AN12133

A71CH Host software package documentation

Rev. 1.3— 23 April 2018
464313

Application note
COMPANY PUBLIC

Document information

Info	Content
Keywords	Security IC, A71CH, TLS, A71CH Host API
Abstract	This document provides a detailed view of application examples provided in the A71CH Host software package.

Revision history

Rev	Date	Description
1.0	20180219	Initial version
1.1	20180302	Updated section 5.4.5 ‘Transport Layer Security Handshake protocol’. Updated figure 4.
1.2	20180308	Modified figures 1 and 4
1.3	20180423	Modified chapter 4, chapter 5 and section 5.3.2 (new HostLib version)

Contact information

For more information, please visit: http://www.nxp.com

1. Introduction

This document provides an overview to the A71CH Host software architecture. It details the support documentation and the A71CH Host software directory structure. It also describes the A71CH Configure tool, the Host API usage and OpenSSL engine application examples included as part of the package.

2. A71CH overview

The A71CH is a ready-to-use solution enabling ease-of-use security for IoT device makers. It is a secure element capable of securely storing and provisioning credentials, securely connecting IoT devices to public or private clouds and performing cryptographic device authentication.

The A71CH solution provides basic security measures protecting the IC against many physical and logical attacks. It can be integrated in various host platforms and operating systems to secure a broad range of applications. In addition, it is complemented by a comprehensive product support package, offering easy design-in with plug & play host application code, easy-to-use development kits, documentation and IC samples for product evaluation.

3. A71CH Host Library overview

The A71CH Host Library translates function calls into APDUs that are transferred through an I^2C interface to the A71CH security IC. The A71CH executes the different APDUs and gives back the results to the Host Library through the same interface. The complete set of A71CH Host Library functions can be called from communication stacks like TLS or an application running on the host. Therefore, the A71CH Host Library behaves as the interface between a host microcontroller application and the A71CH security IC. Fig 1 depicts the A71CH Host Library architecture.

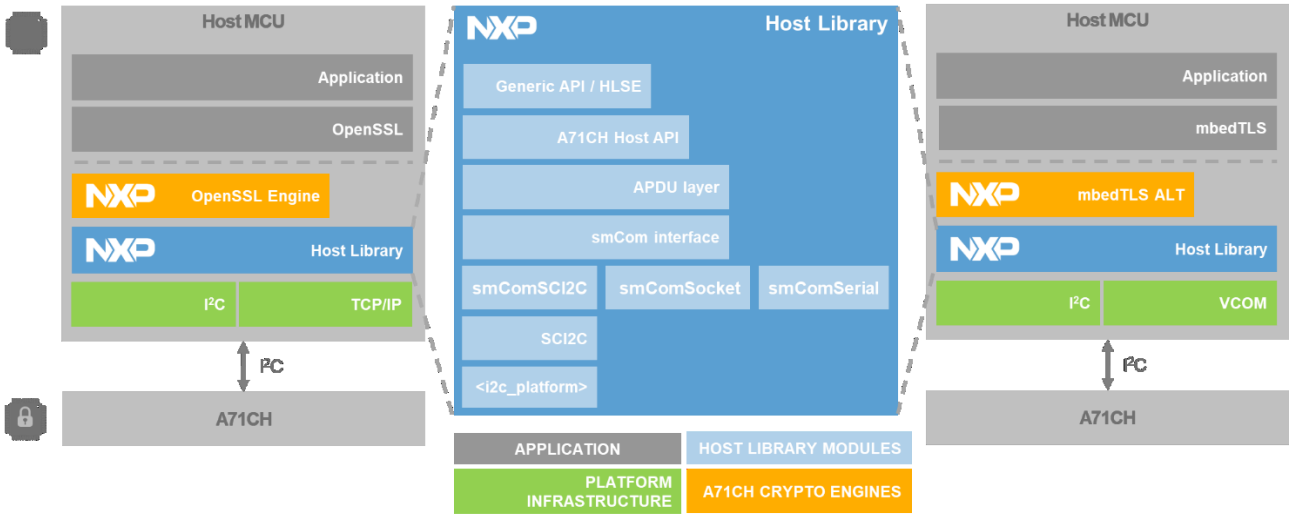

Fig 1. A71CH Host Library architecture

From top to bottom, the layers that compose the A71CH Host Library are:

- **Generic API / HLSE**: An API designed to be generic for secure elements (SE). It isolates an application from the details of the cryptographic device such that it does not have to change to interface to a different type of cryptographic device. The HLSE intends to abstract both the different APDU specification of the applets and the file system details (i.e. how each “object” is stored on the secure element).
- **A71CH Host API**: It is the implementation of the API dealing with A71CH security IC specific functionality. These source files implement the core functionality of the Host Library and provide a C interface abstracting the underlying APDU exchange mechanism between the Host MCU and the A71CH security module.
- **APDU layer:** It is the layer in charge of translating the A71CH Host API function calls to the APDU commands that are delivered to the A71CH via the host interface.
- **smCom interface, smComSocket, smComSerial and smComSCI2C**: The implementation of the API dealing with the data link and setting up the communication layer between the host and the secure element. Additionally, TCP/IP connection can be established between the Host device and a different platform. For instance, this is achieved by using the smComSocket communication layer and the RJCT server also available in the A71CH Host software package.
- **SCI2C and i2c_platform**: Implementation of the Smart Card I²C (SCI2C) protocol using an I²C based physical interface between the Host MCU and the A71CH security IC [SCI2C].

4. A71CH Host software package directory structure

The A71CH Host software package [A71CH_HOST_SW] is structured in 13 folders: **demo, doc, ext, frdmk64f_projects, hostLib, inf, linux, tools, win32** and **vs201x_projects** (2010, 2012, 2015 and 2017):

- The A71CH Host software documentation and release notes are stored inside the **doc** folder.
- The source code, header files and application examples of the A71CH Host Library are stored in **hostLib** folder.
- The **linux** folder contains compilation scripts and makefiles to build and compile the application examples on Linux environments.
- The **vs201x_projects** folder contains the Visual Studio 201x project with the A71CH application examples for Windows platforms.
- The **win32** folder contains the NXPCardServer folder redirected to **ext** folder, where the NXPCardServer source files are located.
- The **demo** folder contains an example to establish a connection to AWS IoT MQTT Platform (Amazon web services).
- **ext** folder contains source files and definitions for the different libraries used in the project, these libraries are, amongst others, NXPCardServer, OpenSSL, mbedTLS and FreeRTOS.
- Finally, a set of MCUXpresso example projects for the MCU Kinetis K64F are stored in **frdmk64f_projects.**

Fig 2 shows the A71CH Host software package directory structure. In this chapter, the most relevant folders are explained in detail.

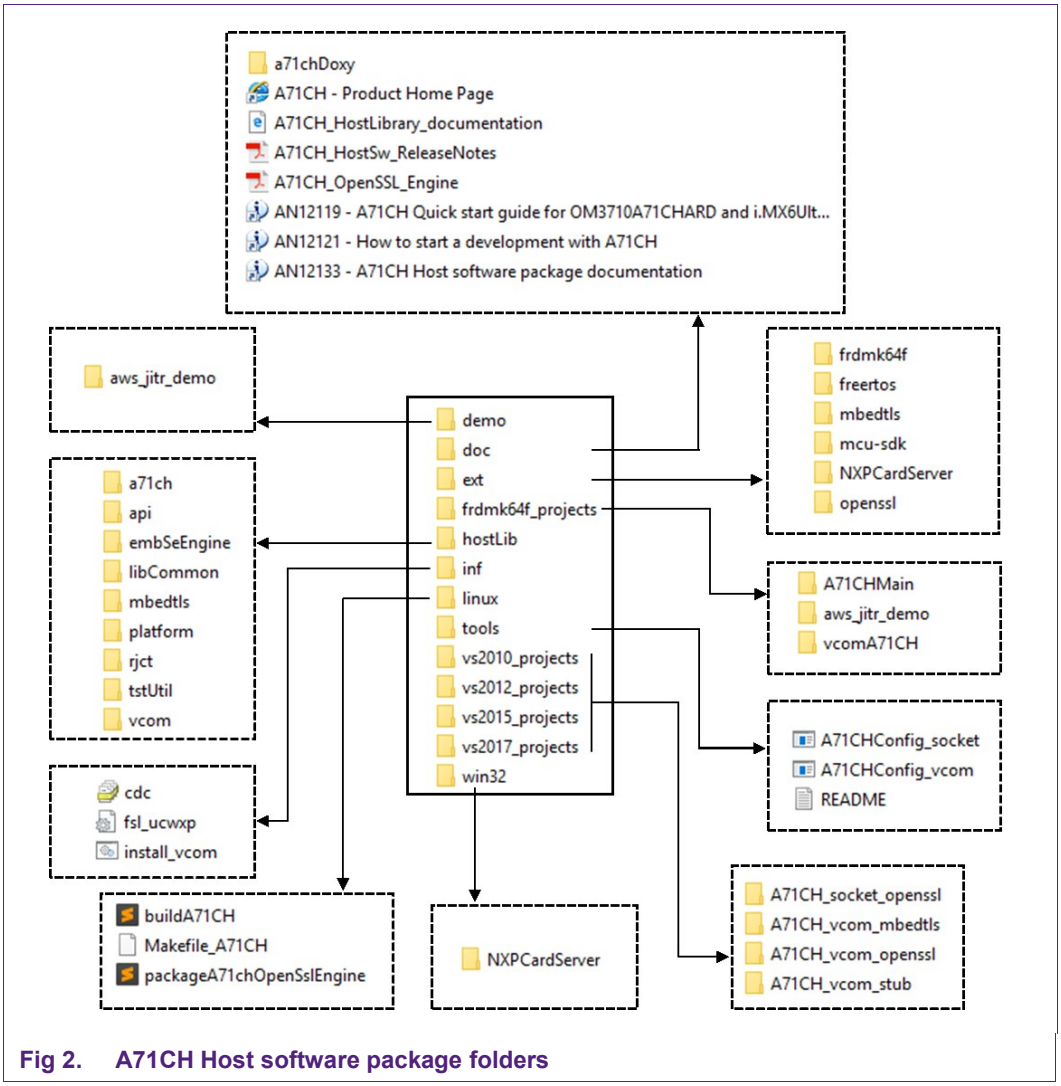

Fig 2. A71CH Host software package folders

4.1 Doc folder

The Doc folder contains a comprehensive hyperlinked documentation (html-files) created by Doxygen. The entry point to open it with a browser is the following:

doc/A71CH_HostLibrary_documentation.html

The A71CH Doxygen documentation provides detailed information about the A71CH API, about A71CH application examples (i.e. A71CH Configure tool, A71CH API usage examples, A71CH OpenSSL Engine and mbed TLS examples) and instructions to execute A71CH application examples in Linux, Windows and FRDM-K64F. Fig 3 shows a screenshot of the landing page of the A71CH Doxygen documentation.

