SEcube Firmware's Programmer Guide

Project Report

Filippo Cottone | Pietro Scandale | Francesco Vaiana
Luca Di Grazia | Julia Roca Garcia

Politecnico Di Torino
MD in Embedded Systems

Contents

1 SEcube overview architecture

The new developed architecture was designed to obtain a hierarchical firmware structure:

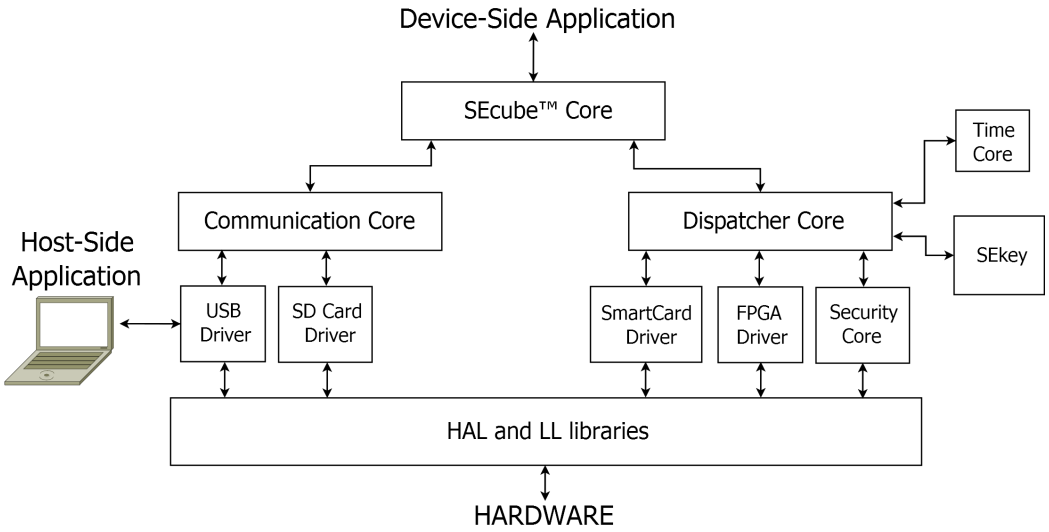

Figure 1.1: *SEcube new layered architecture.*

The request comes in a packet form: there are user written functions, that implement the interface to usb drivers, where there is the execution of its functionalities, allowing a communication through a special .secube file, located in the SD card. The host requires for a service by locking this file through operating system APIs and writing on it block-wise requests. A block is formed of 512 bytes. The device side will loop until the lock is released to serve the incoming request, locking in turn the same file. The host will loop until the locks became available with a certain deadline, and when it will be freed by the device side, it will understand that the response is arrived and will read on it the outcome of the request.

se3_proto_recv is the function to receive a block. It checks whether the block contains a special constant sequence of data, the magic sequence, that is used to let the firmware know the incoming blocks are not addressed to a general mass memory device, but they are operational elements directly referred to the SEcube firmware. Assuming that the device is ready to serve a new request, the block is magic and it has not been stored in the SDcard yet, it will be wrote in it, and the relative presence flags updated. If it has already been written yet, the incoming blocks may be a command or a data one. They will be treated in a different manner: data block will be directly forwarded to SDcard; command blocks, instead, will be unpacked and forwarded to the core. As mentioned before, the host is still locking the file where this data are stored, so no one is able to read this information.

At the reset phase the SEcube core makes the initialization of the main data structures for performing the communication, then it waits in an infinite loop a request from the host side. Once the device got a new request, it will be unpacked and the right command is prepared. Here the command to be executed is chosen among all the available. This task is performed because the core contains a table of all the functions that SECUBE can run; the core understands whether the function can be execute by itself, otherwise the dispatcher is called.

The functions that the core can execute are *factory_init*, to initialize the device, *echo* to share a message and receive it back exactly the same, *bootmode_reset* to reset the device, and a special command, *SE3_MIX*, that tells the device that the packects need furher unpacking to be interpreted, and must be sent to dispatcher core to be executed, since it requires low level security operations.

Once the request is served, it is sent to the communication core back, that will implement the interface of the low level read call by the usb driver. This function, called *SE3_proto_send*, will receive buffer from the user: if this blocks are not "magic", the firmware must act in a transparent way, providing access to SDcard. Otherwise the firmware will understand whether a data block or a response block from the execution of a command is required and will forward this data.

