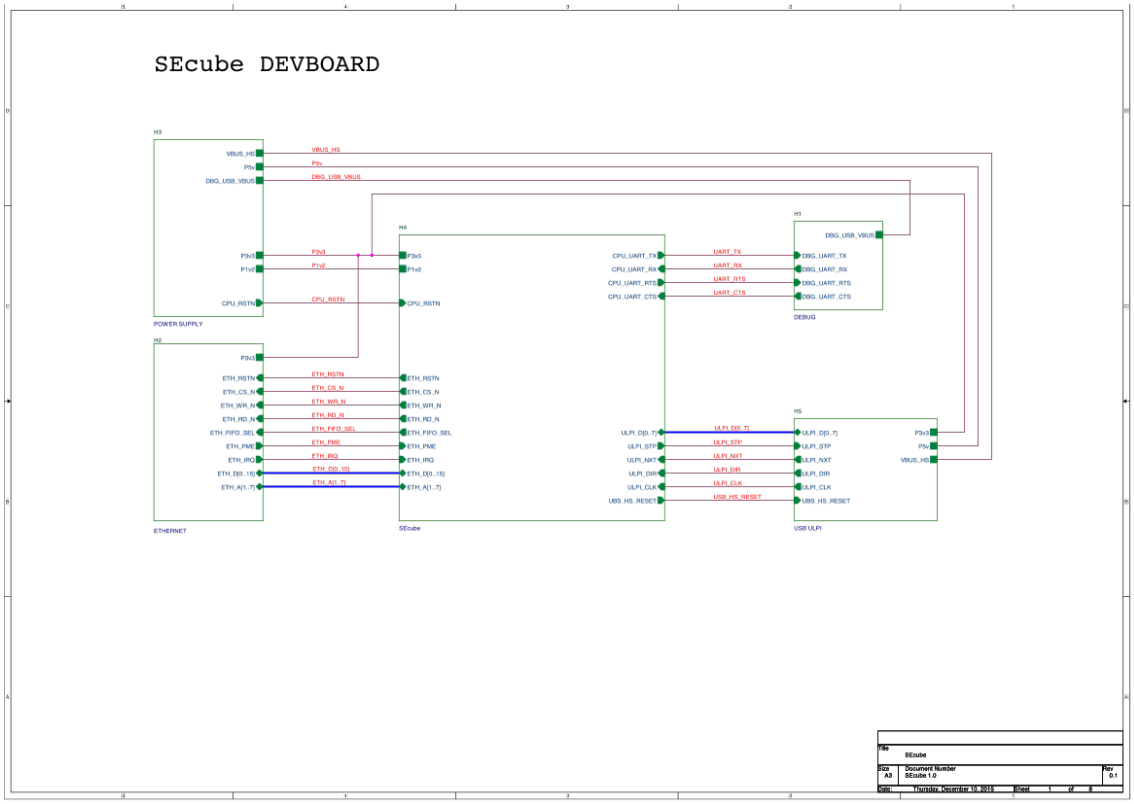
SE*cube*™ Reflow Table

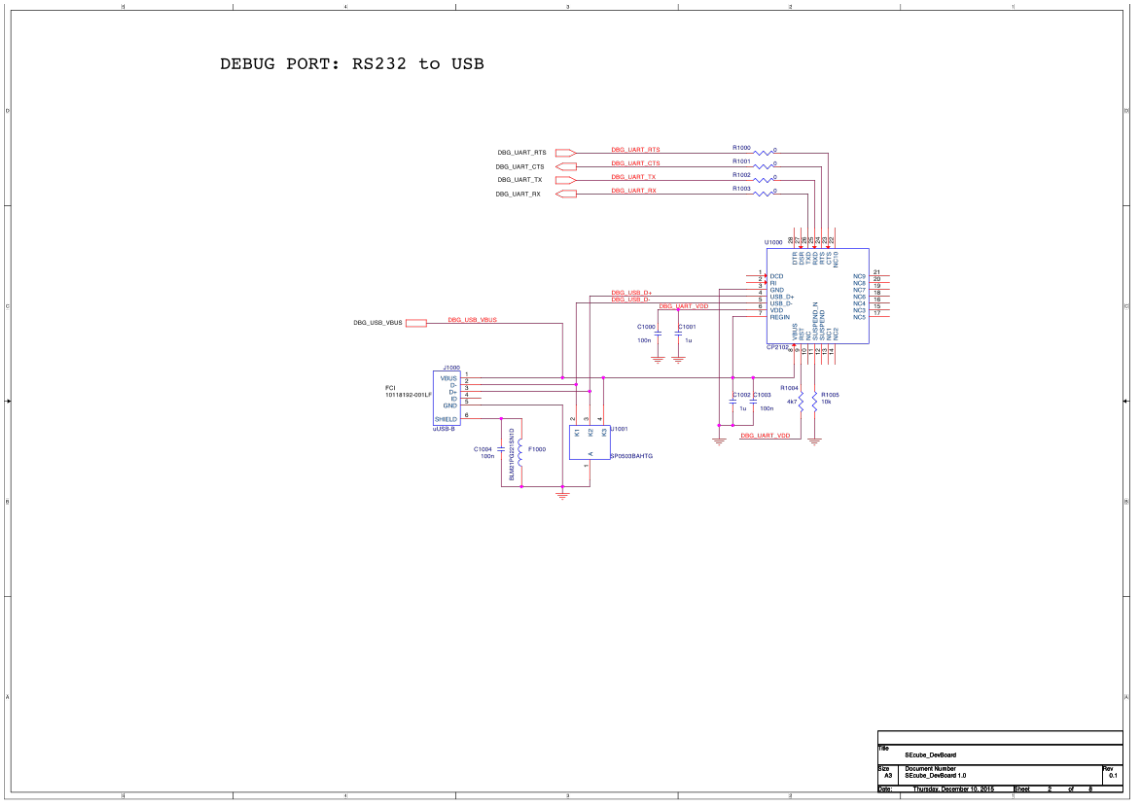
Parameter	Description	Pb-Free and Halogen-Free Packages
Ramp-Up	Average Ramp-Up Rate (T_{SMAX} to T_P)	3 °C/second max.
T_{SMIN}	Preheat Peak Min. Temperature	150 °C
T_{SMAX}	Preheat Peak Max. Temperature	200 °C
t_S	Time between T_{SMIN} and T_{SMAX}	60 seconds–120 seconds
T_L	Solder Melting Point	217 °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.