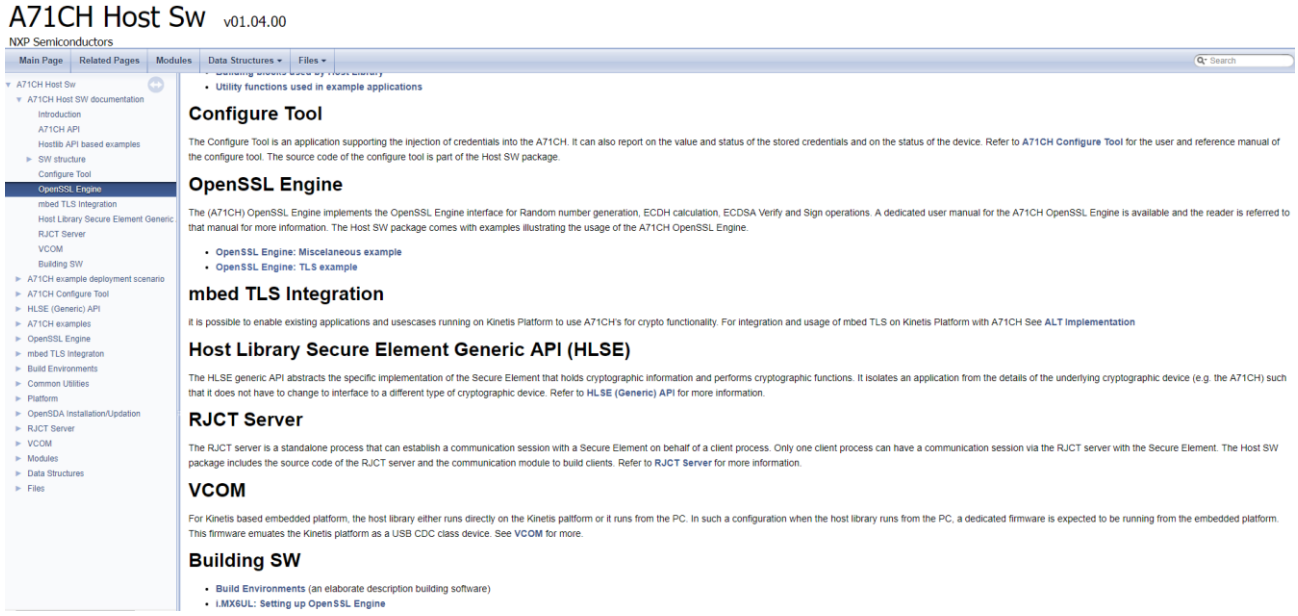

Fig 3. A71CH Host software Doxygen support documentation

4.2 HostLib folder

The **HostLib** folder is structured as follows:

- **HostLib/a71ch**: Source code and header files of the A71CH Host API and Generic API or HLSE API, the A71CH Configure Tool and the A71CH API Usage examples.
- **HostLib/api**: Source code and header files of the A71CH Host Library.
- **HostLib/embSeEngine**: Source code and header files of the A71CH OpenSSL Engine and OpenSSL configuration file.
- **HostLib/libCommon:** Source code and header files of communication and infrastructure functionality such as the SCI2C and smComm drivers.
- **HostLib/mbedtls:** Source code and header files for mbed TLS as well as examples.
- **HostLib/platform**: Specific libraries for each one of the supported platforms (generic, i.MX, Kinetis, Linux).
- **HostLib/rjct**: Remote JC Terminal server source code and header files
- **HostLib/tstUtil**: Source code and header files covering test infrastructure used by the example project.
- **HostLib/vcom**: Virtual COM port implementation for Kinetis.

4.3 Ext folder

The **ext** folder contains different type of external libraries and modules used in the project. These are the source files and headers for the Kinetis K64F MCU as well as the FreeRTOS implementation. Another two important subfolders of **ext** are the implementations of mbedTLS and OpenSSL. These libraries are extremely important as

they are in charge of implementing the SSL/TLS protocol for secure connection. Also, the OpenSSL command line tool is included.

The **ext** folder also contains the NXPCardServer folder that allows a Windows application built on top of the A71CH Host software package to communicate with an A71CH connected via an I²C bird to the USB port of the PC (Fig 4).

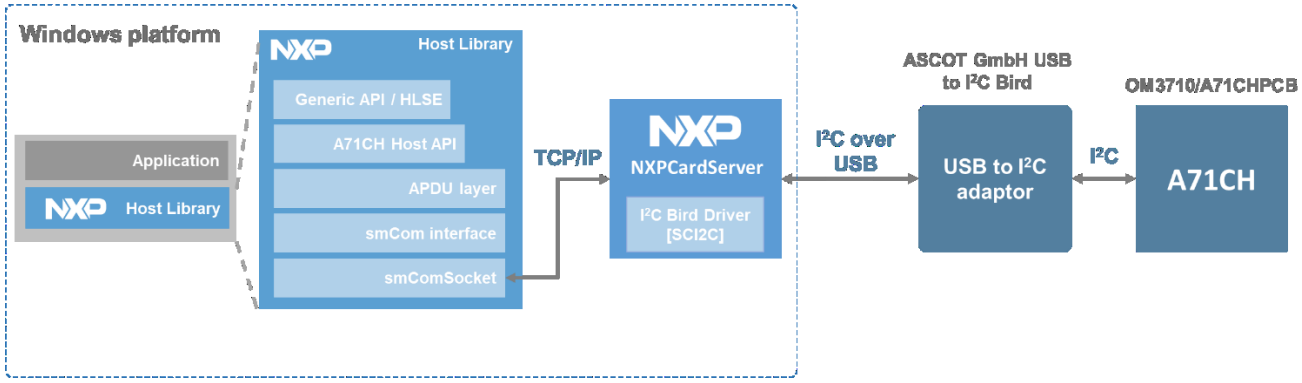

Fig 4. NXPCardServer setup

The Windows software can alternatively connect to the RJCT server on e.g. an i.MX6UltraLite evaluation board to achieve communication of the Windows code with an A71CH security IC. It will be further explained in this document (5.3.1).

4.4 VS201x_projects and Linux folders

The **vs201x_projects** folder contains the Visual Studio projects of the A71CH Host API usage example and A71CH Configure tool. It also includes the pre-compiled OpenSSL header files and binaries used by Visual Studio projects, these are stored under **ext**. Please be aware that security patches are regularly released for OpenSSL; thus, the binaries provided do not contain the latest security relevant patches and shall be updated by the user for the up-to-date development environments and IDE solutions. There are 4 different folders indicating the target Visual Studio version; 2010, 2012, 2015 and 2017. I.e., the 4 folders contain the same projects but each one is prepared to work properly in the indicated Visual Studio version.

Similarly, the **Linux** folder contains a Makefile to build and compile both the A71CH API usage examples and A71CH Configure tool projects.

4.5 Tools folder

The **tools** folder contains precompiled versions of the A71CH Configure tool (for Windows): one for the socket interface and one for the vCOM interface.

4.6 Inf folder

The **inf** folder contains the Windows compatible driver for vCOM.

5. A71CH application examples

The A71CH Host software package contains three example applications:

- **A71CH Host API usage example**: An example showing the entire A71CH Host Library functionality, function by function. The Windows version of the project is contained in the *vs201x_projects* folder, the Linux Makefile script is contained in the *Linux* folder and the source code and header files are contained in *HostLib* folder.
- **A71CH Configure Tool**: A command line tool that supports a set of operations with the A71CH. For example, the injection of credentials, information about the stored key pairs and public keys, etc. Again, the Windows version is contained in the *vs201x_projects* folder, the Linux Makefile script is contained in the Linux folder and the source code and header files are stored in the *HostLib* folder.
- **A71CH OpenSSL Engine examples**: A set of examples demonstrating the OpenSSL functionalities added by the A71CH OpenSSL Engine.
- **mbed TLS examples**: A set of examples demonstrating the mbed TLS functionalities.

A detailed description of these application examples is provided in this chapter.

5.1 Host platforms and supported applications

The A71CH Host software package is supported in the following platforms:

- Microsoft Windows based platforms with Visual Studio.
- Linux distribution for NXP's i.MX6UltraLite (IMX6ULEVK board) and Kinetis K64F.

Table 1 illustrates the application support on the different Host platforms. The (*d*) indicates that the A71CH must have the Debug Mode available in order to run the examples as these first reset the IC to have a defined starting point. The I²C qualifier in the header of the table indicates that the application is only supported when connecting directly over I²C to the A71CH.