The dispatcher manages all that is related to the cryptographic part of the device in a very "easy to develop" way, by offering in a simple manner crypto algorithms and the possibility to easily add new ones. Considering that the security core, the smartcard and the fpga must be accessed and programmed within the chip for security reasons, this kind of interface is necessary to achieve the security requirements.

The dispatcher module contains a table of functions that the core cannot perform by itself, but it delegates the dispatcher in order to perform security algorithms or to retrive security information. So, the core sends a packet with the information about the functions that dispatcher has to execute, and the relative information. For example, if the core sends a cryptographic request, the right function is invoked in the dispatcher and the relative data structure is filled with the right information: in this case they are the algo for the current sessions, the direction (cription or decryption) and the private key of the user. The latter is searched in the SEKey module and if it's present, the session is allocated and 2 variables are set with the information about the security policies to use and also the hardware module that must compute the task, otherwise it returns an error and abort the operation.

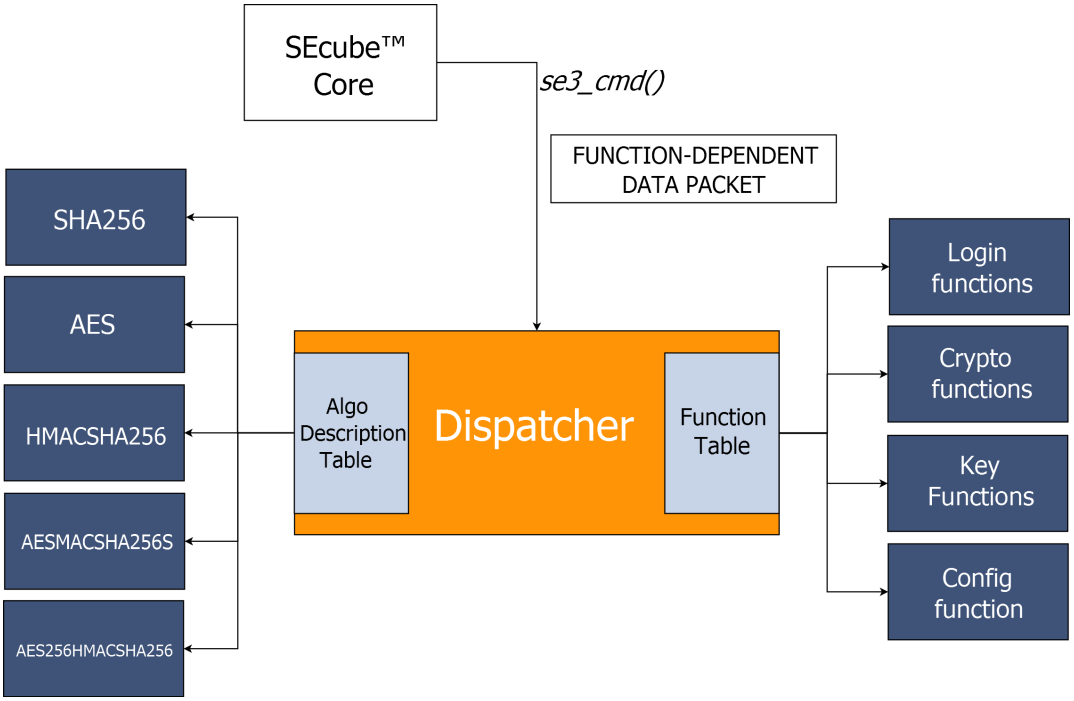

Figure 1.2: *SEcube Dispatcher functions.*

Thanks to SEKey module, it's possible to introduce the concept of groups, so every user has its own unique key and also a list of key for each group he belongs to. 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 "reserved information" used by cryptographic algorithms, and, in the latter one, to set-up secure communication channels. The security policies allow specifying the cryptographic algorithm used to protect the information related to that group and the mechanism to be adopted for generating the session keys.

The group table and which algorithms and hardware to use are decided by the system administrator during the initialization of the device. So SECube provides the possibility of a user mode, who uses the functionalities offered by the device, and a superuser mode, that can configure the device adding algorithms and granting the privileges to next users thanks to SEkey module.