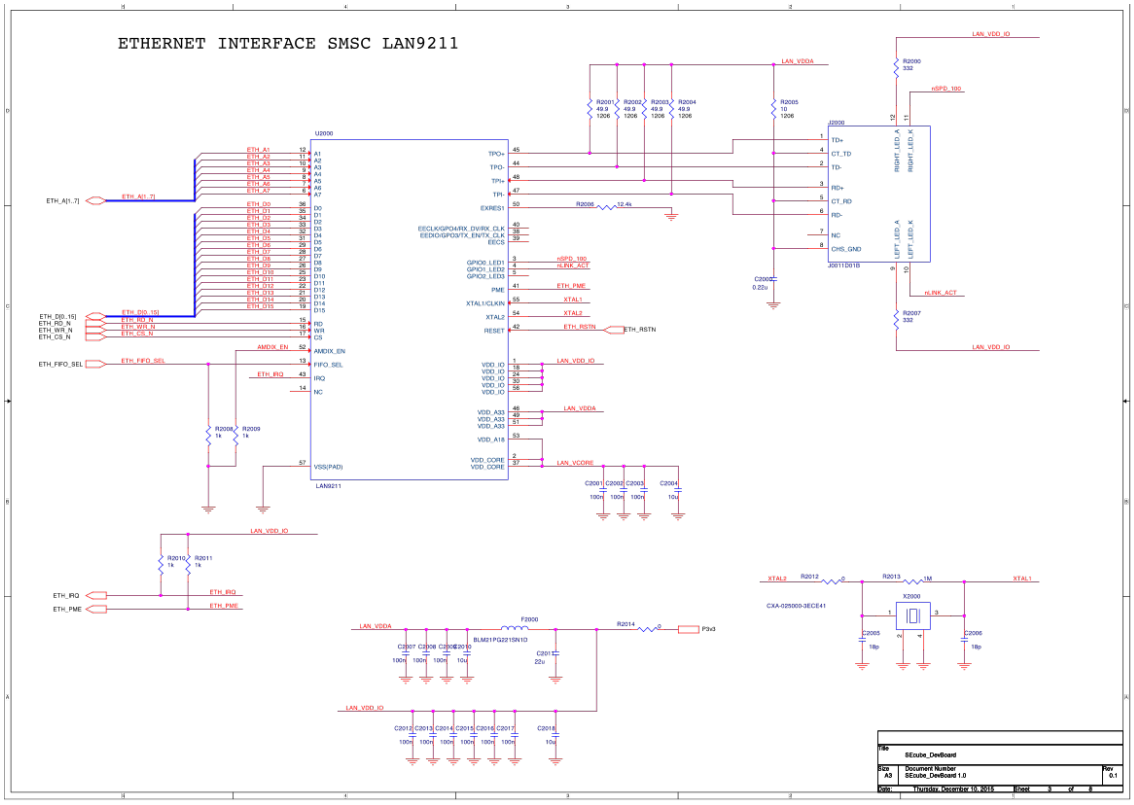


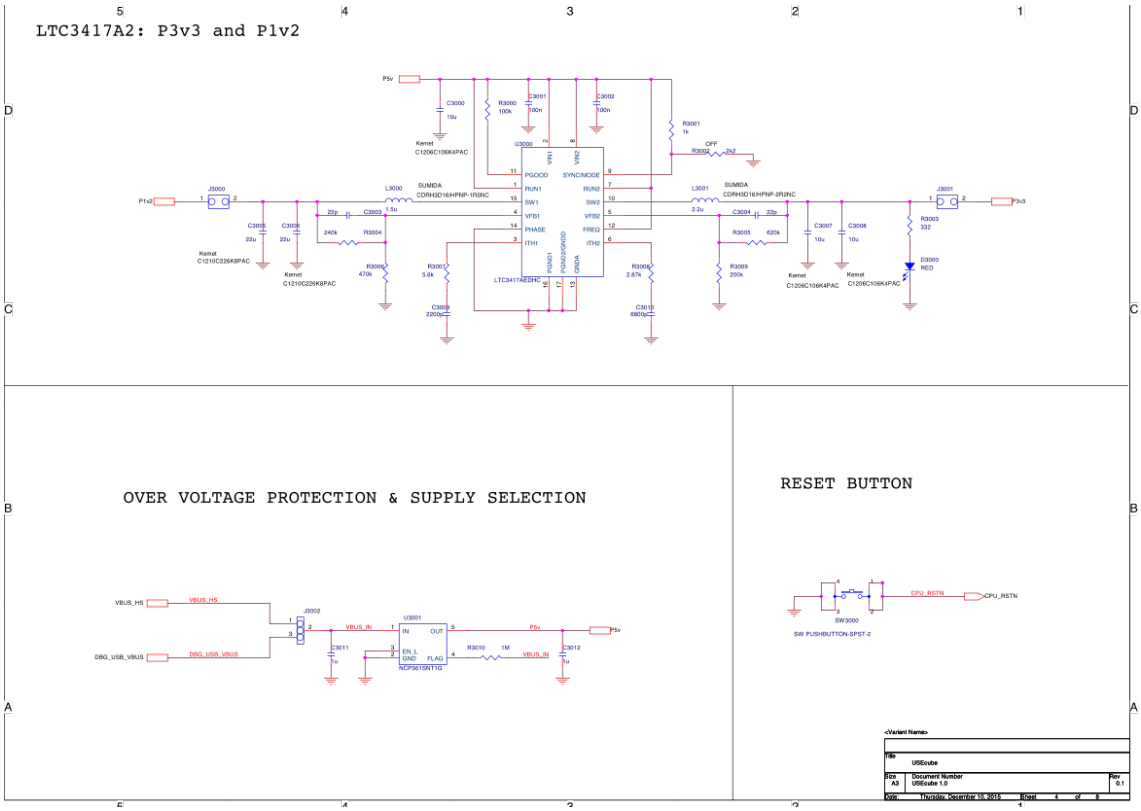
SE*cube*™ Termal