Cardpeek Reference Manual - v0.5

March 22, 2010

Presentation

CARDPEEK is a program that reads the contents of smart cards. This GNU/Linux tool has a GTK GUI and can be extended with the LUA programming language. It requires a PCSC card reader to communicate with a smart card.

Smartcards are becoming ubiquitous in our everyday life. We use them for payment, transport, in mobile telephones and many other applications. These cards often contain a lot of personal information such as, for example, our last purchases or our last journeys in public transport.

CARDPEEK's goal is to allow you to access all this personal information. As such, you can be better informed about the data that is collected about you.

CARDPEEK explores ISO 7816 compliant smart cards and represents their content in an organized tree format that roughly follows the structure it has inside the card, which is also similar to a classical file-system structure.

In this version, this tool is capable of reading the contents of the following types of cards:

- EMV "chip and PIN" bank cards used in many countries throughout the world;
- Electronic/Biometric passports, which have an embedded contacless chip (a contactless reader is required);
- *Navigo* transport cards used in Paris and other *Calypso* cards used elsewhere;
- Moneo, the French electronic purse (with limited data interpretation);
- Vitale 2, the French health card.

Some important card types are missing such as the SIM card used in mobile phones. However, this application can be modified and extended easily to your needs with the embedded LUA scripting language.

For more information on the LUA project see `http://www.lua.org/`.

This software has been tested with traditional PCSC card readers (such as the Gemalto^{TM}PC TWIN) as well as contactless or dual-interface PCSC readers (such as

the Omnikey^{TM}5321). Support for the ACG^{TM}Multi-ISO contactless card reader is still experimental but has been reported to work well for traditional ISO 7816 compliant cards.

Contents

Chapter 1

Installation

CARDPEEK can be compiled from source using the provided `Makefile`. It has currently been tested under *Linux Debian version 5* (both i386 and amd64).

It has been reported to compile and work under *FreeBSD* as well, with a minor tweak.

Note that, in most cases, cryptography export control restrictions do not apply to the source code of CARDPEEK: all cryptographic operations are implemented in the external `openssl` library.

1.1 Compiling and installing

Instructions:

1. Make sure you have the following development packages installed:
   - libgtk 2.0 (`http://www.gtk.org`)
   - liblua 5.1 (`http://www.lua.org`)
   - libpcsclite (`http://pcsclite.alioth.debian.org/`)
   - libssl (`http://www.openssl.org/`)

   (Note: On a Debian system, these packages are all available through `apt`/`aptitude`.)
2. Unpack the source if needed and change directory to the source directory.
3. Type[1] '`make`'
4. The binary executable is '`cardpeek`', use it directly or copy it to the desired location.

[1]Note for FreeBSD: before executing `make`, you will need to make a small change in the Makefile included in the source, as described in the comments included in that file.

1.2 Related files and initial setup

The first time CARDPEEK is run, it will attempt to create the `.cardpeek/` directory in your home directory. This is normal.

The `.cardpeek/` directory will contain two elements: `config.lua` and the `scripts/` directory. The `config.lua` allows you to run commands automatically when the program starts (it should become a full fledged 'config file' in the future). The `scripts/` directory contains all the scripts that allow to explore smart cards. Currently it contains 7 LUA files: "`emv.lua`", "`navigo.lua`", "`calypso.lua`", "`moneo.lua`", "`vitale_2.lua`", "`e-passport.lua`" and "`atr.lua`". These files all show up in the 'analyzer' menu of CARDPEEK (without their extension '.lua'). If you add any LUA file to this directory, it will thus also appear in the menu. The `scripts/` directory contains a subdirectory `lib/` that holds a few LUA files containing frequently used commands that are shared among the card processing scripts.

Each time the program runs, it creates a file `.cardpeek.log` in your home directory. This file contains a copy of the messages displayed in the "log" tab of the application (see next chapter).

Chapter 2

Using Cardpeek

2.1 Quick start

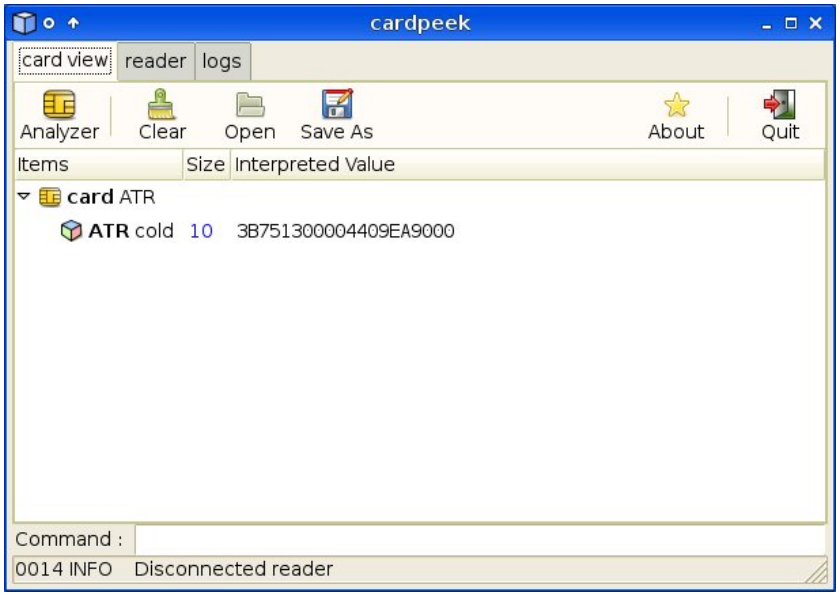
To experiment with CARDPEEK, your may start with your EMV "PIN and chip" smart card for example, by following these steps:

1. Start `cardpeek`.
2. Select your PCSC card reader in the first dialog box.
3. Insert your EMV "PIN and chip" card in the card reader.
4. Select *emv* in the *analyzer* menu. This will run the default *emv* script.
5. View the results in the "card data" tab.

On many bank cards, you will discover a surprising amount of transaction log data (look at the "log data" in the displayed card data).

2.2 User interface

The user interface is divided in four main parts: 3 tabs and a one-line command input field.



Each one of the 3 tabs proposes a different view of card related information:

'card view' shows card data extracted from a card in a structured *tree* form.

'reader' shows raw binary data exchanged between the host PC and the card reader.

'log' displays a journal of application events, mainly usefull for debugging purposes.

2.2.1 card view

The *card view* tab is the central user interface component of CARDPEEK.

It represents the data extracted from a card in a structured *tree* from. This tree structure is initially blank and is entirely constructed by the LUA scripts that are executed (see next chapter). This tree can be saved and loaded in XML format (see 4.9) using the commands in the toolbar.