The dispatcher provides the interface to the other cores that can accelerate the execution of crypto algos, i.e. FPGA or Smartcard Applets. This comes in an easy-to-use way: SEkey sets a variable that will be used to select the right security component. This will directly communicate with security core, fpga, or smartcard interface that will return an error if the module is not able to serve the request, or perform the operation if they could.

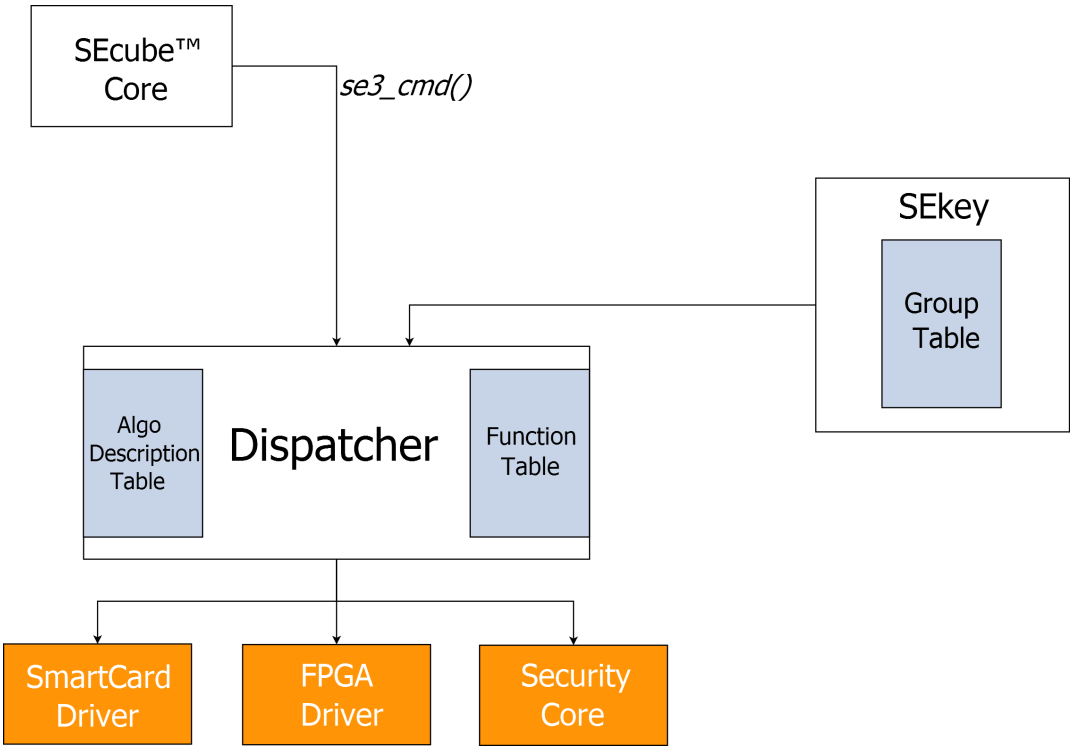

Figure 1.3: *SEcube Dispatcher interface.*

2 Communication Core

2.1 Communication Data Structures

- ***SE3_COMM_STATUS*** *comm*: Contains information about host-device communication status and buffers;
- ***se3_comm_resp_header*** *resp_hdr*: Header buffer containing information about the response to be forwarded to the host.
- ***se3_comm_req_header*** *req_hdr*: Header buffer containing information about the incoming request buffer. It came filled of information if the incoming blocks are magic.
- ***s3_storage_range*** *s3_storage_range*: SDIO read/write request buffer context.

2.2 Communication Functions

- ***void*** *se3_communication_core_init()*: Initializes the communication core structures;
- ***static bool*** *block_is_magic(const uint8_t* buf)*: Check if a block of data contains the magic sequence, used to initialize the special protocol file.
- ***static int*** *find_magic_index(uint32_t block)*: Return the index of the corresponding protocol file block, or -1 if the block does not belong to the protocol file. The special protocol file is made up of multiple blocks. Each block is mapped to a block on the physical storage.
- ***static int32_t*** *se3_storage_range_add(s3_storage_range* range, uint8_t lun, uint8_t* buf, uint32_t block, enum s3_storage_range_direction direction)*: Used to add requests to SDIO read/write buffer. Contiguous requests are processed with

a single call to the SDIO interface, as soon as a non-contiguous request is added.