Reflow Profile

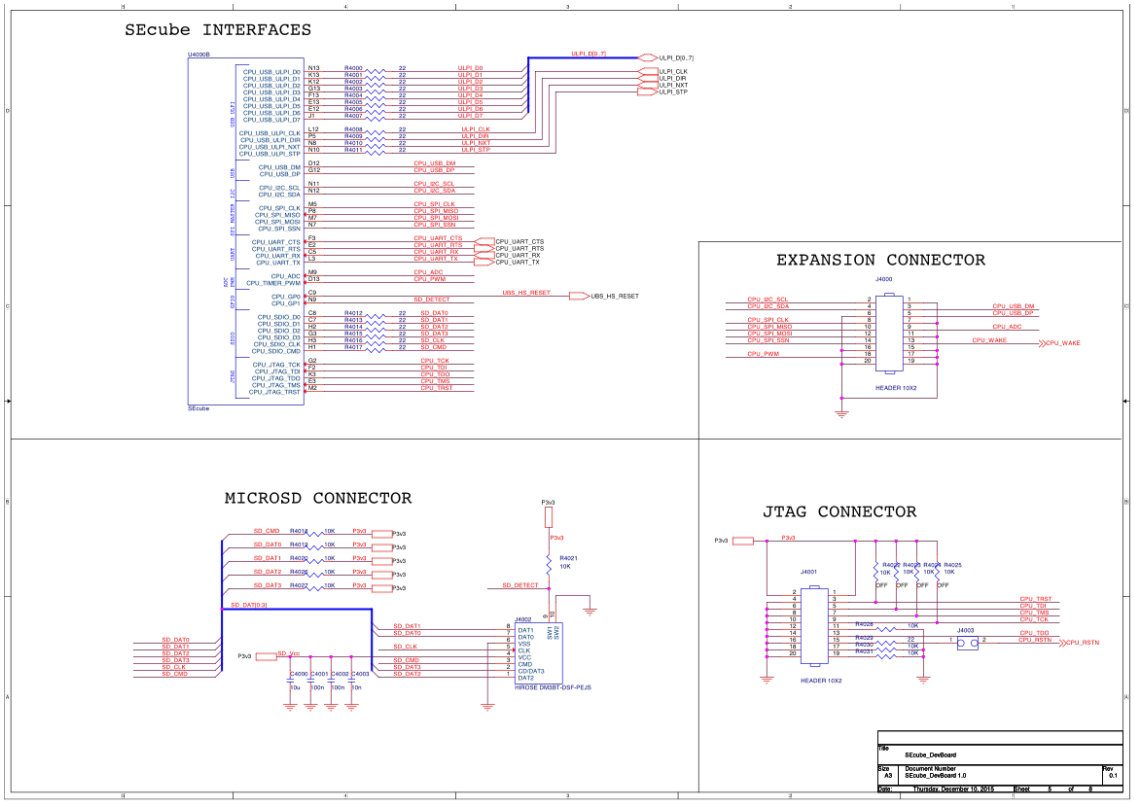
APPENDIX B - SE*cube*™ DevKit Schematics