The *card view* tab offers the following toolbar buttons:

Analyse	Clicking on this button spawns a menu from which an card analysis script can be chosen (see next chapter).
Clear	This button clears the card tree view.
Open	This button allows to load a previously saved card tree representation from an XML file.
Save As	This button allows to save the current card tree representation into an XML file.
About	This button displays a very brief message about CARDPEEK.
Quit	This button quits the application.

The *card view* data is represented in 3 columns. The first column displays the nodes of the card tree view in a hierarchical structure similar to a typical file directory tree browser. The second column displays the size of the node data (most frequently expressed in bytes). Finally, the third column displays the node data itself. The node data can either be represented in "raw" (hexadecimal) form or in a more user friendly "alternative" form (such as a text, or a date). By default, the tree view will display node data in a user friendly "alternative" format if it exists. By clicking on the column title, it is possible to switch between both "raw" and "alternative" data representations.

When node data is over 128 bytes in size, it is truncated for display in the card view window. This indicated by appending **[...]** to the truncated data. By double-clicking on the data, it is possible to switch to the full representation of the node data.

2.2.2 The reader tab

The reader tab displays the raw binary data exchanges between the card reader and the card itself. This data is mainly composed of command APDUs and card responses. Command APDUs are represented by a single block of data, while card responses contain two elements: a card status word and card response data.

One interesting feature of the card reader tab is the ability to save the APDU/response exchanges between the card reader and the card in a file that can later be used to emulate the card. Once card this data is saved in a file (with the .clf extension) and placed in the `.cardpeek/log/` folder, it will appear as a choice in the card reader selection window that appear when `cardpeek` is launched. The name of the file will be prefixed by "`emulator://`" in the card selection window. Selecting such a card data file allows to re-run the script on the previously recorded APDU/response data instead of a real card inserted in the card reader. This is very useful for testing and debugging card scripts without relying on a real card inserted in the card reader.

The *reader* tab offers the following toolbar buttons:

Connect	This button establishes a connexion between the card and the card reader.
Reset	This button performs a warm reset of the card.
Disconnect	This button closes the connexion between the card and the card reader.
Clear	This button clears the APDU/response data displayed in the window.
Save as	This button allows to save the displayed APDU/response data, either for future examination or to be replayed as an emulation of a real card.

"Connect", "Reset" and "Disconnect" operations are usually automatically done by the card scripts. However, it is occasionally practical to manually force the execution of these commands.

2.2.3 The log tab

The *log* tab keeps track of *messages* emitted by the application or the script being run. These messages are useful for monitoring and for debugging purposes. The last message also appears at the bottom of the screen in the status bar.

2.2.4 The one-line command input field

The one-line *command* input field at the bottom of the window allows to type single LUA commands that will be directly executed by the application. This is useful for testing some ideas quickly or for debugging purposes.

2.3 Card-reader selection

When the program starts, you'll be asked to choose a card reader. This will give you 3 main choices :

1. *Select a PCSC card reader to use:* You may have several of PCSC card readers attached to your computer. Card-readers are usually identified by their name, preceded by `pcsc://`.

2. *Select a file containing previously recorded card APDU/response data:* This allows to emulate a smart card that was previously in the reader, and is quite convenient for script debugging purposes. Each time an APDU is sent to the emulated card, Cardpeek will answer with the previously recorded response data (or return an error if the query is new). Files containing previously recorded APDU/response data are identified by a file name, preceded by `emulator://`.

3. *Select "none":* Selecting `none` is useful if you do not wish to use a card reader at all, for example if you only want to load and examine card data that was previously saved in XML format.

Chapter 3

Card analysis tools

Cardpeek provides several card analysis tools, which all appear in the "Analyzer" menu. These tools are actually "scripts" written in the LUA language, and CARDPEEK allows you to add your own scripts easily. As described in chapter 6, these scripts are provided WITHOUT ANY WARRANTY.

3.1 atr

Overview: This script simply prints the ATR (Answer To Reset) of the card.

Notes: In the future this script will be enhanced with a detailed analysis of the ATR.

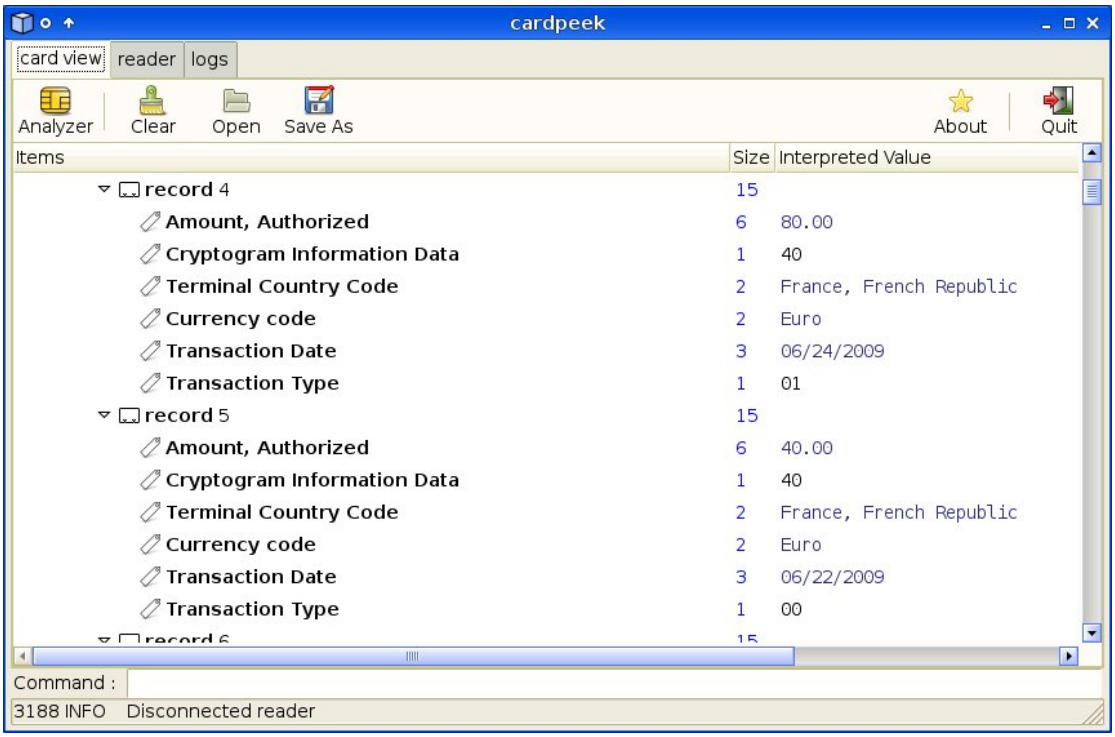
3.2 calypso

Overview: This script provides an analysis of Calypso public transport cards used in many cities.