- ***void*** *se3_proto_request_reset()*: Reset the protocol request buffer, making the device ready for a new request;
- ***static void*** *handle_req_recv(int index, const uint8_t* blockdata)*: Handles a single block belonging to a protocol request. The data is stored in the request buffer. As soon as the request data is received completely, the device will start processing the request;
- ***int32_t*** *se3_proto_recv(uint8_t lun, const uint8_t* buf, uint32_t blk_addr, uint16_t blk_len)*: User-written USB interface that implements the write operation of the driver; it forwards the data on the SD card if the data block does not contain the magic sequence, otherwise the data block is unpacked for further elaborations;
- ***static void*** *handle_resp_send(int index, uint8_t* blockdata)*: Output a single block of a protocol response. If the response is ready, the data is taken from the response buffer, otherwise the 'not ready' state is returned;
- ***int32_t*** *se3_proto_send(uint8_t lun, uint8_t* buf, uint32_t blk_addr, uint16_t blk_len)*: User-written USB interface that implements the read operation of the driver; it sends the data on the SD card if the data block does not contain the magic sequence, otherwise it handles the proto request.

3 SEcube Core

3.1 SEcube Core Data Structures

- ***uint8_t*** *se3_session_buf*: Buffer for sessions;
- ***uint8_t*** *se3_session_index*: Array containing index for sessions;

3.2 SEcube Core Functions

- ***void*** *device_init()*: Initializes all the useful core components and SEcube firmware architecture, including the communication core and the dispatcher core;
- ***void*** *device_loop()*: endless loop to keep the core listening for any possible requests;
- ***void*** *se3_cmd_execute()*: unpacks the request buffer to get the command to be executed. It sets the handler if the requested command can be executed by the core itself, otherwise the dispatcher needs to be invoked. This handler is always forwarded to the *se3_exec* function.
- ***static*** *uint16_t se3_exec(se3_cmd_func handler)*: Here the command is executed with the extracted parameters and the response is prepared with the correct return values. After that this function is invoked, the response buffer is ready to be sent to the host-side. The return value of the function is the number of blocks to be sent.

3.2.1 Command Functions

This section will describe the functions that can be requested by the host-side.

- ***uint16_t*** *echo(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Share a message and receive it back exactly the same.
- ***uint16_t*** *factory_init(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Initialize the device.
- ***uint16_t*** *bootmode_reset(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Reset the device.
- ***static uint16_t*** *invalid_cmd_handler(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: handler for invalid command request;

4 SEcube Core time

4.1 SEcube Core time Functions

- ***void*** *se3_time_init()*: It resets the two time variables: *now_initialized* and *now*;
- ***uint64_t*** *se3_time_get()*: It returns a copy of the variable *now*, because it is a private variable;
- ***void*** *se3_time_set(uint64_t t)*: It sets the variable *now* with the argument value and *now_initialized* to true.
- ***void*** *se3_time_inc()*: It increments timer using software.
- ***bool*** *get_now_initialized()*: It returns a copy of the variable *now_initialized*, because it is a private variable;

5 Memory

5.1 Memory Data Structures

- ***se3_mem*** *mem*: It is the structure of a memory allocator;

5.2 Memory Functions

- ***void*** *se3_mem_init(se3_mem* mem, size_t index_size, uint8_t** index, size_t buf_size, uint8_t* buf)*: It initializes the *se3_mem* structure with the function argument values.
- ***static void*** *se3_mem_compact(uint8_t* p, uint8_t* end)*: It resizes the memory allocation of the pointer passed to the function and if it is possibile reduce memory allocated.
- ***static uint8_t**** *se3_mem_defrag(se3_mem* mem)*:: It finds the first free block, rebuilds pointer table and then returns first empty block.
- ***int32_t*** *se3_mem_alloc(se3_mem* mem, size_t size)*: It allocates the pointer with the size passed on argument;
- ***uint8_t**** *se3_mem_ptr(se3_mem* mem, int32_t id)*: It returns the pointer to entry in buffer;
- ***void*** *se3_mem_free(se3_mem* mem, int32_t id)*: It releases the memory entry allocated for the pointer on the arguments;
- ***void*** *se3_mem_reset(se3_mem* mem)*: It releases all the memory entry allocated;

6 SE3 Keys

Flash key structure

Disposition of the fields within the flash node:
0:3 id
4:7 validity
8:9 data_size
10:11 name_size
12:(12+data_size-1) data
(12+data_size):(12+data_size+name_size-1) name

6.1 SE3 Key Data Structures

- ***se3_flash_key*** *key*: It is the flash key structure of a single node.