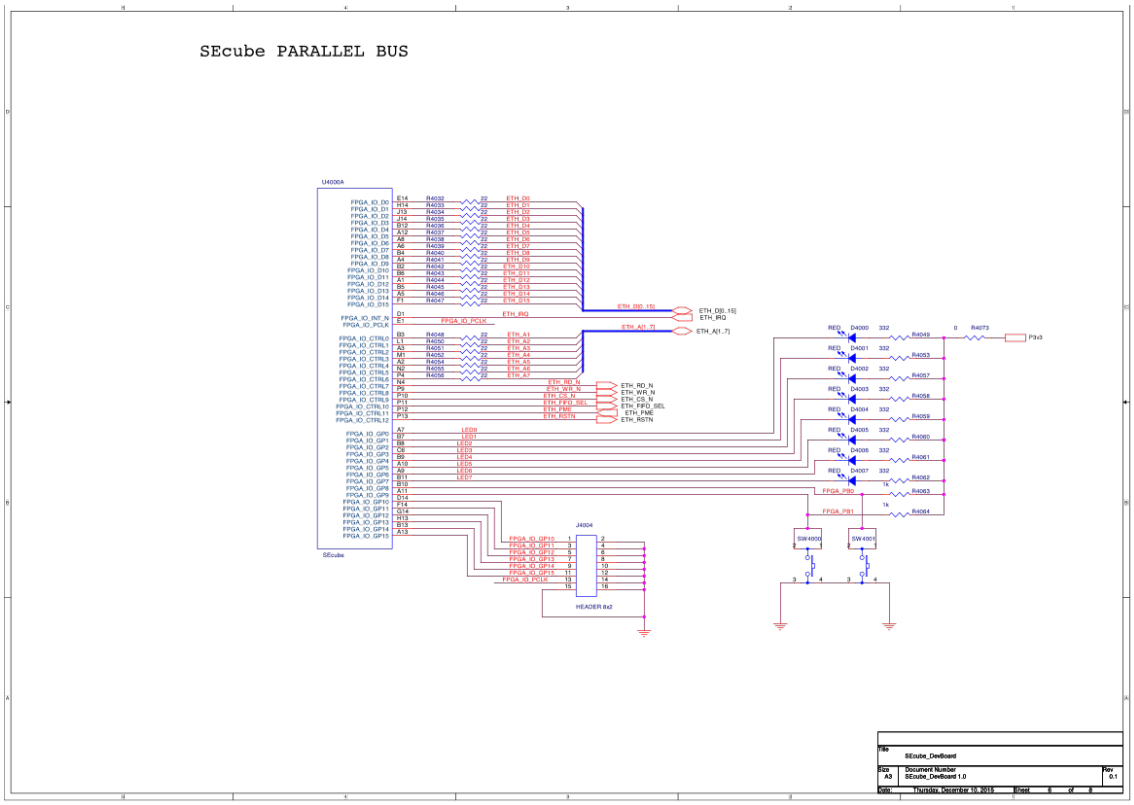


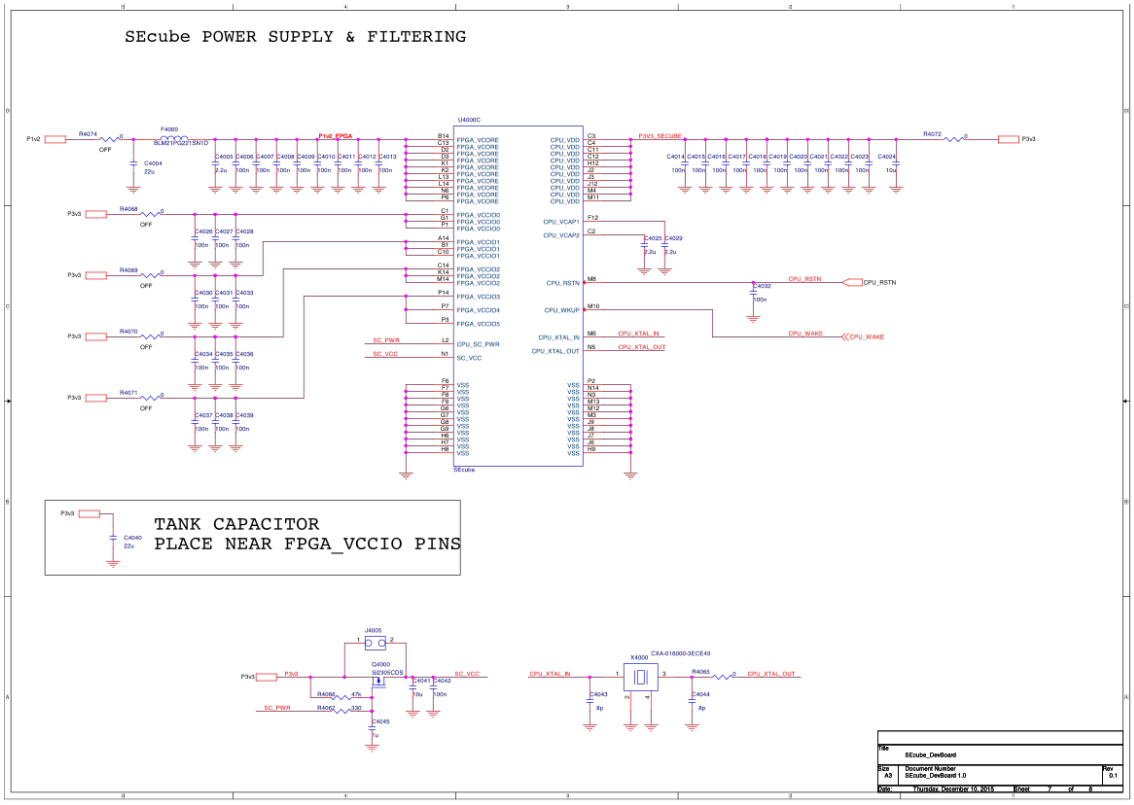


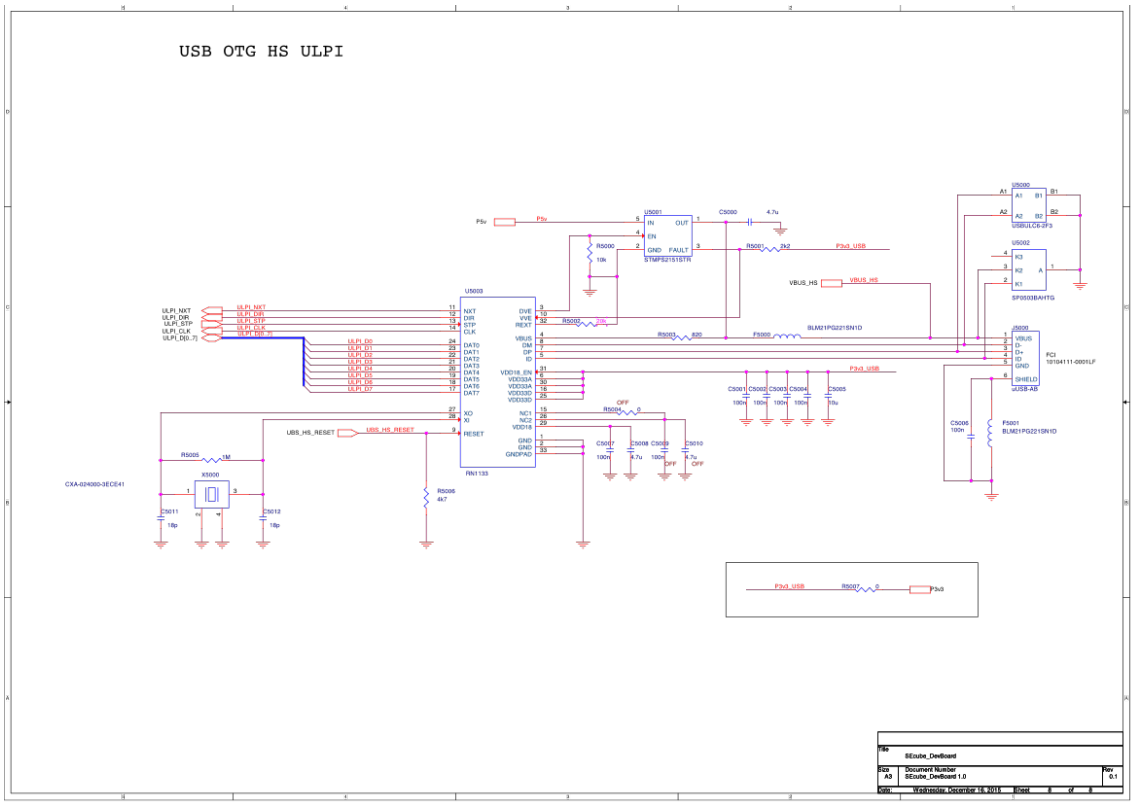

ETHERNET INTERFACE SMSC LAN9211
LAN9211
SEcube_DevBoard
SEcube_DevBoard 1.0


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

APPENDIX C - "main.c" for the HelloWorld application

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>

#include "secube-host/L1.h"

static uint8_t pin_admin[32] = {
  'a','d','m','i','n', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};

int main() {
  se3_disco_it it;
  se3_device dev;
  se3_session s;
  uint16_t return_value = 0;
  bool logged_in = false;

  printf("Welcome to the SEcube Hello World application!");