Notes: The following calypso cards have been reported to work with this script: Navigo/Paris, MOBIB/Brussels (partial support), and Korrigo/Rennes. The Navigo card has a seperate dedicated script which provides more information.

You will notice that these transport cards keep a "event log" describing at least 3 of the last stations/stops you have been through. This "event log", which could pose a privacy risk, is not protected by any access control means and is freely readable.

3.3 emv

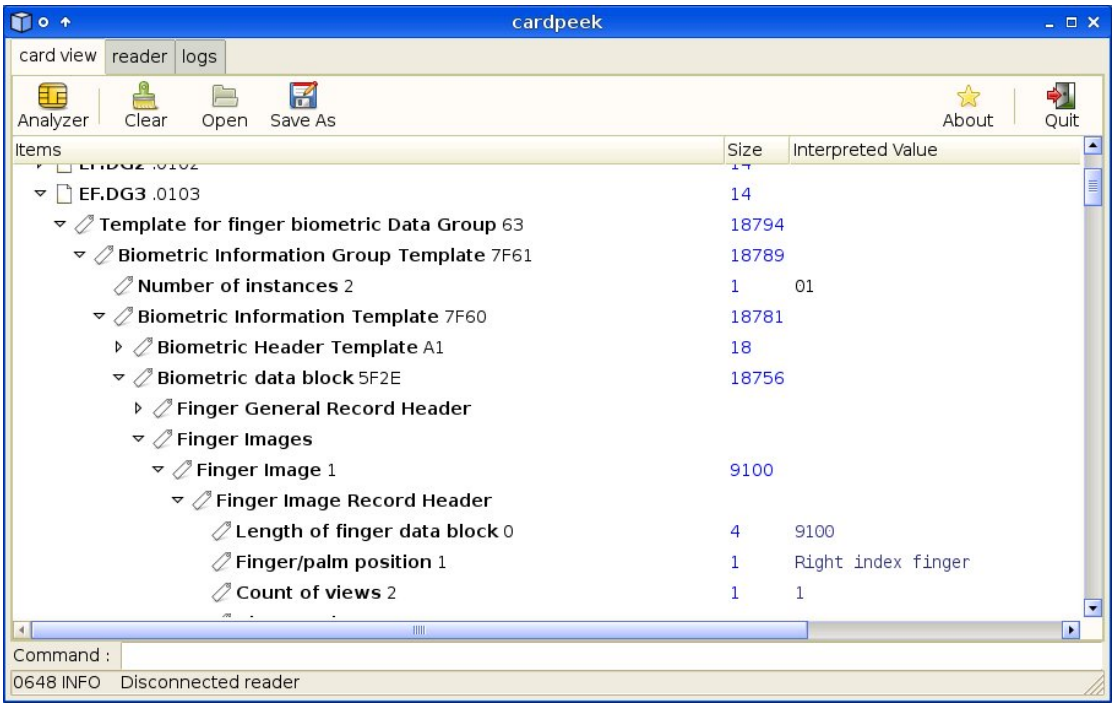


Overview: This script provides an analysis of EMV banking cards used across the world.

Notes: This script will ask you if you want to issue a *Get Processing Option* (GPO) command for each application on the card. Since some cards have several applications (e.g. a national and an international application), this question may be asked twice or more. This command is needed to allow access to some information in the card. Issuing this command will also increase an internal counter inside the card (the ATC).

You will notice that many of these bank cards keep a "transaction log" of the last transactions you have made with your card. Some banks cards keep way over a hundred transactions that are freely readable, which brings up some privacy issues.

3.4 e-passport



Overview: This script provides an analysis of data in a electronic/biometric passport, through a contactless interface.

Notes: This script implements the BAC (Basic Access Control) secure access algorithm to access data in the passport. It will not be able to access data protected with the EAC (Enhances Access Control) algorithm. When the script starts, you will be required to input the lower part of the MRZ (Machine Readable Zone) data on the

passport. This data is needed to compute the cryptographic keys used in the BAC algorithm.

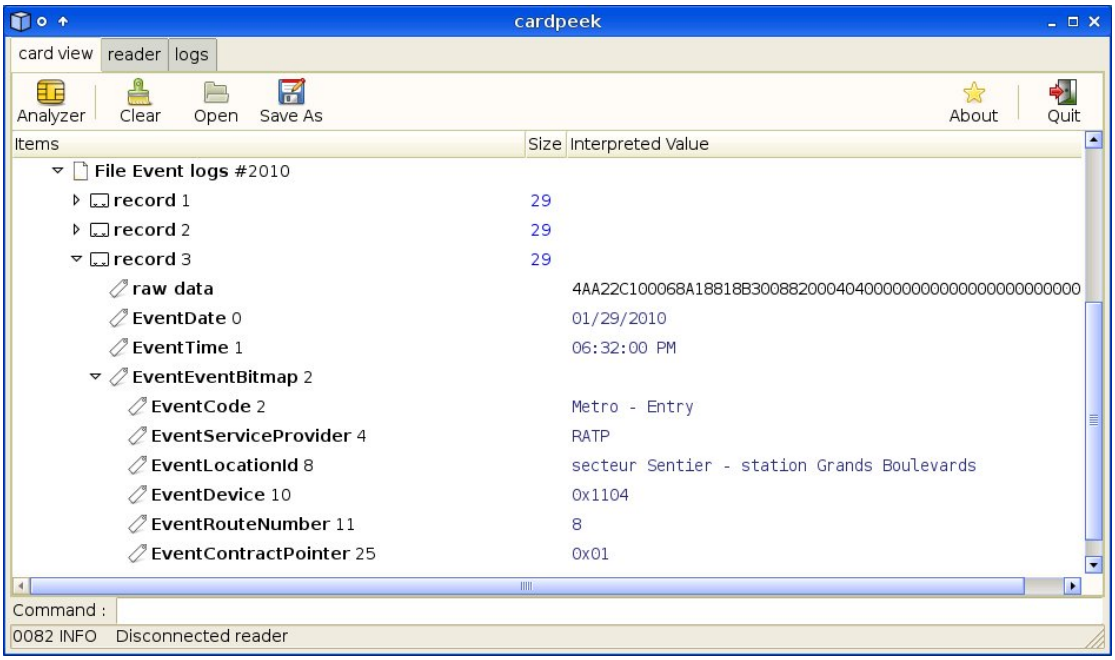
This scripts attempts to parse biometric facial and fingerprint image data.

3.5 moneo

Overview: This scripts provide a (limited) analysis of MONEO electronic purse cards used in France.

Notes: The provided output is very "raw".