6.2 SE3 Key Functions

- ***bool*** *se3_key_find(uint32_t id, se3_flash_it* it)*: It finds a key in the flash;
- ***bool*** *se3_key_remove(se3_flash_it* it)*: It removes a key in the flash;
- ***bool*** *se3_key_new(se3_flash_it* it, se3_flash_key* key)*: It adds a key in the flash;
- ***void*** *se3_key_read(se3_flash_it* it, se3_flash_key* key)*: It reads a key in the flash;
- ***bool*** *se3_key_equal(se3_flash_it* it, se3_flash_key* key)*: It checks if a key is equal to a key stored in the flash;

- ***void*** *se3_key_read_data(se3_flash_it* it, uint16_t data_size, uint8_t* data)*: It reads the key's data from a key node;
- ***bool*** *se3_key_write(se3_flash_it* it, se3_flash_key* key)*: It writes the key data to a flash node;
- ***void*** *se3_key_fingerprint(se3_flash_key* key, const uint8_t* salt, uint8_t* fingerprint)*: It produces a salted key fingerprint;

7 Dispatcher Core

This module is invoked when the request coming from the host needs operations from security hardwares. It holds information about the login status, the allocated sessions, and tables containing all security functions that can be invoked from the outside. All of the security functions have a precise format, specified by the **se3_cmd_func** type.
The dispatcher core will filter and manage all the request intended to the security core, FPGA or Smartcard. All the functions are saved in Matrix of handlers, sized $SE3\_N\_HARDWARE * SE3\_CMD1\_MAX$, where SE3_N_HARDWARE is the number of available hardware and SE3_CMD1_MAX is the number of functions, that SEcube can execute.

7.1 Dispatcher Data Structures

- ***se3_comm_req_header*** *req_hdr*: Header buffer containing information about the security function. It needs to be decrypted.
- ***SE3_LOGIN_STATUS*** *login_struct*: Contains the useful status data for login operations;
- ***se3_cmd_func*** *handlers*: Table of function where all the possible operation of SECube are stored. It it organized as a matrix where at each row corresponds a different hardware or software implementation;

7.2 Dispatcher Functions

- ***void*** *se3_dispatcher_init()*: initializes the dispatcher data structures and the security components (Security core, FPGA, Smartcard) and sets the access level for all the users, both in read and write;

- ***void*** *set_req_hdr(se3_comm_req_header req_hdr_i)*: sets the req_hdr data structure to the structure passed as parameter;
- ***static void*** *login_cleanup()*: clean the security sessions information and access data;

7.2.1 Command Functions

- ***uint16_t*** *dispatcher_call(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: executes a security function, by using the implementation chosen by SEkey.
- ***uint16_t*** *config(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: set or get configuration record from the request buffer;
- ***uint16_t*** *key_edit(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: insert, delete or update key;
- ***uint16_t*** *key_list(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: list all keys in 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 server for the authentication;
- ***uint16_t*** *login(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: respond to challenge, completing login operation and the authentication;
- ***uint16_t*** *logout(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Log out and release resources; set or get configuration record from the request buffer
- ***uint16_t*** *error(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: handler for invalid command request;

8 SEkey

The following functions are provided to interface the Dispatcher Core to the Key Manager SEkey.

8.1 SEkey Function

- ***bool*** *sekey_get_implementation_info(uint8_t* algo_implementation, uint8_t* crypto_algo, uint8_t* key)*: given a key, if it is present inside the SEkey key list this function will return the possible security component that the key allows you to use and the security algorithm that should be performed;
- ***bool*** *sekey_get_auth(uint8_t *key)*: given a key, if it is present inside the SEkey key list a true boolean value is return, false otherwise;

9 Cryptographic Security Components

All the following components implement their own cryptographic algorithm basing on the component nature.

