Libsmartcard Documentation

Release 0.7.0

ANSSI

May 17, 2019

CONTENTS

Contents

This library is a wrapper for smart cards (either contact or contactless) communication.

It handles APDU sending to the cards, and responses received from the card, whatever the lower layer is (ISO7816 for contact card, ISO14443 for contactless cards, . . .).

For now, only contact cards are supported, the ISO14443 is lacking a driver support that is a future work.

CHAPTER

ONE

API

The API exposed by the library is quite simple and mainly exposes:

- Initialization functions to configure a smart card connection
- Primitives to send APDUs (and get back a response)
- Primitives for pretty printing
- Primitives for detecting card insertion

Note: If these APIs are very similar to the ones exposed by the ISO7816 library this is not a coincidence since they are mainly wrappers to them

1.1 Card abstraction

A card is abstracted with the following structure:

```
typedef union {
      SC_ATR atr;
      SC_NFC nfc;
} SC_Info;
typedef struct {
      /* Card type (contact or contactless) */
      smartcard_types type;
      /* Card information (ATR for contact cards, ATQ/ATS for contactless, ... */
      SC_Info info;
      /* the protocol we use */
      uint8_t T_protocol;
} SC_Card;
```

As we can see, this captures the abstraction layer above contact and contactless cards.

1.2 Initialization functions

The initialization functions are the following:

```
int SC_fsm_early_init(sc_map_mode_t map_mode);
int SC_fsm_init(SC_Card *card, uint8_t do_negotiate_pts, uint8_t do_change_baudrate,␣
↪uint8_t do_force_protocol, uint32_t do_force_etu);
```

The `SC_fsm_early_init` simply initializes the lower levels (it is a mere call to the low level ISO7816 library and driver that initializes the necessary hardware IPs such as USART and GPIOs).

The `SC_fsm_init` function is the core function that establishes the contact with the smart card. Its purpose is to get the ATR for contact cards, the ATQ for contactless cards, and optionally negotiate the PSS (when possible). The arguments are the following:

- `SC_Card *card`: the structure that will receive the card abstract representation

The following arguments have a signification only for contact cards for now (contactless cards support is a work in progress):

- `uint8_t do_force_protocol`: does the user want to force the protocol. A value of 0 means no protocol is forced, a value of 1 means T=0 is forced, a value of 2 means T=1 is forced
- `uint8_t do_negotiate_pts`: effectively performs the PTS protocol negotiation if set to non zero
- `uint8_t do_change_baudrate`: effectively modifies the baudrate with the card if set to non zero
- `uint32_t do_force_etu`: forces a target ETU if set to non zero. If the provided ETU is not achievable, we use a **best fit** algorithm to get the closest ETU lower than the one asked by the user

This function returns 0 on success, and non zero on error.

Note: The following elements mainly concern contact cards. Contacless cards are work in progress

---

A default usage of `SC_fsm_init` is:

```
SC_Card card;
if(SC_fsm_init(&card, 0, 0, 0, 0)){
  goto err;
}
```

This initialization establishes a communication channel with the smart card if present, or waits its presence if not, and does not negotiate anything. The ETU stays at the default value of 372 ETU (default value as defined by the standard) and the protocol is the preferred one provided by the card ATR (or T=0 as standardized default if no preferred protocol is given).

A more advanced usage can be:

```
#include "libsmartcard.h"
SC_Card card;
if(SC_fsm_init(&card, 1, 1, 2, 64)){
  goto err;
}
```

This call asks for a PSS negotiation, asks for a baud rate change, forces the T=1 protocol and asks for a 64 ETU value.

The user can also perform a negotiation attempt and then fallback to default:

```
#include "libsmartcard.h"
SC_Card card;
if(SC_fsm_init(&card, 1, 1, 2, 64)){
  if(SC_fsm_init(&card, 0, 0, 0, 0)){
    goto err;
  }
}
```

Note: Forcing elements such as the protocol or the ETU heavily depends on the smart card: some values and/or some smart cards are not compatible or supported. This is why it is recommended to fallback to a non negotitated `SC_fsm_init` if the negotiated one fails

When a card communication must be reinitialized/reset, it is advised to wait for some timeouts using the following API:

```
int SC_wait_card_timeout(SC_Card *card);
```

Finally, two APIs are used to explicitly ask the lower level driver to map or unmap the smart card device from the task''s memory space:

```
int SC_fsm_map(void);
int SC_fsm_unmap(void);
```

1.3 Primitives to send APDUs

The library provides a unique API to send an APDU to a smart card and receive its response:

```
int SC_send_APDU(SC_APDU_cmd *apdu, SC_APDU_resp *resp, SC_Card *card);
```

The `apdu` argument is a pointer to an input APDU structure, the `resp` response is a pointer to a response structure that will be filled by the function, the `card` structure is a pointer to an abstract card that has been obtained in the initialization phase with `SC_fsm_init`. The library automatically handles the physical layer (T=0 or T=1, ISO14443) depending on the initialization phase.