Table 1. Host Platforms and supported applications

	Host Library	Configure Tool	Host API usage	A71CH Engine (I²C)
Windows	Yes	Yes	Yes (*d*)	-
IMX6ULEVK	Yes	Yes	Yes (*d*)	OpenSSL (*d*)
Kinetis	Yes	Yes	Yes (*d*)	mbed TLS (*d*)

5.1.1 SCI2C on Host platform

The A71CH Host software package comes bundled with an SCI2C protocol implementation. The protocol is defined in [SCI2C]. To ensure compatibility with SCI2C, the host platform's I²C driver needs to support:

- Block read (i.e. the response length is encoded in the first byte of the response; a feature typically associated with SMBUS)
- Repeated start

5.2 A71CH Host API usage example applications

The A71CH Host API usage example applications is a sample project oriented to show the functionality of the A71CH Host library using both the A71CH Host API and the Generic API or HLSE. A set of use cases is sequentially executed to show the user each Host API function call, the APDU's that are sent to the A71CH and the received responses.

5.2.1 A71CH Host API usage examples

The A71CH Host API usage example is divided into the following use cases:

- **ex_aes.c**: It demonstrates wrapped setting and retrieving of symmetric keys. It also demonstrates crypto applications of stored symmetric keys.
- **ex_boot.c:** It demonstrates that the Host does not need to know the SCP03 Base Keys to establish an SCP03 session.
- **ex_config.c:** It demonstrates the storage and usage of configuration keys.
- **ex_ecc_nohc.c:** It demonstrates ECC crypto functionality of the A71CH without relying on Host Crypto module.
- **ex_gpstorage.c:** It demonstrates general purpose storage and monotonic counter functionality.
- **ex_misc.c:** It demonstrates miscellaneous module functionality: get the module information and unique identifier, obtain information about the stored credentials, fetch a random number, send a raw APDU command and calculate a SHA256.
- **ex_psk.c:** It demonstrates plain or ECDH enhanced pre-shared, key-based master key creation.
- **ex_scp.c:** It demonstrates how to set up an SCP03 channel between the Host and the A71CH.
- **ex_sst.c:** It demonstrates storage of symmetric and public keys.
- **ex_sst_kp.c:** It demonstrates invocation of ECC key pair protected storage specific functionality.
- **ex_boot.c:** It demonstrates the handover of SCP03 session keys from bootloader to OS.
- **ex_walkthrough.c:** It demonstrates the usage of the A71CH from a system integrator perspective.
- **ex_debug.c:** It demonstrates miscellaneous functionality supported by the Debug Mode of A71CH.

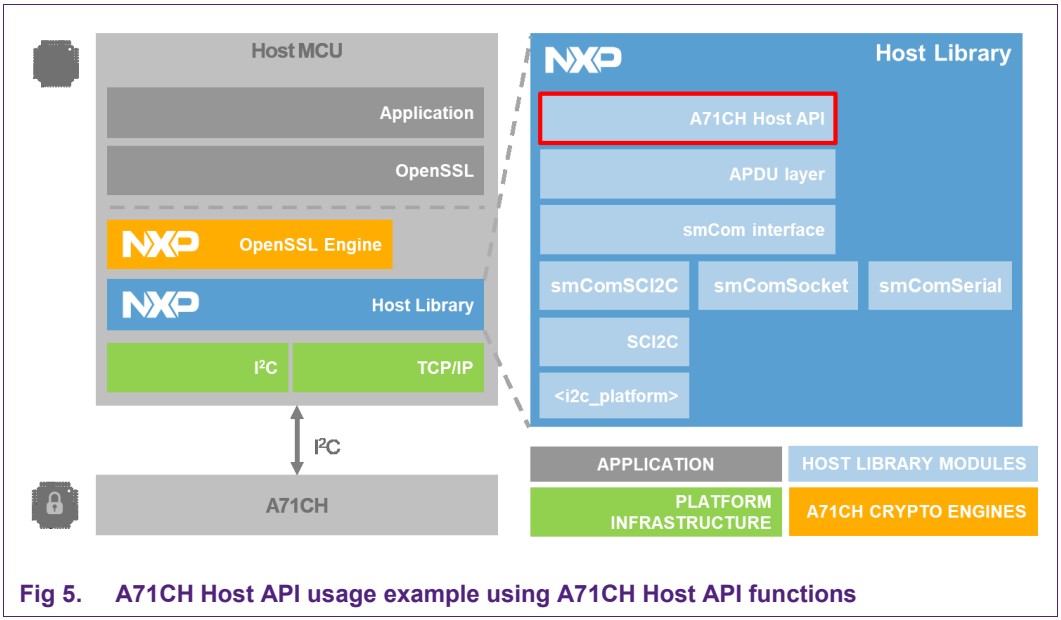

Fig 5. A71CH Host API usage example using A71CH Host API functions

5.2.2 Generic API or HLSE usage examples

The Generic API or HLSE usage examples offer similar functionality to the A71CH API usage examples but use the HLSE API instead of the A71CH API. This generic API or HLSE is designed to be generic for Secure Elements (SE). It isolates an application from the details of the cryptographic device such that it does not have to change to interface to a different type of cryptographic device.

- **ex_hlse_cert.c**: It demonstrates storage of certificates
- **ex_hlse_aes.c:** It demonstrates wrapped setting and retrieving of symmetric keys. It also demonstrates crypto applications of stored symmetric keys.
- **ex_hlse_boot.c:** It demonstrates that the Host does not need to know the SCP03 Base Keys to establish an SCP03 session.
- **ex_hlse_config.c:** It demonstrates the storage and usage of configuration keys.
- **ex_hlse_ecc_nohc.c:** It demonstrates ECC crypto functionality of the A71CH without relying on Host Crypto module.
- **ex_hlse_gpstorage.c:** It demonstrates general purpose storage and monotonic counter functionality.
- **ex_hlse_misc.c:** It demonstrates miscellaneous module functionality: get the module information and unique identifier, obtain information about the stored credentials, fetch a random number, send a raw APDU command and calculate a SHA256
- **ex_hlse_psk.c:** It demonstrates plain or ECDH enhanced pre-shared, key-based master key creation.
- **ex_hlse_scp.c:** It demonstrates how to set up an SCP03 channel between the Host and the A71CH.

- **ex_hlse_sst.c:** It demonstrates invocation of ECC key pair protected storage specific functionality.
- **ex_hlse_sst_kp.c:** It demonstrates invocation of ECC key pair protected storage specific functionality.
- **ex_hlse_walkthrough.c:** It demonstrates the usage of the A71CH from a system integrator perspective.
- **ex_hlse_debug.c:** It demonstrates miscellaneous functionality supported by the Debug Mode of A71CH.

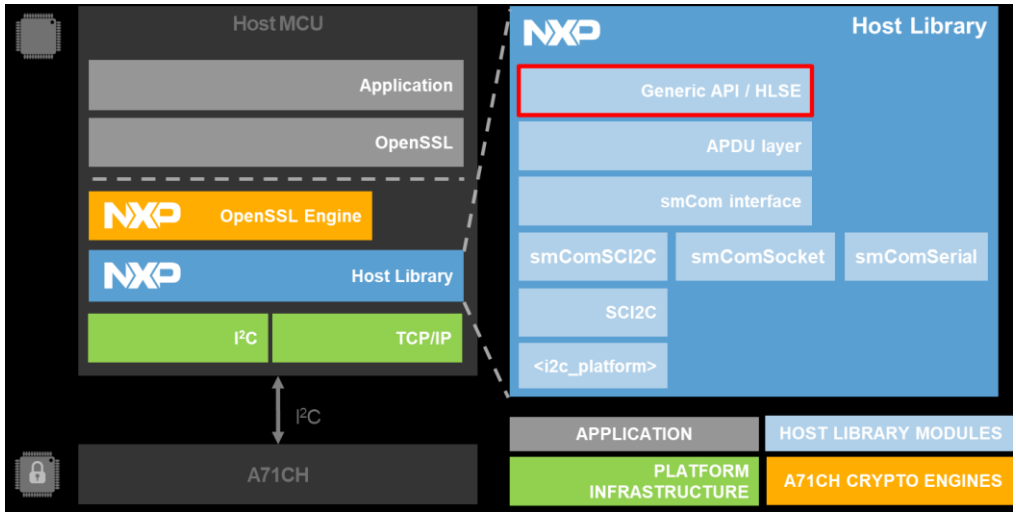

Fig 6. A71CH Host API usage example using HLSE API functions

More information about the A71CH Host API and HLSE usage example application and how to execute it can be found in A71CH Doxygen documentation.

5.3 A71CH Configure tool

The A71CH Configure Tool is an application supporting the injection of credentials into the A71CH. It can also report on the value and status of the stored credentials and on the status of the device

5.3.1 Usage and implementations

The A71CH Configure tool can be deployed either on a development PC or in a standalone embedded target. In the first case, the A71CH Configure Tool is installed on a PC, and the communication with the embedded target containing the A71CH is achieved over TCP/IP. Fig 7 illustrates the communication between the A71CH Configure Tool installed on a development PC and the A71CH connected to the i.MX6UltraLite host target.

The A71CH Configure Tool is executed in the Development PC and the APDUs are prepared and sent over the TCP/IP protocol. Once in the i.MX 6UltraLite embedded target, these packed APDUs are unpacked with the Remote JC Terminal Server. The unpacked APDUs are finally sent to the A71CH Security IC through the I²C interface.

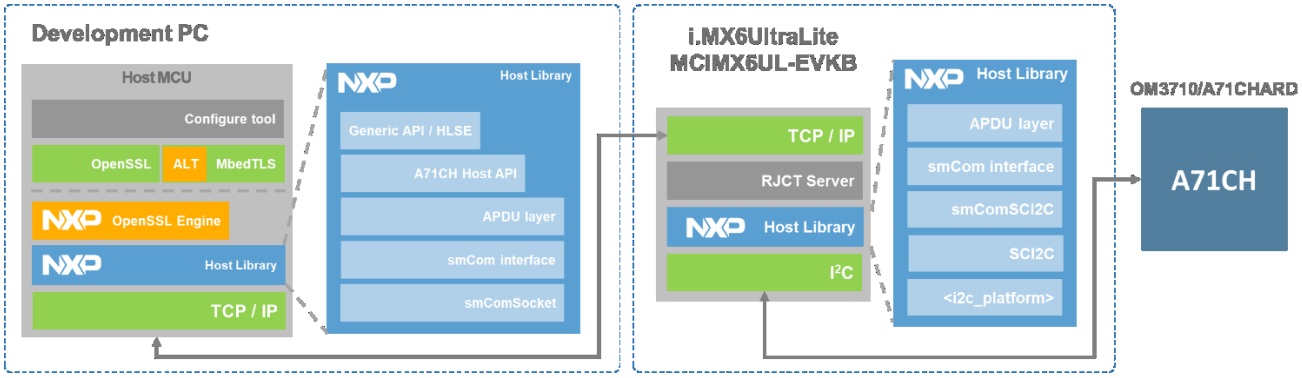

Fig 7. A71CH Configure tool installed on Development PC. Example using iMX6UltraLite.