3.6 navigo



Overview: This script is a specialized version of the `calypso` script, specifically targeted for the Navigo cards used in Paris. It provides enhanced "event log" analysis notably with subway/train station names, as illustrated in the example above. It has been successfully tested on *Navigo Découverte*, *Navigo* and *Navigo Intégrale* cards.

Notes: You must use the contact interface to read a Navigo card, because they cannot be read with a normal contactless card reader (these cards use a specific protocol that is not fully compatible with ISO 14443 B). See the section about *calypso* for other information.

3.7 vitale 2

Overview: This script provides an analysis of the second generation French health card called "Vitale 2".

Notes: This analysis is based on a lot of guesswork and needs further testing. Some zones, notably the one containing the cardholder's photography, seem protected: this is a good design choice in terms of privacy protection.

3.8 Adding your own scripts to Cardpeek

Adding or modifying a script in CARDPEEK is easy: simply add or modify a script in the `$HOME/.cardpeek/scripts/` directory.

If you want to go further and make a script permanently part of the source code of CARDPEEK, you should follow these additional steps:

1. Go to the directory containing the source code of CARDPEEK.
2. Execute the `update_dot_cardpeek_dir.sh` script (e.g. type "`. update_do_cardpeek.sh`")
3. Run `make` to rebuild CARDPEEK.

Chapter 4

Script language description

The individual scripts that allow to process different types of smart cards are located in your `$HOME/.cardpeek/scripts/` directory. These scripts are written in LUA, a programming language which shares some similarities with Pascal and Javascript. To allow LUA scripts to communicate with smart cards and to manipulate card data, the LUA language was extended with custom libraries. This section first starts with a brief example of the script language and then describes the library extensions.

4.1 Introductory examples

Here's a short LUA script that demonstrates how to get and print the ATR (Answer To Reset) of a card to the console.

```
card.connect()
print(card.last_atr())
card.disconnect()
```

Next, here's (a slightly longer) script that performs a similar task, while presenting the results using CARDPEEK's user interface instead of the console.

```
card.connect()

atr = card.last_atr()

if atr then
   mycard = ui.tree_add_node(nil,"card")
   ref = ui.tree_add_node(mycard,"ATR","cold")
   ui.tree_set_value(ref,atr)
```

```
end

card.diconnect()
```

The fist command `card.connect()` powers-up the card in the card reader and prepares the card for communication. Next `card.last_atr()` returns the ATR of the card. If the value of the ATR is non-nil, the script creates a node called "card" (with `ui.tree_add_node()`). This node will appear at the root of the card data tree-view. A child node called "ATR" is added to the root "card" node. The hexadecimal value of the ATR is associated with the child node. Finally, the card is powered down with the `card.disconnect()` function.

The final output of the script should have roughly the following structure:

▷ **card**
  └**ATR** cold `3B6F0000805A0803040002002531F405909000`

The example above is equivalent to the "atr" script provided with CARDPEEK. The LUA language is easy to learn and we refer the reader to `http://www.lua.org/` for more information.

4.2 the `bit` library

Since LUA does not have native bit manipulation functions, the following functions have been added.

4.2.1 bit.AND

SYNOPSIS

`bit.AND(A,B)`

DESCRIPTION

Compute the binary operation `A` *and* `B`.

4.2.2 bit.OR

SYNOPSIS

`bit.OR(A,B)`

DESCRIPTION

Compute the binary operation `A` *or* `B`.

4.2.3 bit.XOR

Synopsis

`bit.XOR(A,B)`

Description

Compute the binary operation `A` *xor* `B`.

4.2.4 bit.SHL

Synopsis

`bit.SHL(A,B)`

Description

Shift the bits of A by B positions to the left. This is equivalent to computing $\texttt{A} \times 2^{\texttt{B}}$.

4.2.5 bit.SHR

Synopsis

`bit.SHR(A,B)`

Description

Shift the bits of A by B positions to the right. This is equivalent to computing $\texttt{A} / 2^{\texttt{B}}$

4.3 The `bytes` library

The `bytes` library provides a new opaque type to LUA: a *bytestring*, which is used to represent an array of binary elements.

Bytestrings are mainly used to represent binary data exchanged with the card reader in the application.

The elements in the array are most commonly (8 bit) bytes, but it is also possible to construct arrays of (4 bit) half-bytes or arrays of individual bits. All elements in a bytestring have the same size (8, 4 or 1), which is referred as the "*width*" of the bytestring. The width of each element is specified when the array is created with the function `bytes.new()` described in this section. A function to convert between bytestrings of different widths is also provided.

Individual elements in a bytestring array can be accessed the same way traditional arrays are accessed in LUA. Thus, if `BS` is a bytestring the following expressions are valid:

```
BS[0]=1
print(BS[0])
```

Contrary to the LUA tradition, the first index in a bytestring is 0 (instead of 1). The number of elements in a bytestring is indicated by prefixing the bytestring with the "#" operator, just as with an array (e.g. #BS).

Bytestrings cannot be copied like arrays with a simple assignment using the "=" operator, the bytes.assign() function or the bytes.clone() function must be used instead.

The functions of the bytes library are next described.

4.3.1 Operators on bytestrings

The operators that can be used on bytestrings are "..", "==", "∼=" and "#"

Synopsis

```
A..B
A==B
A∼=B
#A
```

Description

The ".." operator creates a new bytestring by concatenating two bytestrings together. The concatenation operator also works if one of the operands is a string or a number, by converting it to a bytestring first, following the rules described in the bytes.assign() function. A..B is equivalent to bytes.concat(A,B).

The "==" and "∼=" operators allow to compare two bytestrings for equality or non-equality respectively. To be equal, two bytestrings must have the same width and the same elements in the same order.

Finally the "#" operator returns the number of elements in a bytestring.

4.3.2 bytes.append

Synopsis

bytes.append(BS, $\texttt{value}_0$ [, $\texttt{value}_1$, ..., $\texttt{value}_n$])

Description

Append a value to BS.

The appended value is composed of $\texttt{value}_0$, optionally concatenated with any additional value $\texttt{value}_1$, ..., $\texttt{value}_n$ (from left to right).

This function is equivalent to `bytes.assign(BS, BS, value0 [, value1, ..., valuen])`. See `bytes.assign()` for further details.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.3 bytes.assign

Synopsis

`bytes.assign(BS, value0 [, value1, ..., valuen])`

Description

Assigns a value to `BS`.

The assigned value is composed of `value0`, optionally concatenated with any additional value `value1`, ..., `valuen` (from left to right).

Each `valuei` can be either a bytestring, a string or a number. If `valuei` is a bytestring, each element of `valuei` is appended to `BS`, without any conversion.