  printf("\n\n\n");
  Sleep(1000);

  printf("Looking for SEcube devices...\n");
  Sleep(3000);

  L0_discover_init(&it);
  if (L0_discover_next(&it)) {
    printf("SEcube device found!");
    return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT)
       ;
  }
  if (return_value != SE3_OK) {
    printf("Error opening device.\nPlease check board connection
       .\n\n");
    Sleep(3000);
    return(1);
  }
  else {
    printf("Open Device success\n");
  }

  /* Log in */
  printf("Logging in as admin...\n");
  Sleep(3000);
  return_value = L1_login(&s, &dev, pin_admin, SE3_ACCESS_ADMIN)
     ;
```

```
  if (return_value != SE3_OK) {
    printf("Error, login failed.\nPlease check security pin.\n\n
       ");
    Sleep(3000);
    return(2);
  }
  else {
    printf("Login success\n");
    logged_in = true;
  }
  /* */


  if (logged_in){
    printf("You are logged in SEcube device!");
  }

  /* Log out */
  printf("Logging out...\n");
  Sleep(3000);
  return_value = L1_logout(&s);
  if (return_value != SE3_OK) {
    printf("Error, logout failed.\nPlease check board connection
       .\n\n");
    Sleep(3000);
    return(3);
  }
  else {
    printf("Logout success\n");
    printf("\n\nConnection to SEcube device was successful\n");
    printf("Press [ENTER] to finish\n");
  }
  /* */

  getchar();
  return (0);
}
```