In the second case, the A71CH Configure Tool is installed on the i.MX6UltraLite embedded target. No Remote JC Terminal Server is needed now since APDU's are prepared in the embedded target itself. Only the A71CH Configure Tool commands will be sent from the Development PC over TCP/IP. Fig 8 illustrates the described scenario.

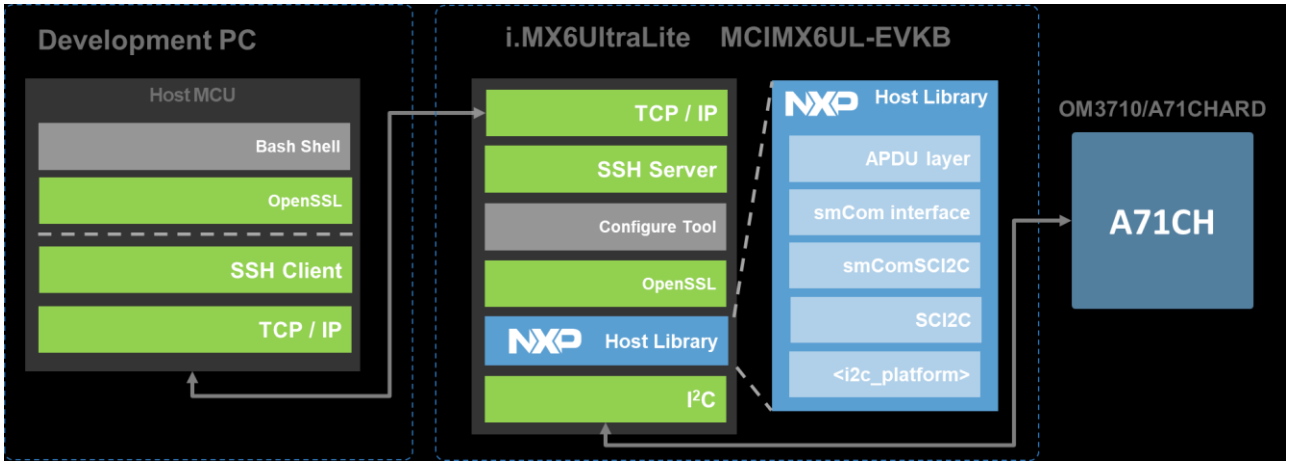

Fig 8. A71CH Configure tool installed on the embedded target

The last deployment scenario is depicted in Fig 9. In this case, the A71CH Configure tool is launched from the development PC and the APDUs are prepared and sent over a Kinetis board acting as a virtual COM interface, thus the used layers are the smComSerial and the platform vCOM USB driver. The Kinetis will be seen by the development PC as a virtual COM and will send the APDUs to the A71CH through I²C.

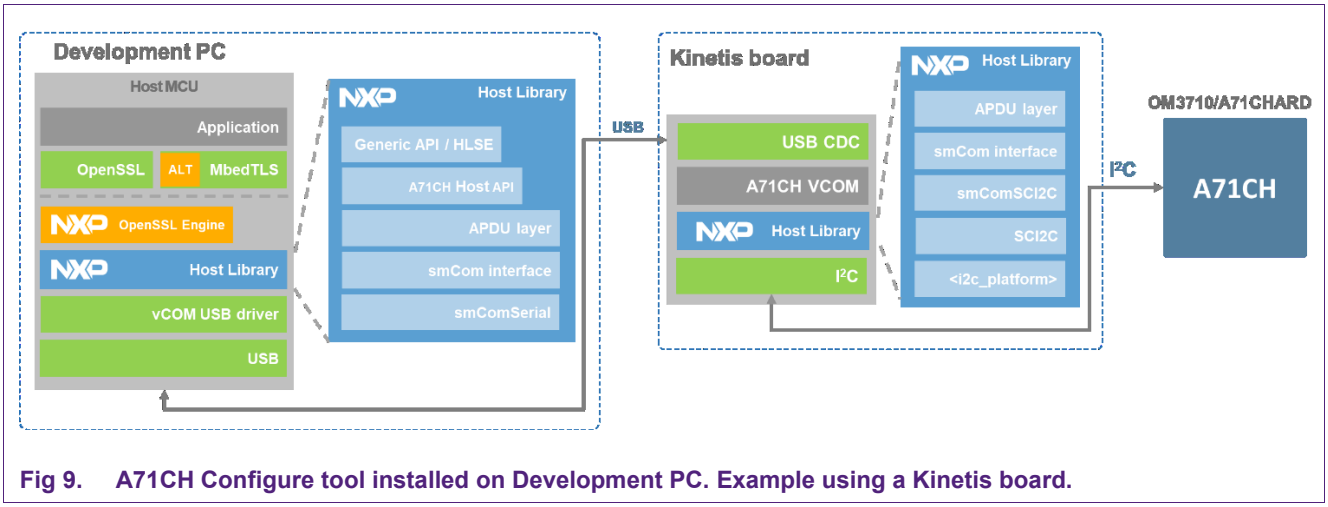

Fig 9. A71CH Configure tool installed on Development PC. Example using a Kinetis board.

The A71CH Configure tool can be used in three modes: *interactive mode*, *command line mode* and *batch file mode*.

In *interactive mode*, the A71CH Configure tool opens a communication session between the host and the A71CH module. Once this session has been open, the user can start sending configuration commands to the A71CH.

In *command line mode*, the user passes configuration parameters as command line arguments. Each command line will invoke the A71CH Configure Tool and, therefore, will open a new communication session between the Host and the A71CH.

In *batch file mode*, the A71CH Configure Tool can send multiple commands contained in a file. All commands contained in the file are handled in the same communication session between Host and A71CH.

5.3.2 Execution and supported input argument

The A71CH Configure Tool is commonly executed from a console by calling its executable file name followed by the supported input arguments. For instance, it can be called from an i.MX6UltraLite platform by typing the following command in a terminal:

```
./a71chConfig_i2c_imx <arg1> <arg2>
```

The supported arguments are the following:

- **interactive**: It starts the A71CH Configure Tool in interactive mode.
- **apdu**: It allows the user to Exchange an APDU in raw format between the Host and the A71CH.
- **connect**: It closes or re-open the connection with an attached Secure Element.
- **debug [permanently_disable_debug | reset]**: Resets the A71CH to initial state (if debug mode is still on) or permanently disables the debug mode of the A71CH.
- **ecrt**: Erases a certificate from the GP storage area by index.
- **erase [cnt | pair | pub | sym]**: It erases the value of the specified stored credential.
- **gen pair**: A valid ECC key pair is generated by the A71CH.

- **get**: Retrieves the public key value from either a public key or key pair at the index passed as argument and stores it (in pem format) in a file provided as argument.
- **info [all | device | cnt | pair | pub | status | objects]**: It echoes the value and/or status of the A71CH or a specified credential.
- **lock [pair | pub | sym]**: It locks credentials or general-purpose storage data segments.
- **obj erase**: Erases the object at the provided index.
- **obj get**: Gets the value of a data object, it retrieves the data from a specific offset within the data object (fetching the specified amount of byte). Optionally, the data is written to a file.
- **obj update**: Updates the value of a data object. It updates de data relative to an internal offset passed as a parameter. The data can be passed on the command line of be contained in a file.
- **obj write**: This command creates and object, the value of the which can be passed through the command line or can be contained in a file.
- **rcrt**: Reads a certificate from the GP storage area by index. Optionally, the command can save the certificate read to a CRT file.
- **refpem:** It creates OpenSSL specific reference .pem files.
- **script:** It issues Configure Tool commands contained in a file.
- **scp [put | auth]:** It writes a set of SCP03 keys to establish an active SCP03 channel between the Host and the A71CH or it clears the SCP03.
- **set [gp | pair]:** Writes a given data into the GP storage or a given key pair into the corresponding key storage.
- **set [cfg | cnt | sym]**: It sets a credential stored on the A71CH to a specific value. Set command can be used to modify the value of a general-purpose storage, key pair, public key, etc.
- **transport [lock | unlock]**: It enables or disables the transport lock on the A71CH.
- **ucrt [raw | pem | crt]:** Updates a certificate of the GP storage area by index. The certificate can be provided as raw data (-h), PEM (-p) or DER (-c).
- **wcrt [raw | pem | crt]:** Writes a certificate to the GP storage area by index. The certificate can be provided as raw data (-h), PEM (-p) or DER (-c).

5.3.3 Evaluating A71CH Configure tool

The following example (Fig 10) shows how A71CH Configure tool can be used to:

- Inject an OpenSSL generated ECC key pair and ECC public key into the A71CH security IC
- Create a reference key file and retrieve information about the stored keys

In this case, the A71CH Configure Tool is executed from a bash file from a Linux environment. Therefore, each command will open a new communication session between the Host and the A71CH.