If `valuei` is a string, it is interpreted as a text representation of a bytestring (as returned by the `tostring()` operator). This string representation is interpreted by taking into consideration the width of elements of `BS` and is appended to `BS`.

If `valuei` is a number, it is converted into a single bytestring element and appended to `BS`.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.4 bytes.clone

Synopsis

`bytes.clone(BS)`

Description

Creates and returns a copy of `BS`.

Return value

This function returns `nil` if it fails.

4.3.5 bytes.concat

Synopsis

`bytes.concat(value`$_0$`, value`$_1$ `[,value`$_2$ `, ..., value`$_n$`])`

Description

Returns the concatenation of `value`$_0$, ..., `value`$_n$ (from left to right).

For the rules governing the processing of `value`$_0$ ... `value`$_n$, see the `bytes.assign()` function above.

Return value

This function returns a bytestring upon success and `nil` otherwise.

4.3.6 bytes.convert

Synopsis

`bytes.convert(w,BS)`

Description

Converts `BS` to a new bytestring where each element has a width `w`.

Depending on `w`, the elements in the converted bytestring are obtained by either splitting elements of `BS` into several smaller elements in the new bytestring or by grouping several elements of `BS` into a single element in the new bytestring.

If the conversion requires splitting elements of `BS`, then the original elements will be split with the most significant bit(s) first (the most significant bits of each original element of `BS` will have a lower index than the least significant bits).

If the conversion requires grouping elements together, `BS` is will first be right-padded with zeros to a size that is a multiple of `w`. Next, new elements are formed by considering elements of `BS` with a lower index as more significant than elements with a higher index.

Return value

This function returns a new bytestring upon success and `nil` otherwise.

4.3.7 bytes.insert

Synopsis

`bytes.insert(BS, pos, value`$_0$ `[, value`$_1$`, ..., value`$_n$`])`

Description

Inserts a value in `BS` at index `pos`.

The elements in `BS` of index 0 to $pos - 1$ will remain untouched. The elements in `BS` of index `pos` to `#BS` are pushed to the right to make room for the inserted value.

The inserted value is composed of $\texttt{value}_0$, optionally concatenated with any additional value $\texttt{value}_1$, ..., $\texttt{value}_n$ (from left to right).

For the rules governing the processing of $\texttt{value}_0$ ... $\texttt{value}_n$, see the `bytes.assign()` function above.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.8 bytes.invert

Synopsis

```
bytes.invert(BS)
```

Description

Reverses the order of elements in `BS`.

If `BS` has `N` elements then `BS[0]` is swapped with `BS[N-1]`, `BS[1]` is swapped with `BS[N-2]` and so forth until all elements are in reverse order in `BS`.

This function modifies its main argument.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.9 bytes.is_printable

Synopsis

```
bytes.is_printable(BS)
```

Description

Returns `true` if all elements in `BS` can be converted to printable ascii characters, and `false` otherwise.

Return value

This function always returns `false` if the width of `BS` is not 8 (elements of width 4 or 1 are not printable ascii values).

4.3.10 bytes.maxn

Synopsis

`bytes.maxn(BS)`

Description

Returns the last index in `BS` (equivalent to $\#BS - 1$).

Return value

This function returns `nil` if `BS` is empty.

4.3.11 bytes.new

Synopsis

`bytes.new(width [,value`$_0$`, value`$_1$`, ..., value`$_n$`])`

Description

Creates a new bytestring, where each element is `width` bits in size. `width` can be either 8, 4 or 1.

A value can optionally be assigned to the bytestring by specifying one or several values `value`$_0$, `value`$_1$, ..., `value`$_n$ that will be concatenated together to form the content of the bytestring. See the function `bytes.assign()` for more details.

Return value

This function returns a bytestring upon success and `nil` otherwise.

4.3.12 bytes.pad_left

Synopsis

`bytes.pad_left(BS, length, value)`

Description

Pads `BS` on the left with the element `value` until the number of elements of `BS` reaches a multiple of `length`.

If the size of `BS` is already a multiple of `length`, `BS` is left untouched.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.13 bytes.pad_right

Synopsis

```
bytes.pad_right(BS, length, value)
```

Description

Pads `BS` on the right with the element `value` until the number of elements of `BS` reaches a multiple of `length`.

If the size of `BS` is already a multiple of `length`, `BS` is left untouched.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.14 bytes.remove

Synopsis

```
bytes.remove(BS, start [,end])
```

Description

Deletes a part of `BS`.

Removes all elements of `BS` that have an *index* that verifies *index* $\geq$ `start` and *index* $\leq$ `end`.

The elements in `BS` are re-indexed: `BS[end+1]` becomes `BS[start]`, `BS[end+2]` becomes `BS[start+1]`, and so forth.

If `end` is not specified it will default to the last index of `BS`. `start` and `end` may be negative to refer to the position of an element by starting from the end of the bytetring as described in `bytes.sub()`.

This function modifies its main argument `BS`.

Return value

This function returns `true` upon success and `false` otherwise.

4.3.15 bytes.sub

Synopsis

```
bytes.sub(BS, start [,end])
```

Description

Returns a copy of a substring from `BS`.

The returned value represents a bytestring containing a copy of all the elements of `BS` that have an index that verifies $index \geq$ `start` and $index \leq$ `end`. If `end` is not specified it will default to the last index of `BS`. If `start` (or `end`) is negative, it will be replaced by `#BS+start` (or `#BS+end` resp.).

Return value

This function returns a bytestring upon success and `nil` otherwise.

4.3.16 bytes.tonumber

Synopsis

```
bytes.tonumber(BS)
```

Description

Converts the bytestring `BS` to a the unsigned decimal value of `BS`.

This conversion considers `BS[0]` as the most significant element of `BS`, and `BS[#BS-1]` as the least significant.

Return value

This function returns a number.

4.3.17 bytes.toprintable

Synopsis

```
bytes.toprintable(BS)
```

Description

Converts each element in `BS` into an ascii character and returns the resulting string.

If an element in `BS` cannot be converted to a printable character it is replaced by the character "?".

If `BS` is empty, the resulting string is also empty.

Return value

A string.

4.3.18 bytes.width

Synopsis

```
bytes.width(BS)
```

DESCRIPTION

Return the width of the elements in `BS`.

RETURN VALUE

This function may return 1, 4 or 8.

4.4 The `asn1` library

The ASN1 library allows to manipulate ASN1 TLV bytestrings following the DER/BER encoding rules (Distinguished/Basic Encoding Rules). These bytestrings must be 8 bit wide.

The library provides the following functions.

4.4.1 asn1.enable_single_byte_length

SYNOPSIS

```
asn1.enable_single_byte_length(enable)
```

DESCRIPTION

This function is only used in rare cases with erroneous card implementations.

