Hash driver Documentation

Release 0.9.0

ANSSI

Apr 28, 2020

CONTENTS

Contents

This library is an implementation of the ISO7816-3 standard to communicate with contact smart cards.

It should be hardware independent and makes use of a hardware abstraction layer that must provide low level interactions with the Vcc and RST (Reset) lines, the I/O line (pushing and getting a byte), the clock and baudrate handling, etc.

This library implements the T=0 and T=1 protocols, ATR (Answer To Reset) parsing and the PSS/PTS negotiation. It offers high level functions to send APDUs to and receive Responses from a smart card.

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: Some notions discussed here such as APDU (Application Protocol Data Unit), ETU (Elementary Time Unit), ISO7816 baudrate, T=0 and T=1 are defined in the ISO7816-3 ISO standard. Please refer to it for definitions of these concepts

1.1 Initialization functions

The initialization functions are the following:

```
int SC_iso7816_fsm_early_init(sc_iso7816_map_mode_t map_mode);
int SC_iso7816_fsm_init(SC_ATR *atr, uint8_t *T_protocol, uint8_t do_negotiate_pts,␣
↪uint8_t do_change_baudrate, uint8_t do_force_protocol, uint32_t do_force_etu);
```

The `SC_iso7816_fsm_early_init` simply initializes the lower levels (it is a mere call to the low level ISO7816 driver that initializes the necessary hardware IPs such as USART and GPIOs).

The `SC_iso7816_fsm_init` function is the core function that establishes the contact with the smart card. Its purpose is to get the ATR and optionally negotiate the PSS. The arguments are the following:

- `SC_ATR *atr`: the structure that will receive the ATR from the card
- `uint8_t *T_protocol`: this argument takes as output the negotiated protocol
- `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.

A default usage of `SC_iso7816_fsm_init` is:

```
#include "libiso7816.h"
uint8_t T;
SC_ATR atr;
if(SC_iso7816_fsm_init(&atr, &T, 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 "libiso7816.h"
uint8_t T;
SC_ATR atr;
if(SC_iso7816_fsm_init(&atr, &T, 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 "libiso7816.h"
uint8_t T;
SC_ATR atr;
if(SC_iso7816_fsm_init(&atr, &T, 1, 1, 2, 64)){
  if(SC_iso7816_fsm_init(&atr, &T, 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_iso7816_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_iso7816_wait_card_timeout(SC_ATR *atr, uint8_t T_protocol);
```

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_iso7816_fsm_map(void);
int SC_iso7816_fsm_unmap(void);
```

1.2 Primitives to send APDUs

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

```
int SC_iso7816_send_APDU(SC_APDU_cmd *apdu, SC_APDU_resp *resp, SC_ATR *atr, uint8_t␣
↪T_protocol);
```

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 `atr` structure is a pointer to an ATR that has been obtained in the initialization phase with `SC_iso7816_fsm_init`. The library automatically handles the physical layer depending on the asked `T_protocol` argument (T=0 or T=1).

Warning: The user can force any protocol when calling `SC_iso7816_send_APDU`. However, consistency should be observed between the protocol negotiated during the initialization phase and the one used when sending APDUs!

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

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

```
#include "libiso7816.h"
/* Initialize a communication with the card */
uint8_t T;
SC_ATR atr;
if(SC_iso7816_fsm_init(&atr, &T, 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:
```

(continues on next page)

(continued from previous page)

```
 * 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_iso7816_send_APDU(&apdu, &resp, &atr, T)){
  goto err;
}
/* If there is no error, resp is filled with the card response! */
```

1.3 Pretty printing

We have straightforward API for pretty printing on the debug console the ATR:

```
void SC_iso7816_print_ATR(SC_ATR *atr);
```

1.4 Card insertion detection

The following API:

```
uint8_t SC_iso7816_is_smartcard_inserted(void);
```

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:

```
void SC_iso7816_register_user_handler_action(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_iso7816_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 ISO7816 library self-contained?

The automatons of the ISO7816-3 are self-contained in the library, but the library relies on a hardware abstraction layer to handle the ISO7816 pins (I/O, Vcc, RST, clock).