It is assumed that .pem files containing the ECC keys have already been generated with OpenSSL commands. These keys are contained in *ecc_kp.pem* and *ecc_pub.pem* files.

```
# Connect with A71CH, debug reset A71CH and insert keys
./a71chConfig_i2c_imx debug reset
# Send key pair to the A71CH
./a71chConfig_i2c_imx set pair -x 0 -k ecc_kp.pem
# Obtain key pair reference from the A71CH
./a71chConfig_i2c_imx refpem -c 10 -x 0 -k ecc_kp.pem -r ecc_kp_ref.pem
# Send public key to the A71CH
./a71chConfig_i2c_imx set pub -x 0 -k ecc_pub.pem
# Obtain public key reference from the A71CH
./a71chConfig_i2c_imx refpem -c 20 -x 0 -k ecc_pub.pem -r ecc_pub_ref.pem
# Retrieve information about the A71CH key pairs and public keys
./a71chConfig_i2c_imx info pair
./a71chConfig_i2c_imx info pub
```

Fig 10. Configure Tool example of use

First, the debug mode of the A71CH is used to bring the A71CH in its initial state. Then, an ECC key pair is stored into the A71CH using **set** input arguments and its reference file *ecc_kp_ref.pem* is obtained with **refpem** argument. Similarly, an ECC public key is injected into the A71CH and the reference file *ecc_pub_ref.pem* is obtained. Finally, information about the stored keys is retrieved with the **info** input argument.

More information about the A71CH Configure Tool supported commands and deployment can be found in the 'A71CH Configure Tool' section of A71CH Doxygen documentation.

5.4 A71CH OpenSSL engine examples

A set of examples demonstrating the OpenSSL functionalities added by the A71CH OpenSSL Engine is contained in the A71CH Host software package. A brief description of TLS/SSL protocol and OpenSSL tool is provided in this section, then the A71CH OpenSSL Engine is presented and it functionalities are listed. Finally, the A71CH OpenSSL engine examples are described.

Extended information about the A71CH OpenSSL Engine and OpenSSL example scripts can be found in [A71CH_OPENSSL_ENGINE]. When using the windows installer for the A71CH Host SW package, this document will be available in the **doc** folder.

5.4.1 Transport Layer Security Protocol (TLS)

Transport Layer Security protocol (TLS), and its predecessor Secure Sockets Layer (SSL), are cryptographic protocols that provide communications security over unsecure

networks. These protocols are created from the necessity of establishing a connection preserving confidentiality, integrity and authenticity.

Fig 11 illustrates the protocol stack of a TLS communication over a TCP/IP network. In the well-known ISO/OSI layer architecture, SSL/TLS would belong to the presentation layer in charge of encrypting and securing the entire communication. The transport and network protocol TCP/IP and the medium access control (MAC) would fall in layers from 4 to 2, respectively. Finally, data would be electrically transferred according to ethernet (or wireless ethernet) protocols

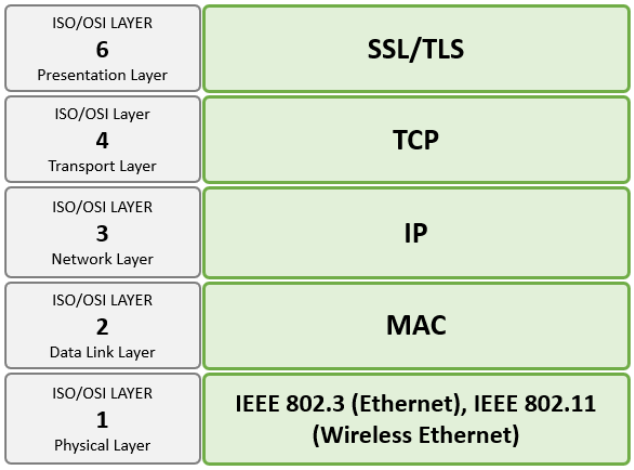

Fig 11. Communication stack. ISO/OSI Layers.

5.4.2 Transport Layer Security software libraries

There are several full-featured TLS software libraries that can be used such as OpenSSL, mbedTLS, WolfTLS, etc. OpenSSL [OPEN_SSL] is an open-source implementation of SSL/TLS protocol. It is written in C language, although there are several wrappers to use this library in other languages. It implements all the cryptography functions needed and it is widely used. Starting with OpenSSL 0.9.6 an 'Engine interface' was added allowing support for alternative cryptographic implementations. This Engine interface can be used to interface with external crypto devices as e.g. HW accelerator cards or security ICs like the A71CH.

The OpenSSL toolkit including an A71CH OpenSSL Engine is available as part of the A71CH Host software package. The A71CH OpenSSL Engine gives access to several A71CH features via the A71CH Host Library not natively supported by OpenSSL implementation. In other words, the Engine links the OpenSSL libraries to the A71CH Host API and overwrites some of the native OpenSSL functions in order to include the use of the A71CH crypto functionality such as sign, verify and key exchange operations or random messages generation, that can be used for instance during the TLS Handshake protocol.

The A7CH OpenSSL Engine is fully compatible with the i.MX6UltraLite embedded platform. Nevertheless, more support will be added in future revisions.

Fig 12 illustrates the Host MCU software architecture using OpenSSL. As it can be observed, the software stack is formed by an application that will call OpenSSL functions. Some of these functions will be overwritten by the A71CH OpenSSL Engine, thus the A71CH crypto functionality will be used through the A71CH Host Library over I²C.

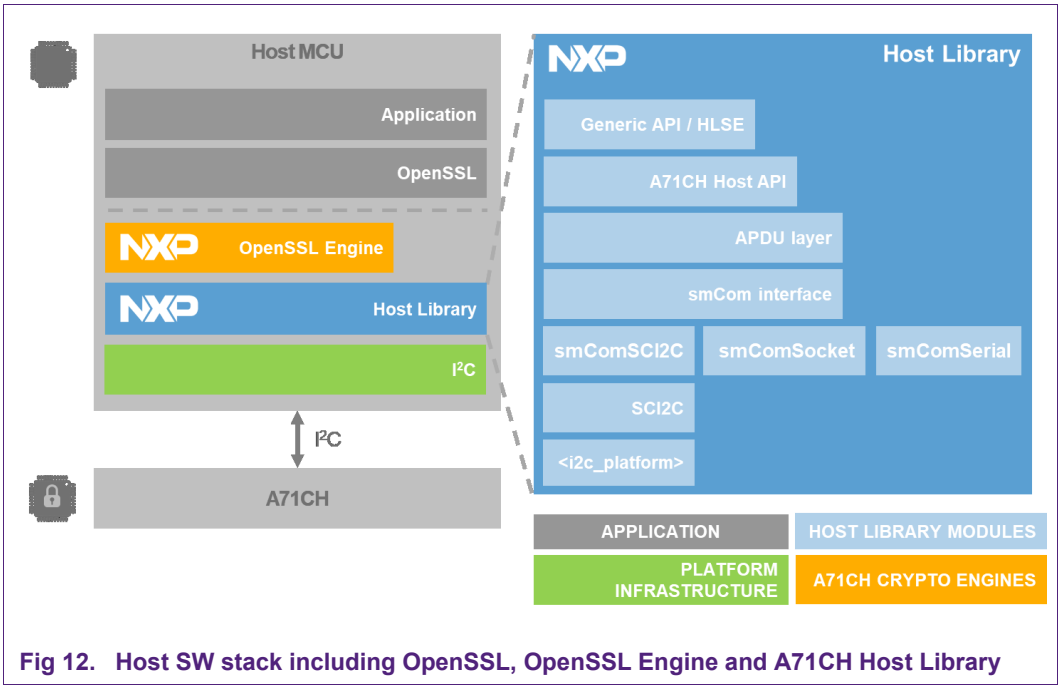

Fig 12. Host SW stack including OpenSSL, OpenSSL Engine and A71CH Host Library

The A71CH OpenSSL Engine examples can be divided into miscellaneous examples and TLS/SSL communication examples.

5.4.3 OpenSSL Engine miscellaneous examples

The OpenSSL Engine miscellaneous examples comprise all the A71CH Engine use cases. These are:

- **Initialize communication with the A71CH**: A series of ECC key pairs are generated using the native OpenSSL functions. The A71CH is then provisioned with these key pairs by using the A71CH Configure Tool.
- **Random number generation**: Up to three different-sized random numbers are requested to the A71CH using the OpenSSL Engine functionality.
- **CSR generation**: A certificate signing request (CSR) is generated and verified using either a key file or a key reference file.
- **DH Key Agreement**: Illustrates a Diffie-Hellman key agreement operation between a remote and a local party. The local party uses a key pair stored in the Secure Element.
- **ECDSA sign and verify operations**: Examples on how to sign or verify an encrypted file using the required key from the A71CH.

5.4.4 OpenSSL TLS communication example scripts

The A71CH OpenSSL Engine TLS connection examples show how to initiate a TLS/SSL based communication between a device acting as a client and a device acting as a server (e.g., IoT device with the A71CH security IC and an OEM server). The entire communication establishment procedure is divided into four parts, each one executed from a bash script:

- **tlsCreateCredentialsRunOnClientOnce**: Generation of all the necessary files, e.g., client and server ECC key pairs and TLS/SSL certificates.
- **tlsPrepareClient**: Injection of the client ECC key pair into the A71CH security IC using the A71CH Configure Tool.
- **tlsServer**: Server connection start. The server will start to listen to TLS/SSL communication requests through a specified port.
- **tlsSeClient**: Client connection start. The client will establish a TLS/SSL-based communication with a give IP address and port.

This section presents a description of the workflow of each script to provide a clear view of the TLS/SSL initialization process and the integration of both the A71CH security IC and the A71CH OpenSSL Engine on it.

5.4.4.1 tlsCreateCredentialsRunOnClientOnce.sh

This bash script can be executed to prepare all the required ECC keys and certificates for the TLS/SSL connection. The parameters of P-NIST 256 elliptical curves are generated using OpenSSL commands. These parameters are used to generate elliptical key pairs (private and public).

To simplify the credential creation process, the script creates both client and server credentials (once) on the client platform. One must transfer the server credentials created to the server platform.

First, the root CA is simulated and created by generating its root key pair and its root CA certificate using OpenSSL commands. In a real scenario, the CA is an external trusted entity whose root CA certificate and private key are securely stored on an HSM (Hardware Security Module).