If `enable` = *true* the behavior of TLV decoding functions (such as `bytes.tlv_split()`) are modified by forcing the ASN1 length to be 1 byte long.

This means that even if the first byte of the encoded length is greater than `0x80` it will be interpreted as the length of the TLV value.

RETURN VALUE

None.

4.4.2 asn1.join

SYNOPSIS

```
asn1.join(tag, val [,extra])
```

DESCRIPTION

Performs the opposite of `bytes.tlv_split`: creates a bytestring representing the ASN1 DER encoding of the TLV {`tag`, *len* , `val`} where *len*=`#val` and appends `extra` to the result.

`tag` is positive integer number, `val` is a bytestring and `extra` is a bytestring or `nil`.

Return value

This function returns a bytestring.

4.4.3 asn1.split

Synopsis

```
asn1.split(str)
```

Description

Parses the beginning of the bytestring `str` according to ASN1 BER TLV encoding rules, and extracts a tag `T` and a bytestring value `V`.

Return value

The function returns 3 elements {`T`, `V`, *extra*}, where *extra* is an optional bytestring representing the remaining part of `str` that was not parsed or `nil` if no data remains.

If this function fails it returns a triplet of `nil` values.

4.4.4 asn1.split_length

Synopsis

```
asn1.split_length(str)
```

Description

Parses the beginning of the bytestring `str` according to ASN1 BER and extracts a length `L`.

Return value

The function returns {`L`, *extra*}, where *extra* is an optional bytestring representing the remaining part of `str` that was not parsed or `nil` if no data remains.

If this function fails it returns a pair of `nil` values.

4.4.5 asn1.split_tag

Synopsis

```
asn1.split_tag(str)
```

Description

Parses the beginning of the bytestring `str` according to ASN1 BER and extracts a tag `T`.

Return value

The function returns {`L`, *extra*}, where *extra* is an optional bytestring representing the remaining part of `str` that was not parsed or `nil` if no data remains.

If this function fails it returns a pair of `nil` values.

4.5 The `card` library

The `card` library is used to communicate with a smart card in a card reader.

Cardpeek internally defines a minimal set of card functions in the `card` library. Some additional extensions to the `card` library are written in LUA and can be found in the file `$HOME/.cardpeek/scripts/lib/apdu.lua`, which should be loaded automatically when cardpeek starts.

This library contains the following functions.

4.5.1 card.connect

Synopsis

```
card.connect()
```

Description

Connect to the card currently inserted in the selected smart card reader.

This command is used at the start of most smart card scripts.

Return value

This function returns `true` upon success, and `false` otherwise.

4.5.2 card.disconnect

Synopsis

```
card.disconnect()
```

Description

Disconnect the card currently inserted in the selected smart card reader.

This command concludes most smart card scripts.

Return value

This function returns `true` upon success, and `false` otherwise.

4.5.3 card.get_data

Synopsis

```
card.get_data(id [, length_expected])
```

Description

Execute the GET_DATA command from ISO 7816-4 where:

- `id` is the tag number of the value to read from the card.
- `length_expected` is an optional value specifying the length of the resulting expected result (defaults to 0, which means 256 bytes).

The value of "CLA" in the command sent to the card is defined by by the variable `card.CLA`.

This function is implemented in `apdu.lua`.

Return value

The card status word and response data, as described in `card.send` (section 4.5.9).

4.5.4 card.last_atr

Synopsis

```
card.last_atr()
```

Description

Returns a bytestring representing the last ATR (Answer To Reset) returned by the card.

Return value

This function returns a bytestring.

4.5.5 card.make_file_path

Synopsis

```
card.make_file_path(path)
```

Description

This function is designed to be a helper function for the implementation of `card.select`.

It converts a human readable path string (representing a file location in a smart card) into a format that is compatible with the SELECT_FILE command from ISO 7816-4.

This function parses the string `path` and returns a pair of values {`path_binary`, `path_type`} where:

- `path_binary` is a bytestring representing the encoded binary value of `path`, and
- `path_type` is a number describing the path type (i.e. a relative path, an AID, ...)

The general rules needed to form a path string can be summarized as follows:

- A file ID is represented by 4 hexadecimal digits (however, there is an exception for ADFs that can also be represented by their AID, which requires 10 to 32 hexadecimal digits, or 5 to 16 bytes).
- If `path` starts with the '`#`' character, the file is selected directly by its unique ID or AID.
- If `path` starts with the '`.`' character, the file is selected relatively to the current DF or EF.
- Files can also be selected by specifying a relative or absolute path, where each element in the path is represented by a 4 digit file ID separated by the '`/`' character:
  - If `path` starts with '`/`' the file is selected by its full path (excluding the MF).
  - If `path` starts with '`./`' the file is selected by its relative path (excluding the current DF).

The next table describes the format of the string `path` and how it is interpreted more precisely. In this table, as a convention, hexadecimal characters are represented with the character '`h`' and repeated elements are summarized by writing "[...]".

`path` format	interpretation	`path_type`
`#`	Directly select the MF (equivalent to `#3F00`)	0
`#hhhh`	Directly select the file with ID=`hhhh`	0
`#hhhhhh`[...]`hh`	Directly select the DF with AID=`hhhhhh`[...]`hh`	4
`.hhhh`	Under the current DF, select the file with ID=`hhhh`	1
`.hhhh/`	Under the current DF, select the DF with ID=`hhhh`	2
`..`	Select the parent of the current EF or DF.	3
`./hhhh/hhhh/hh`[...]	Select a file using a relative path from the current DF. All intermediary DF's are represented by their file ID separated by the '`/`' character.	9
`/hhhh/hhhh/hh`[...]	Select a file with an absolute path from the MF (the MF is omitted) All intermediary DF's are represented by their file ID separated by the '`/`' character.	8

The resulting bytestring `path_binary` is simply produced from the concatenation of the hexadecimal values in `path` (represented by '`h`' in the table above.)

Return value

Upon success this function returns a pair of values consisting of a bytestring and a number. Upon failure, this functions returns a pair of `nil` values.

4.5.6 card.read_binary

Synopsis

```
card.read_binary(sfi [, address [, length_expected]])
```

Description

Execute the READ_BINARY command from ISO 7816-4 where:

- `sfi` is a number representing a short file identifier ($1 \leq$ `sfi` ≤ 30) or the string '.' to refer to the currently selected file.
- `address` is an optional start address to read data (defaults to 0).

- `length_expected` is an optional value specifying the length of the resulting expected result (defaults to 0, which means 256 bytes).

The value of "CLA" in the command sent to the card is defined by the LUA variable `card.CLA`.

This function is implemented in `apdu.lua`.

Return value

The card status word and response data, as described in `card.send` (section 4.5.9).