The APDU structure is the following:

```
/* An APDU command (handling extended APDU) */
typedef struct
{
      uint8_t cla;  /* Command class */
      uint8_t ins;  /* Instruction */
      uint8_t p1;   /* Parameter 1 */
      uint8_t p2;   /* Parameter 2 */
      uint16_t lc;  /* Length of data field, Lc encoded on 16 bits since it is always
↪< 65535 */
      uint8_t data[APDU_MAX_BUFF_LEN];  /* Data field */
      uint32_t le;   /* Expected return length, encoded on 32 bits since it is <=␣
↪65536 (so we must encode the last value) */
      uint8_t send_le;
} SC_APDU_cmd;
```

The response has the following structure:

```
/* An APDU response */
typedef struct
{
      uint8_t data[APDU_MAX_BUFF_LEN + 2]; /* Data field + 2 bytes for temporaty SW1/
↪SW2 storage */
      uint32_t le; /* Actual return length. It is on an uint32_t because we increment␣
↪it when receiving (this avoids integer overflows). */
      uint8_t sw1; /* Status Word 1 */
```

(continues on next page)

(continued from previous page)

```
      uint8_t sw2; /* Status Word 2 */
} SC_APDU_resp;
```

Sending an APDU and getting back a response is as simple as:

```
#include "libsmartcard.h"
/* Initialize a communication with the card */
SC_Card card;
if(SC_fsm_init(&card, 1, 1, 2, 64)){
  goto err;
}
/* Prepare our APDU and response */
SC_APDU_cmd apdu;
SC_APDU_resp resp;
/* Fill in the APDU we want to send:
 * In this case, we send CLA=00 INS=01 P1=00 P2=00 DATA="000102" (Lc=3) and Le=00
 */
apdu.cla = 0x00; apdu.ins = 0x01; apdu.p1 = apdu.p2 = 0x00;
apdu.lc = 3; apdu.data[0] = 0x00; apdu.data[1] = 0x01; apdu.data[2] = 0x02;
apdu.le = 0x00; apdu.send_le = 1;
/* Send the APDU and get the response */
if(SC_send_APDU(&apdu, &resp, &card)){
  goto err;
}
/* If there is no error, resp is filled with the card response! */
```

The smartcard library also provides two helper functions to help APDU fragmentation on the physical line, which proves helpful when dealing with lower layers protocols (T=0 and T=1 for contact cards, ISO14443, etc.). These helpers are exposed but are mainly for an internal usage of the library:

```
unsigned int SC_APDU_get_encapsulated_apdu_size(SC_APDU_cmd *apdu);
uint8_t SC_APDU_prepare_buffer(SC_APDU_cmd *apdu, uint8_t *buffer, unsigned int i,␣
↪uint8_t block_size, int *ret);
```

1.4 Pretty printing

We have straightforward APIs for pretty printing on the debug console the abstract card, APDUs and responses:

```
void SC_print_Card(SC_Card *card);
void SC_print_APDU(SC_APDU_cmd *apdu);
void SC_print_RESP(SC_APDU_resp *resp);
```

1.5 Card insertion detection

The following API:

```
uint8_t SC_is_smartcard_inserted(SC_Card *card);
```

can be used for polling the smart card presence (returns 0 is card is absent, non zero otherwise).

For asynchronous detection, a callback registration mechanism is also offered through:

```
int SC_register_user_handler_action(SC_Card *card, void (*action)(void));
```

Finally, there is an API to call the lower layers of the libraries/drivers stack when a smart card is detected as lost:

```
void SC_smartcard_lost(void)
```

this function helps the hardware layers to reinitialize and flush elements, and eventually notify other drivers. It should be called when the library indeed detects a smart card loss.

CHAPTER

TWO

FAQ

2.1 Is the smartcard library complete?

No, this library is still a work in progress. More particularly, even though all the abstraction for contact and contactless cards are present, only the contact cards are fully supported because only these have a proper underlying ISO7816-3 driver.

Warning: For now the project does not have an ISO14443 driver: only contact cards (ISO7816-3) are fully operational, contacless cards are future work