After the CA certificate, the client and server credentials are created. The ECC private and public keys are generated using the P-NIST 256 ECC curves parameters as an OpenSSL function input argument. A CSR is then generated and sent to the CA. This CSR may contain, for instance, the client device public key, information about the IoT manufacturer, its main functionality, contact details, etc. A new certificate based upon information contained in the CSR is created and signed with the root CA private key. Finally, the client certificate is sent back to the client.

Similarly, the server ECC key pair is created and the CSR is prepared and sent to the CA. At the end of the process, the client and the server credentials are ready. Fig 13 illustrates the script workflow and the created credentials.

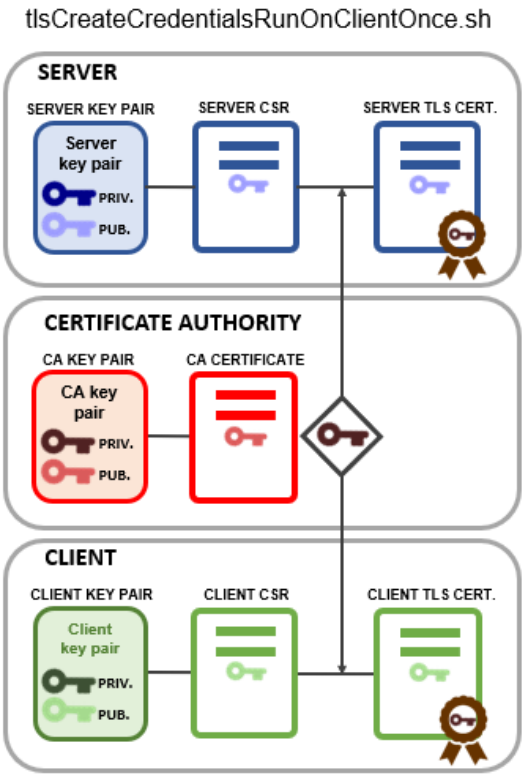

Fig 13. Certificate Authorizer, Client and Server key pair and TLS certificate creation

5.4.4.2 tlsPrepareClient.sh

This bash script injects the available client ECC private and public keys into the A71CH security IC. This will ensure that these are kept safe and protected: once injected, the private key will never leave the module. Whenever these keys are required to digitally sign or verify a file, it is possible to use them within the A71CH, using the A71CH OpenSSL Engine functionality. The keys injection is carried out using the A71CH Configure Tool commands line, provisioning the A71CH with the client keys.

Alternatively, it would be possible to generate the ECC key pair inside the A71CH using the A71CH Configure Tool or an application on top of the Host Library and prepare the CSR by retrieving the client public key with the OpenSSL Engine functionality. This would be the most recommendable approach since the keys are generated inside the A71CH security IC from the start and there is no need of injecting them from the outside. Also, the client digital certificate could be stored inside the A71CHI IC.

Fig 14 illustrates the state of the client device, composed by a MCU and the A71CH. The client contains its digital certificate and the key pair is securely stored inside the A71CH using the Configure Tool.

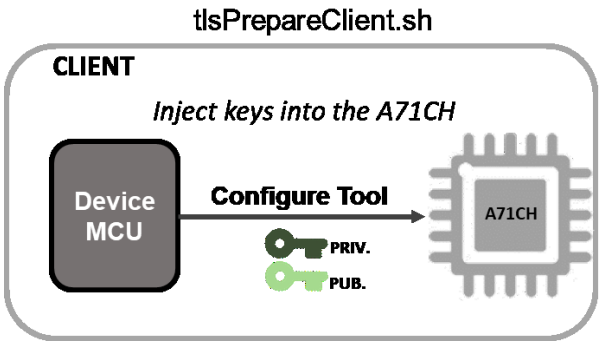

Fig 14. A71CH provisioning

5.4.4.3 tlsServer.sh and tlsSeClient.sh

This ***tlsServer.sh*** bash script starts the server with an OpenSSL command whose input arguments are the server key pair, the server certificate, the CA certificate and the port that it will be listening to.

Correspondingly, the ***tlsClient.sh*** script starts the client with an OpenSSL command that requires the client key pair, the client certificate, the CA certificate and the IP address and port of the server. Alternatively, the TLS client program (provided in source code) ***a71chTlsClient.c*** illustrates in detail how a user specific application can use the A71CH OpenSSL Engine to establish a TLS link and the retrieval of the client certificate from the A71CH.

Once the client and the server have been initialized, the TLS handshake starts. Fig 15 illustrates the connection between the client and the server through the port 8080 and the local IP address. An example showing how to connect to an OEM cloud can be found in [OEM_CONNECTION].

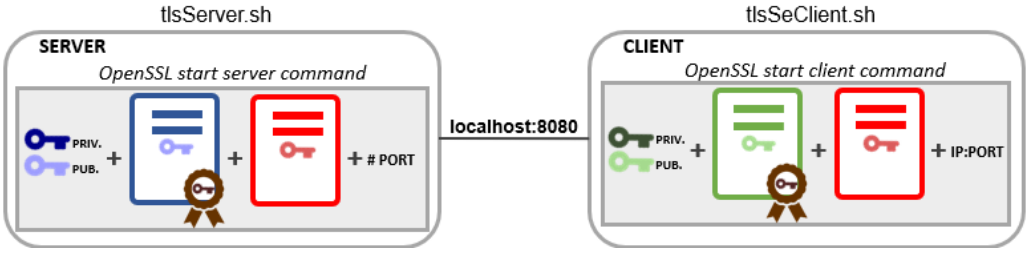

Fig 15. Server and Client TLS initialization

5.4.5 Transport Layer Security Handshake protocol

The TLS Handshake Protocol is responsible for the authentication and key exchange necessary to establish or resume secure sessions. When establishing a secure session, the TLS Handshake Protocol manages the following:

- Agree on the TLS protocol version to be used.
- Select cipher suite.
- Authenticate each other by exchanging and validating digital certificates.
- Use asymmetric encryption techniques to generate a shared secret key, which avoids the key distribution problem. SSL or TLS then uses the shared key for the symmetric encryption of messages, which is faster than asymmetric encryption.

The TLS Handshake Protocol involves the following steps:

- Exchange Hello messages to agree on algorithms, exchange random values, and check for resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a pre-master secret.
- Exchange certifications and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the pre-master secret and exchanged random value.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

The A71CH security IC supports the TLS Handshake Protocol version 1.2 and supports pre-shared key cipher suites (TLS-PSK) using either:

- Pre-Shared Key Cipher suites for TLS as described in [RFC4279]: A set of cipher suites for supporting TLS using pre-shared symmetric keys (*TLS_PSK_WITH_xxx*)
- ECDHE_PSK Cipher suites for TLS as described in [RFC5489]: A set of cipher suites that use a pre-shared key to authenticate an Elliptic Curve Diffie-Hellman exchange with Ephemeral keys (*TLS_ECDHE_PSK_WITH_xxx*).

In this section, The TLS 1.2 Handshake protocol is explained to provide the reader with the necessary background to understand how the connection is established using the scripts presented in 5.4.4.3., and also to show how the A71CH is involved during the process by using the A71CH OpenSSL Engine functionality.

The tlsServer.sh and tlsSeClient.sh scripts use Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) key agreement mechanisms and Elliptic Curve Digital Signature Algorithm (ECDSA). Therefore, the explanation is slightly oriented to the use of ECC and the ECDHE-ECDSA suite.

The TLS handshake explanation has been divided into the following steps:

- Exchange Hello messages phase
- Key Exchange phase: Server side
- Key Exchange phase: Client side

- Master secret calculation

5.4.5.1 Exchange Hello messages phase

To initiate the TLS handshake phase, the client prepares and sends a *ClientHello* message to the server. The *ClientHello* messages are used to establish the following attributes: TLS version, Session ID, Cipher Suite (set of algorithms for each cryptographic operation) and Compression Method. Additionally, a random value *ClientRandom* is generated with the A71CH using the A71CH OpenSSL Engine and sent within the message.

In the case of ECC, two extensions are included: the supported elliptic curves extension and the supported point formats extension. These two extensions allow a client to enumerate the elliptic curves it supports and/or the point formats it can parse.

The *ClientHello* message is received by the server. The server must use the client's enumerated capabilities to select an appropriate Cipher Suite. In this example, the server should select one of the proposed ECC Cipher Suites. Then, the server responds with a server Server*Hello* message that contains the chosen cryptographic algorithm from the list provided by the client, the session ID, a list of supported ECC curves and points extensions and another random byte string *ServerRandom*. Fig 16 illustrates the *Hello* messages exchange phase.

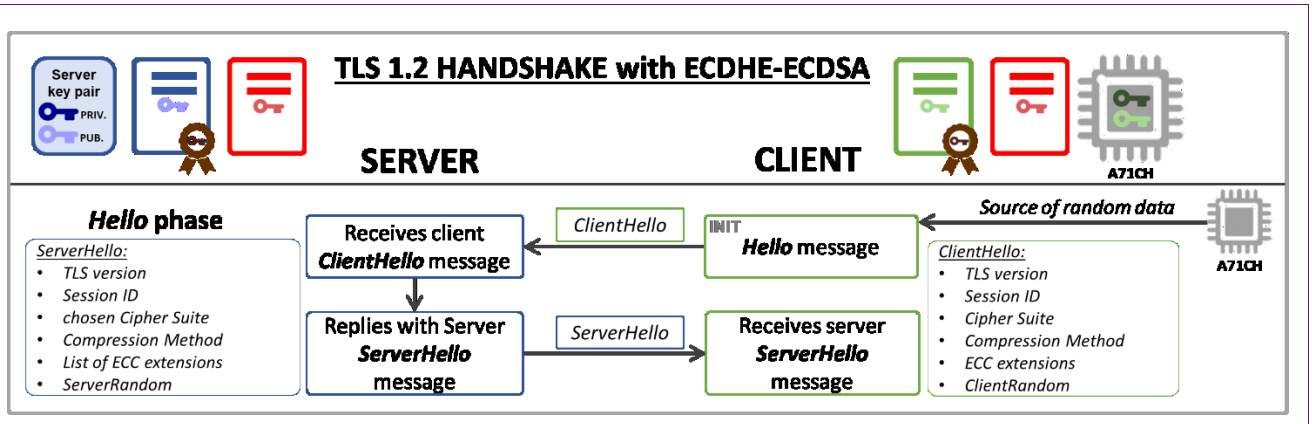

Fig 16. TLS 1.2 Handshake. Exchange *Hello* messages phase

5.4.5.2 Key Exchange phase: Server side

In TLS-secured connections, message exchange between two parties is secured with a symmetrical algorithm based on a shared secret key. The process of sharing a secret key between two parties, each having an elliptic curve public-private key pair, is known as Elliptic curve Diffie-Hellman (ECDH) key agreement.