4.5.7 card.read_record

Synopsis

```
card.read_record(sfi, r, [, length_expected])
```

Description

Execute the READ_RECORD command from ISO 7816-4 where:

- `sfi` is a number representing a short file identifier ($1 \leq$ `sfi` ≤ 30) or the string '.' to refer to the currently selected file.
- `r` is the record number to read.
- `length_expected` is an optional value specifying the length of the resulting expected result (defaults to 0, which means 256 bytes).

The value of "CLA" in the command sent to the card is defined by the LUA variable `card.CLA`.

This function is implemented in `apdu.lua`.

Return value

The card status word and response data, as described in `card.send` (section 4.5.9).

4.5.8 card.select

Synopsis

```
card.select(file_path [, return_what [, length]])
```

Description

Execute the SELECT_FILE command from ISO 7816-4 where:

- `file_path` is string describing the file to select, according to the format described in `card.make_file_path`.
- `return_what` is an optional value describing the expected result, as described in the table below (defaults to 0).
- `length` is an optional value specifying the length of the resulting expected result (defaults to nil).

The following constants have been defined for `return_what` (some can be combined together by addition):

Constant	value
card.SELECT_RETURN_FIRST	0
card.SELECT_RETURN_LAST	1
card.SELECT_RETURN_NEXT	2
card.SELECT_RETURN_PREVIOUS	3
card.SELECT_RETURN_FCI	0
card.SELECT_RETURN_FCP	4
card.SELECT_RETURN_FMD	8

The value of "CLA" in the command sent to the card is defined by by the variable `card.CLA`. The value of "P1" in the command sent to the card corresponds to the file type computed by `card.make_file_path`. The value of "P2" in the command sent to the card corresponds to `return_what`.

This function is implemented in `apdu.lua`.

Return value

The card status word and response data, as described in `card.send` (section 4.5.9).

4.5.9 card.send

Synopsis

```
card.send(APDU)
```

Description

Sends the command `APDU` to the card.

Return value

The function returns a pair of values: a number representing the status word returned by the card (ex. 0x9000) and the response data returned by the card.

Both the command `APDU` and the response data are bytestrings (see the `bytes` library).

4.5.10 card.info

Synopsis

```
card.info()
```

Description

Return detailed information about the state of the card reader.

Return value

This function returns an associative array of ($name \Rightarrow value$) pairs.

4.5.11 card.warm_reset

Synopsis

```
card.warm_reset()
```

Description

Performs a warm reset of the card (reconnects the card currently inserted in the selected smart card reader).

Return value

None

4.6 The `crypto` library

This library proposes a limited number of cryptographic functions. Currently these functions offer mainly DES, Triple-DES, and SHA1 based transformations.

4.6.1 crypto.create_context

Synopsis

```
crypto.create_context(algorithm [,key])
```

Description

This function creates a cryptographic "context" that holds a description of a cryptographic algorithm, along with a (optional) key. The created context is later used as a parameter to other generic functions in the `crypto` library, such as `crypto.encrypt()`, `crypto.mac()`, `crypto.digest()`, ...

The first parameter `algorithm` allows to describe the cryptographic algorithm to be used. It can currently take the following values :

Algorithm	**Description**
`crypto.ALG_DES_ECB`	Simple DES in ECB mode (no IV).
`crypto.ALG_DES_CBC`	Simple DES is CBC mode.
`crypto.ALG_DES2_EDE_ECB`	Triple DES with a double length 112 bit key in ECB mode (no IV).
`crypto.ALG_DES2_EDE_CBC`	Triple DES with a double length 112 bit key in CBC mode.
`crypto.ALG_ISO9797_M3`	ISO 9797 MAC method 3 with a 112 bit key: a simple DES CBC MAC iteration with triple DES on the final block.
`crypto.ALG_SHA1`	The SHA1 digest algorithm.

Some of the previous algorithms only operate on data that has been padded to a reach a proper size, that is usually a multiple of a defined "block size". The value of `algorithm` can be used to specify the padding method that is used, by combining (with the '+' operator) one of the following values to the algorithm previously specified:

Padding method	**Description**
`crypto.PAD_ZERO`	Add 0's if needed to reach block size.
`crypto.PAD_OPT_80_ZERO`	If the size of cleartext is not already a multiple of block size then add one byte `0x80` and then 0's, if needed, to reach block size.
`crypto.PAD_ISO9797_P2`	ISO 9797 padding method 2 (add a mandatory byte `0x80` and pad with optional 0's to reach block size).

The optional bytestring `key` must be used to specify the value of the cryptographic key used for encryption or MAC algorithms.

Return value

This function returns a bytestring representing the created context. Programmers should consider the result as an opaque value and should not modify its content.

4.6.2 crypto.decrypt

Synopsis

```
crypto.decrypt(context, data [, iv])
```

Description

Decrypt the bytestring `data`, using the key and algorithm provided in `context`. When the decryption algorithm requires a initial vector, it must be specified in `iv`. All parameters and the return value are 8 bit wide bytestrings.

Return value

This function returns the decrypted data as a bytestring.

4.6.3 crypto.digest

Synopsis

```
crypto.digest(context, data)
```

Description

Compute the digest (also often called a hash) of `data`, using the algorithm provided in `context`.
All parameters and the return value are 8 bit wide bytestrings.

Return value

This function returns the digest value as a bytestring.

4.6.4 crypto.encrypt

Synopsis

```
crypto.encrypt(context, data [, iv])
```

Description

Encrypt the bytestring `data`, using the key and algorithm provided in `context`. When the encryption algorithm requires a initial vector, it must be specified in `iv`. All parameters and the return value are 8 bit wide bytestrings.

Return value

This function returns the encrypted data as a bytestring.

4.6.5 crypto.mac

Synopsis

```
crypto.mac(context, data)
```

Description

Computes the MAC (Message Authentication Code) of `data`, using the key and algorithm provided in `context`.

All parameters and the return value are 8 bit wide bytestrings.

Return value

This function returns the (untruncated) MAC as a bytestring.

4.7 The `ui` library

The `ui` library allows to control some elements of the user interface of Cardpeek, and in particular the tree structure representing the data extracted from the card.

The functions in this library rely on a value called a '*path*' to identify each node in the card data tree represented on the screen. A *path* is a string that is composed of a list of positive numbers separated by a colon[1]. Each number represents the position (starting at 0) of a node relatively to its parent. The *path* "0:2:1" represents the second child node of the third child node of the first top node, and "0" simply represents the first top node. The detailed value of a *path* is usually not important for the programmer, who normally manipulates it as an opaque value.

The `ui` library functions are described in the following paragraphs.