9.1 Security Core

The security core is the microprocessor-based cryptographic component able to encrypt or decrypt data, initialize and manage the cryptographic algorithms and the relative context for each session.

9.1.1 Security Core Data Structures

- ***se3_algo_descriptor*** *algo_table [SE3_ALGO_MAX]*: this array contains the Cryptographic algorithm handlers and information about the contained algorithms (init and update functions, name of algorithm, ...).
- ***SE3_SECURITY_INFO*** *se3_security_info*: contains information about the security sessions, including the algorithm for each session and the records;

9.1.2 Security Core Functions

- ***void*** *se3_security_core_init()*: Initializes the Security Core structure;
- ***static bool*** **record_find(uint16_t record_type, se3_flash_it* it)**: Given a record type (user pin or admin pin), it returns a true boolean value if the type is found in the flash memory and the passing iterator is set to the memory location;
- ***bool*** *record_set(uint16_t type, const uint8_t* data)*: Set data of a record; if a flash operation fails, the Hwerror flag is set;

- ***bool*** *record_get(uint16_t type, uint8_t* data)*: get data from a record; the data will be returned by using the *data* pointer;
- ***uint16_t*** *crypto_init(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Initialize a crypto context; it sets the request parameters from the *req* buffer (algo, mode, key_id, session ID) by checking whether the read data are consistent with the expectable ones;
- ***uint16_t*** *crypto_update(uint16_t req_size, const uint8_t* req, uint16_t* resp_size, uint8_t* resp)*: Use a crypto context, by calling the update function relative to the chosen algorithm;
- ***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 list of available algorithms;
- ***void*** *se3_payload_cryptoinit(se3_payload_cryptoctx* ctx, const uint8_t* key)*: Initialize the crypto algorithms;
- ***bool*** *se3_payload_encrypt(se3_payload_cryptoctx* ctx, uint8_t* auth, uint8_t* iv, uint8_t* data, uint16_t nblocks, uint16_t flags, uint8_t crypto_algo)*: encrypt the *data* buffer by using the algorithm assigned by SEkey decribed by the *crypto_algo* input parameter;
- ***bool*** *se3_payload_decrypt(se3_payload_cryptoctx* ctx, const uint8_t* auth, const uint8_t* iv, uint8_t* data, uint16_t nblocks, uint16_t flags, uint8_t crypto_algo)*: decrypt the *data* buffer by using the algorithm assigned by SEkey decribed by the *crypto_algo* input parameter;

9.2 FPGA

In order to add functions related to FPGA HW component, the corresponding Dispatcher Matrix handler column must be filled and the function must be developed inside the *se3_FPGA.c* file. Furthermore, additional information on the Key Manager SEkey must be provided to allow the functioning of the security component.

9.3 Smartcard

In order to add functions related to Smartcard HW component, the corresponding Dispatcher Matrix handler column must be filled and the function must be developed inside the *se3_smartcard.c* file. Furthermore, additional information on the Key Manager SEkey must be provided to allow the functioning of the security component.

10 Flash

Structure of flash:

0:31 magic
32:2047 index
2048:131071 data

The data section is divided into 2016 64-byte blocks. Each byte of the index stores the type of the corresponding data block. A special value (SE3_FLASH_TYPE_CONT) indicates that the block is the continuation of the previous one. If the block is invalid, its type is 0. If it has not been written yet, the type is 0xFF.

10.1 Flash Data Structures

- ***se3_flash_it*** *it*: Data structure to iterate over the on-chip flash memory. It contains the node address, its size, the type of the block, and the relative offset.
- ***SE3_FLASH_INFO*** *flash*: Data structure that contains informations about the active sector number, the base address, the type of the block, the data, the first free position, the value of memory used and allocated.