More concretely, Elliptic curve Diffie-Hellman (ECDH) is an anonymous key agreement protocol that allows two parties to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key which can then be used to encrypt subsequent communications using a symmetric key cipher (e.g., obtain the master secret with a pre-master secret). The Ephemeral version of ECDH is called ECDHE, which means that a distinct Diffie-Hellman key is used in every key agreement instead of the already contained elliptic key pair (which will be referred as

static public-private pair from now on). ECDHE requires more cryptographic operations, since an additional key pair has to be generated, but provides forward secrecy which protects from deciphering the communication later even if the authentication keys get known.

In this example, both the Client and the Server have a pair of static elliptic curve public-private keys, and an ECDHE key agreement is carried out to start a secure messages exchange between the Client and the Server.

The Server will send its *Certificate* to convey its static public key to the Client. The conveyed public key must respect the agreed supported elliptic curves extensions and supported point formats extension. For example, if ECDHE and ECDSA have been accorded as the key exchange and signature algorithms (during the Hello exchange phase), the certificate must be signed with ECDSA using the Server static private key.

Once the Server certificate is received by the Client, this validates the certificate chain of trust (using the CA public key), extracts the Server's static public key and checks that the key matches with the negotiated Cipher Suite (e.g., checks that it is suitable for the negotiated key exchange algorithm). Now, the Client has the Server static public key, which means that it can validate every message signed by the Server static private key with ECDSA. The signature validation could be done by the A71CH through the A71CH OpenSSL Engine if the CA public key is stored inside it.

Following, an ephemeral elliptical key pair is created for the ECDHE key agreement. This new key pair is based on the domain parameters negotiated during the 'Hello' phase. Note that, the Server now contains two elliptic key pairs: the static key pair, and a new ephemeral key pair.

The ephemeral public key and the domain parameters are signed by the Server's static private key (which corresponds to the static public key in the Server certificate). Then, the ephemeral public key, domain parameters, and the signature block are sent to the Client through the *ServerKeyExchange* message. The *ServerKeyExchange* is sent by the Server only when the *Certificate* message does not contain enough data to allow the Client to exchange a pre-master secret. In this case, the *ServerKeyExchange* message is required since the Server's ephemeral public key must be sent to the Client for the pre-master secret computation. It would not be necessary in the case of ECDH or RSA key exchange algorithms, given that in both cases, the public key contained in the certificate is enough for the pre-master secret calculation.

Optionally, the Server may request the Client certificate (*CertificateRequest* message) to validate its digital identity and to obtain the Client public key. In that case, the Client must send the *Certificate* message containing the certificate.

The *CertificateRequest* message is mandatory, for instance, if RSA or ECDH algorithms would have been accorded as the key exchange protocols. In the case of ECDH, both Client and Server static public keys are required for the pre-master secret calculation and, again, the Server needs to know the Client's public key.

Finally, the Server will send a *ServerHelloDone* message, indicating that both the Hello phase and Key exchange phase have been finished. Fig 17 shows the Server key exchange phase of the TLS 1.2 Handshake.

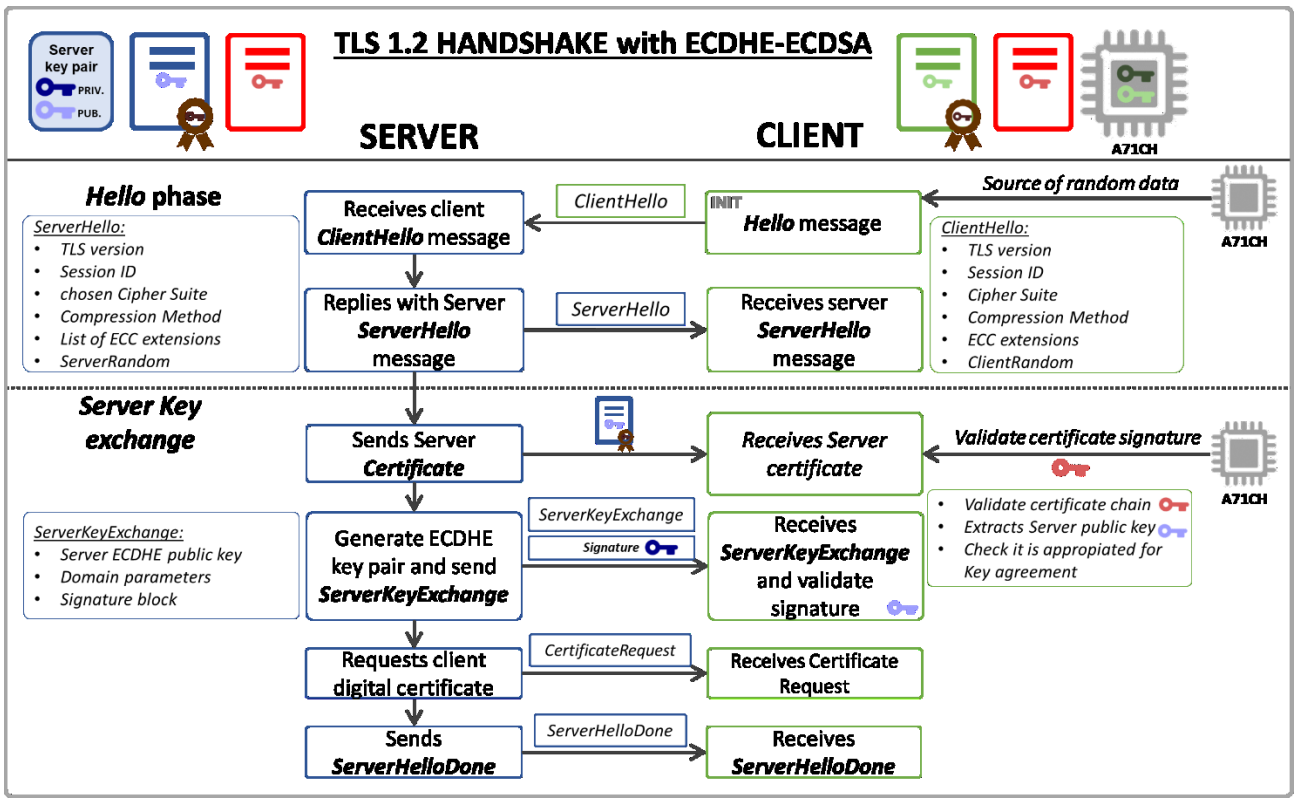

Fig 17. TLS 1.2 Handshake. Server key exchange

5.4.5.3 Key exchange phase: Client side

The Client will send its certificate in case a request has been received from the Server. This certificate shall contain the Client static public key, and it must be signed with the Client private key, according to the negotiated CipherSuite (i.e., ECDSA). As it is mentioned in section 5.4.4.2, this certificate could be stored in the A71CH, thus the application setting up the TLS link must ensure it retrieves the certificate from the A71CH.

If the Client has sent a certificate with signing ability, a digitally signed *CertificateVerify* message is sent to verify possession of the private key in the certificate. The *CertificateVerify* message is signed with the Client private key stored in the A71CH using the OpenSSL Engine.

Then, the Client generates an ECDHE ephemeral key pair from the domain parameters sent in the *ServerKeyExchange* message. In this way, both generated ephemeral elliptic key pairs belong to the same finite domain, and thus the shared secret computation can be correctly performed. Similarly as with the *Hello* message, the A71CH can be used as a source of random data to generate the ECDHE ephemeral key pair.

The *ClientKeyExchange* message containing the Client's ephemeral ECDHE public key is sent to the Server. This *ClientKeyExchange* message is not optional and shall always be sent.

Finally, the Server retrieves the client's ephemeral ECDHE public key from the *ClientKeyExchange* message and checks that it is on the same elliptic curve as the Server's ECDHE ephemeral public key. Fig 18 shows the key exchange phase of the TLS 1.2 Handshake.