4.7.1 ui.question

Synopsis

```
ui.question(text, buttons)
```

Description

Asks the user a question requesting him to answer by selecting a response.

[1]For programmers familiar with GTK++ 2.0, these are the same conventions as in the GTK "tree model"

The question is described in the string `text`, while the set of possible answers described in the table `buttons`. Each element in `buttons` is string representing a possible answer.

Return value

Upon success, the function returns the index of the answer selected by the user in the table `buttons` (LUA table indexes are usually numbers greater or equal to 1).

Upon failure the function returns 0.

4.7.2 ui.readline

Synopsis

```
ui.readline(text [,len [,default]])
```

Description

Request the user to enter a text string.

The user's input can optionally be limited to `len` characters and can hold a predefined value `default`.

Return value

The function returns the user's input upon success and `false` otherwise.

4.7.3 ui.tree_add_node

Synopsis

```
ui.tree_add_node(path_to_parent, name, [id [, length [,type]]])
```

Description

Adds a node in the card tree structure.

The new node will be appended to the children of the node identified by the *path* `path_to_parent`. If `path_to_parent` is `nil` the new node will be added at the top level.

`name` describes the type of data that is represented by the node (such as a "file" or a "date of birth" for example).

`id` is an optional identifier that identifies the node uniquely within a context (such as an application "AID" or a "record number").

`length` is an optional value number describing the length of the data element associated to the node.

`type` is an optional informative string that provides additional information describing the type of data represented by the node. This value will affect the choice of the icon

that is associated with the node in the displayed card tree structure.

The following `type` values are associated with a distinct icon: "application", "block", "card", "file" and "record".

Return value

Upon success the node returns a string representing the *path* to the newly created node. If the function fails, it returns `nil`. Once the node is created with this function, data can be associated to it with the `ui.tree_set_value` function.

4.7.4 ui.tree_delete_node

Synopsis

```
ui.tree_delete_node(path_to_node)
```

Description

Deletes the node identified by `path_to_node` as well as all its children.

Return value

The function returns `true` upon success and `false` otherwise.

4.7.5 ui.tree_find_node

Synopsis

```
ui.tree_find_node(origin, name, id)
```

Description

Searches inside the sub-tree of root `origin` for the first node that has name `name` and/or the id `id`.

If `name=nil` or `id=nil` they are ignored in the search.

Return value

If a node is found, the function returns the path to that node otherwise it returns `nil`.

4.7.6 ui.tree_get_alt_value

Synopsis

```
ui.tree_get_alt_value(path_to_node)
```

Description

Returns the alternative value associated with the node identified by `path_to_parent`, or `nil` if no value is associated with the node or if the function fails.

Return value

This function returns a string.

4.7.7 ui.tree_get_node

Synopsis

```
ui.tree_get_node(path_to_node)
```

Description

Returns an array of 5 elements associated to the node identified by `path_to_node`.

These elements are: *name*, *id*, *length*, *comment*, *num_children*. The first 4 elements are the same as the parameters of the function `ui.tree_add_node`, and the last parameter describes the number of children attached to that node in the tree (0 means none).

Return value

Upon success, this function returns an array. If the function fails, it returns `nil`.

4.7.8 ui.tree_get_value

Synopsis

```
ui.tree_get_value(path_to_node)
```

Description

Returns the value associated with the node identified by `path_to_parent`, or `nil` if no value is associated with the node or if the function fails.

Return value

This function returns a bytestring.

4.7.9 ui.tree_load

Synopsis

```
ui.tree_load(file_name)
```

Description

Loads the tree from the XML file `file_name`.

Return value

The function returns `true` upon success and `false` otherwise.

4.7.10 ui.tree_save

Synopsis

```
ui.tree_save(file_name)
```

Description

Saves the tree in XML format inside the file `file_name`.

Return value

The function returns `true` upon success and `false` otherwise.

4.7.11 ui.tree_set_alt_value

Synopsis

```
ui.tree_set_alt_value(path_to_node, val)
```

Description

Associate the alternative string data `val` to the node identified by the *path* `path_to_node`.

The value `val` is a string (not a bytestring) and should be used to provide a more "human friendly" representation of data associated with the node.

Return value

The function returns `true` upon success and `false` otherwise.

4.7.12 ui.tree_set_value

Synopsis

```
ui.tree_set_value(path_to_node, val)
```

Description

Associate the bytestring data `val` to the node identified by the *path* `path_to_node`.

The value `val` is a bytestring as constructed by the `bytes` library functions.

Once a value `val` is associated to a node `path_to_node` it is not possible to add a child node with the function `ui.tree_add_node`.

Calling `ui.tree_set_value()` automatically resets to `nil` any alternative value associated with the node that was previously set with `ui.tree_set_alt_value()`.

Return value

The function returns `true` upon success and `false` otherwise.

4.7.13 ui.tree_to_xml

Synopsis

```
ui.tree_to_xml(path_to_node)
```

Description

Returns an XML representation of the sub-tree that has `path_to_node` as a root. If `path_to_node` is `nil` the representation of the whole tree is returned.

Return value

This function returns a string upon success. If the function fails, it returns `nil`.

4.8 The `log` library

The `log` library contains just one function described below, which allows to print messages in the "log" tab of the application.

4.8.1 log.print

Synopsis

```
log.print(level, text)
```

Description

Prints a message `text` in the console window.

`level` describes the type of message that is printed. `level` can take the following values: `log.INFO`, `log.DEBUG`, `log.WARNING`, or `log.ERROR`.

All messages printed on the screen with this function are also saved in the file "`$HOME/.cardpeek.log`".

Return value

None.

4.9 XML tree view format

The XML format is quite straightforward, as shown in the following example:

```
<?xml version="1.0"?>
<cardtree>
  <node name="card" id="ATR">
    <node name="ATR" id="cold">
       <val>3B6F0000805A0803040002002531F405909000</val>
    </node>
  </node>
</cardtree>
```

The format of the XML tree view file is constructed according to the following rules:

- The root element of the XML structure is `<cardtree>`, which contains one or more `<node>` elements.
- A `<node>` element may either contain one or several `<node>` elements or a single `<val>` element (but not both).
- A `<val>` element may be optionally followed by a `<alt>` element within an node.
- A `<val>` element or a `<alt>` element can only contain text.
- A `<node>` element can have the following attributes: `name`, `id`, `size` and `type`.

todo: a more formal description of the XML file structure should be inserted in this document.

Chapter 5

Future developments

The short term development 'road map' is:

GSM: Add a SIM card explorer script.

Contactless: Explore the contactless world (e.g. Mifare).

Translation: Translate this document in French.

Bugs: Fix some issues in the code and add some needed error checks.

Chapter 6

Licence

CARDPEEK is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

CARDPEEK is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.