10.2 Flash Functions

- ***bool*** *se3_flash_init()*: Initialize flash;
- ***void*** *se3_flash_it_init(se3_flash_it* it)*: Initialize flash iterator;
- ***bool*** *se3_flash_it_next(se3_flash_it* it)*: Increment flash iterator;

- ***bool*** *se3_flash_it_new(se3_flash_it* it, uint8_t type, uint16_t size)*: Allocate new node in the flash and points the iterator to the new node;
- ***bool*** *se3_flash_it_write(se3_flash_it_write(se3_flash_it* it, uint16_t off, const uint8_t* data, uint16_t size)*: Write to flash node;
- ***bool*** *se3_flash_it_delete(se3_flash_it* it)*: Delete flash node;
- ***bool*** *se3_flash_pos_delete(size_t pos)*: Delete flash node by index;
- ***size_t*** *se3_flash_unused()*: Get unused space in the flash memory, including the space marked as invalid. If space is available, it does not mean that flash sectors will not be swapped. It return unused space in bytes;
- ***bool*** *se3_flash_canfit(size_t size)*: Check if there is enough space for new node;
- ***void*** *se3_flash_info_setup(uint32_t sector, const uint8_t* base)*: Initializes the structures for flash management, selecting a sector and its base address;
- ***bool*** *se3_flash_bootmode_reset(uint32_t addr, size_t size)*: Reset the USEcube device to boot mode by erasing the signature - zeroise.
- ***static bool*** *flash_fill(uint32_t addr, uint8_t val, size_t size)*: Fill the flash memory cells with the value val starting from the address addr for size value;
- ***static bool*** *flash_zero(uint32_t addr, size_t size)*: Fill the flash memory cells with the value 0 starting from the address addr for size value;
- ***static bool*** *flash_program(uint32_t addr, const uint8_t* data, size_t size)*: Write the values contained in *data starting from the address addr for size value;
- ***static bool*** *flash_erase(uint32_t sector)*: Erase the selected sector of the memory;
- ***static bool*** *flash_swap()*: Swap the sector

11 Common

The se3_common.c file contains all the data structures and functions that are shared among all the components. It implies the developed Debug functions.

WARNING: The developed functions were tested and validated by using a **16 GB FAT32** SD card.

11.1 Common Data Structures

- ***SE3_SERIAL*** *serial*: Indicates whether the serial number has been set (by FACTORY_INIT);
- ***uint8_t*** *se3_magic[SE3_MAGIC_SIZE]*: It contains the magic sequence;

11.2 Common Functions

- ***uint16_t*** *se3_req_len_data(uint16_t len_data_and_headers)*: Compute length of data in a request in terms of SE3_COMM_BLOCK blocks;
- ***uint16_t*** *se3_req_len_data_and_headers(uint16_t len_data)*: Compute length of data in a request accounting for headers;
- ***uint16_t*** *se3_resp_len_data(uint16_t len_data_and_headers)*: Compute length of data in a response in terms of SE3_COMM_BLOCK blocks;
- ***uint16_t*** *se3_resp_len_data_and_headers(uint16_t len_data)*: Compute length of data in a response accounting for headers;
- ***uint16_t*** *se3_nblocks(uint16_t len)*: Compute number of SE3_COMM_BLOCK blocks, given length in Bytes;

11.2.1 Debug Tool Functions

In the following, the developed functions usage and a simple practical example are presented:

- ***bool*** *se3_create_log_file()*: Create the user-accessible file on the SD card, called *se3_trace_log.txt* used as trace log buffer. For sake of simplicity, the buffer will have a fixed (oversized) dimension, able to acquire about ten thousands debug strings;
- ***bool*** *se3_debug_sd_flush(uint32_t start_address, uint32_t end_address)*: flushes the data contained in the addresses between *start_address* and *end_address*; it's also called inside *se3_create_log_file()* to erase the content of the previous trace file;
- ***char**** *se3_debug_create_string(char* string)*: prepares the input string to be written in the SD card; the output string will be passed to the *se3_write_trace(char* buf, uint32_t blk_addr)* function;
- ***bool*** *se3_write_trace(char* buf, uint32_t blk_addr)*: writes the string buffer *buf* in the address *blk_addr* of the SD card. In the developed code, a global variable, *debug_address*, is used to write the strings one after the other following a time line, in order to keep track of the temporal dependencies among the function calls.

```
1 //creates the se3_trace_log.txt file and flushes it
2 se3_create_log_file();
3
4 /* write string on SD card on address 'debug_address' and updates the
     address for the next write operation */
5 se3_write_trace(se3_debug_create_string("\nCiao!\0"), debug_address++));
6
7 //Flushes the first 100 strings of the previous debug execution
8 se3_debug_sd_flush(DATA_BASE_ADDRESS, DATA_BASE_ADDRESS+100);
```