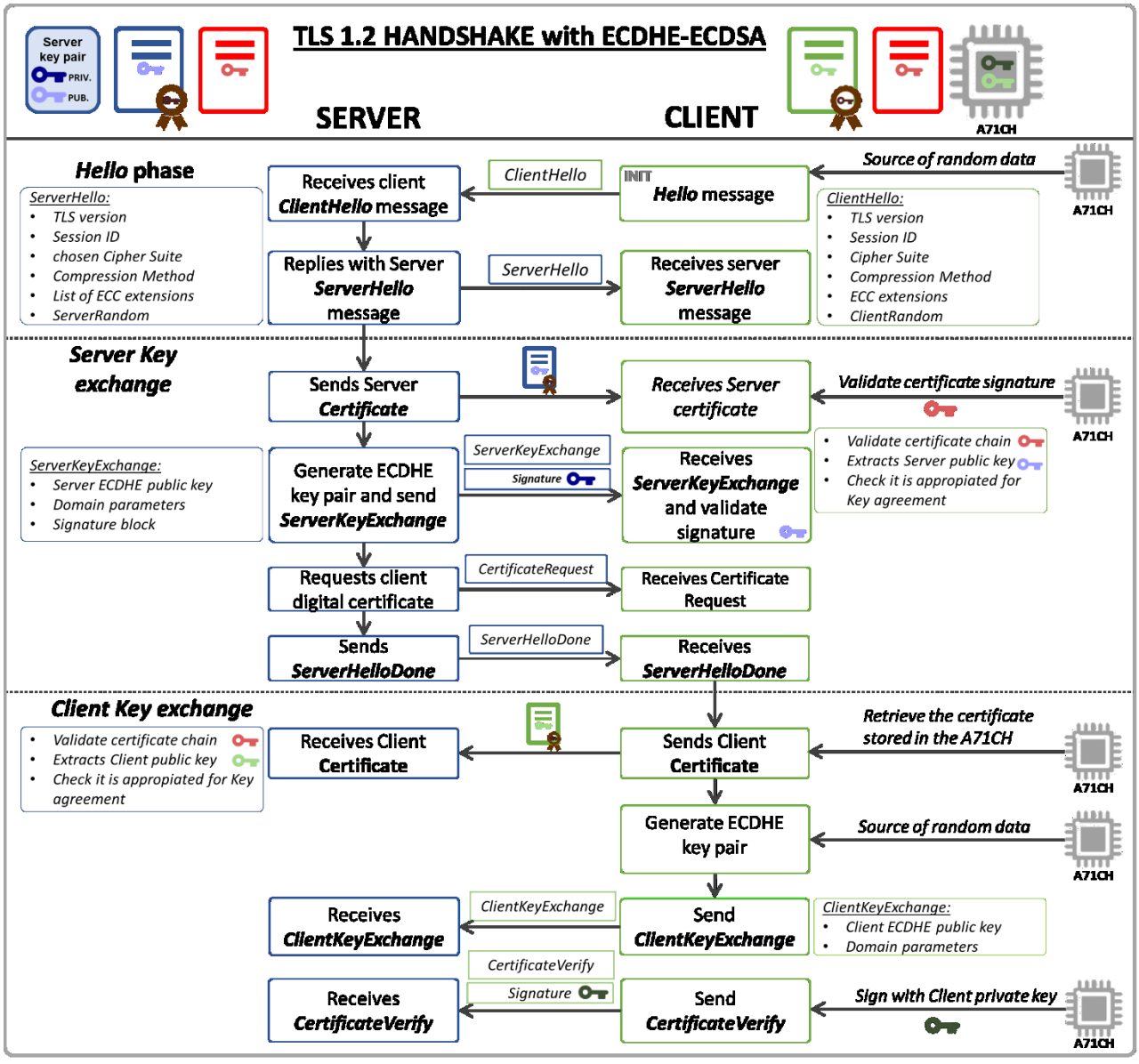

Fig 18. TLS 1.2 Handshake. Client key exchange

5.4.5.4 Master secret calculation

To complete the Key Exchange, the Client and the Server perform an ECDH operation to generate the pre-master secret. These pre-master secrets, client and server random bytes (*ClientRandom* and *ServerRandom*) generated during the Hello phase and an identifier label are used by both the Client and the Server to generate the same master secret independently.

Finally, the Client sends a *ChangeCipherSpec* message to inform the Server that key exchange has been successfully completed. This message is followed by a *Finished* message encrypted with the calculated master secret.

Similarly, when the Server receives the client's *Finished* message, it sends a *ChangeCipherSpec* message to also inform the Client that the key exchange has been a success, followed by a *Finished* message encrypted with a key derived from the master secret.

At this point, both the Client and the Server are in possession of the master secret key and can start to securely exchange messages using a symmetric cryptographic algorithm. Also, both the Client and the Server have agreed on the used algorithms and have authenticated themselves. Finally, Fig 19 shows the complete TLS 1.2 Handshake diagram, including this last step.

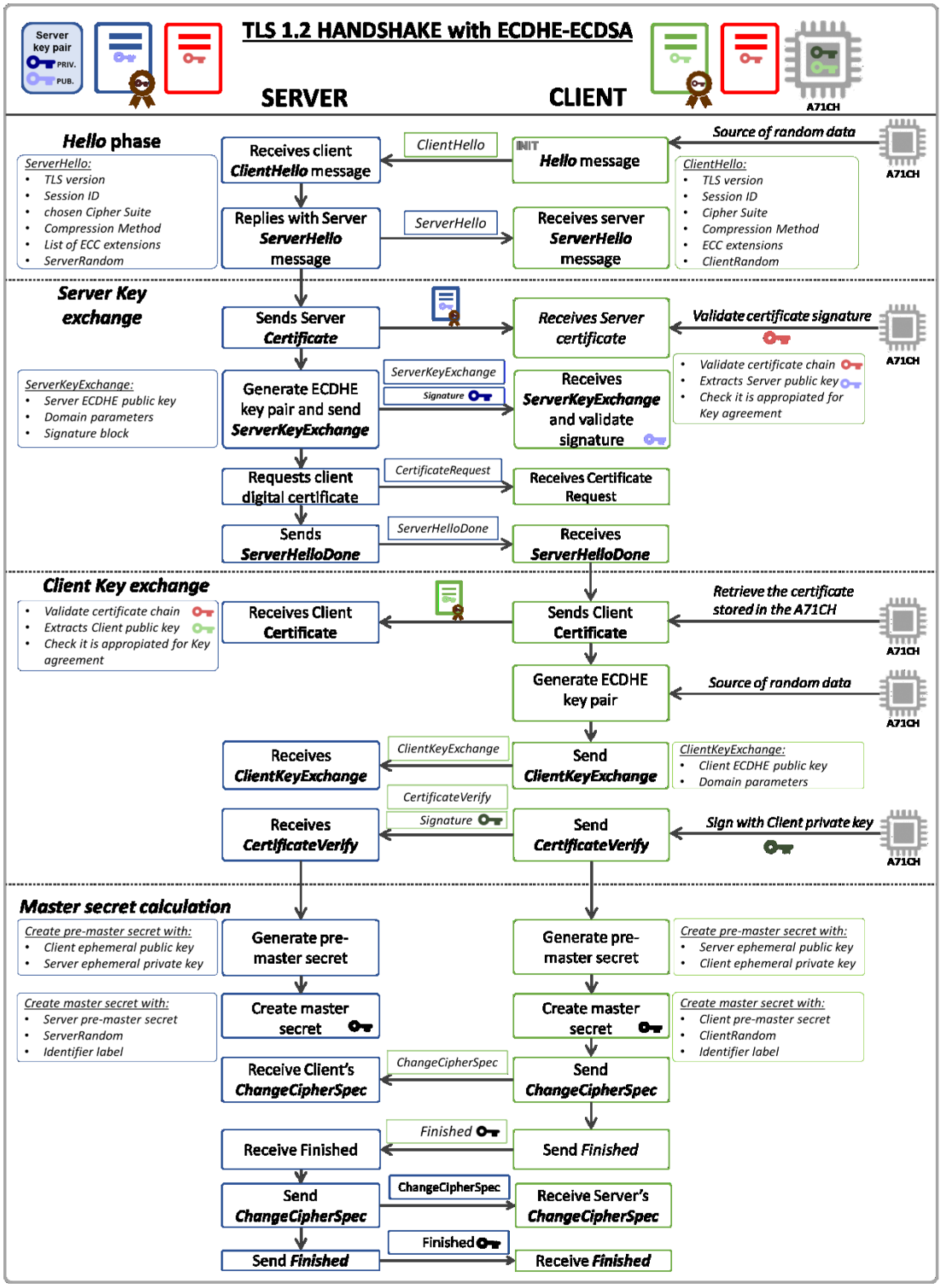

Fig 19. TLS 1.2 Handshake. Complete diagram

More information about the TLS 1.2 protocol can be obtained from the standard specifications document [RFC5246]. Additionally, a detailed view of each one of the exchanged messages during the Handshake can be found in the Doxygen documentation available in [A71CH_HOST_SW].

6. Referenced Documents

Table 2. Referenced Documents

[SCI2C]	**SCI2C Protocol Specification** – Revision 1.x only, Docstore an1950**[1]
[A71CH_APDU]	**APDU Specification of A71CH Security Module -** DocStore ds4094**[1]
[SCP03]	**Global Platform Card Specification v2.3** – Amendment D v1.1.1.
[A71CH_HOST_SW]	**A71CH Host Software Package (Bash installer for Windows) –** DocStore, document number sw4673xx[1]**,** Version 01.03.00 (or later), available on www.nxp.com/A71CH **A71CH Host Software Package (Bash installer for Linux) –** DocStore, document number sw4672xx[1]**,** Version 01.03.00 (or later), available on www.nxp.com/A71CH
[OPEN_SSL]	**OpenSSL Cryptography and SSL/TLS Toolkit information** - www.openssl.org
[RFC4279]	**Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)** - December 2005
[RFC5489]	**ECDHE_PKE Cipher Suites for Transport Layer Security (TLS)** - March 2009
[RFC5246]	**The Transport Layer Security (TLS) Protocol** - Version 1.2, August 2008
[RFC4492]	**Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) -** May 2006
[OEM_CONNECTION]	**AN12132 A71CH for secure Connection to OEM Cloud** – Application note, document number 4642**[1]
[A71CH_OPENSSL_ENGINE]	**A71CH OpenSSL Engine** – DocStore, document number um4334**[1]

[1] **… document version number

7. Legal information

7.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

7.1 Licenses

ICs with DPA Countermeasures functionality

NXP ICs containing functionality implementing countermeasures to Differential Power Analysis and Simple Power Analysis are produced and sold under applicable license from Cryptography Research, Inc.

7.2 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

FabKey — is a trademark of NXP B.V.

I²C-bus — logo is a trademark of NXP B.V.

8. List of figures

9. List of tables

10. Contents

For more information, visit: http://www.nxp.com

Date of release: 23 April 2018 464313

Document identifier: AN12133