CARDPEEK 0.8 Reference Manual

Alain Pannetrat <L1L1@gmx.com>
February 2013
revision 1

Table of Contents

Chapter 1

Presentation

CARDPEEK is a program that reads the contents of smart cards. This open-source 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, either in contact or contactless mode.

Smart cards 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.

As of version 0.8.2, this tool is capable of reading the contents of the following types of cards:

- EMV "chip and PIN" bank cards, including:
  - VISA, MasterCard, CB and UK Post Office Account contact cards;
  - PayWave (VISA) and PayPass (MasterCard) contactless cards;
- Electronic/Biometric passports, which have an embedded contactless chip;
- The Belgian eID card;
- *Calypso* transport cards including:
  - *Navigo* transport cards used in Paris;
  - MOBIB cards used in Brussels;
  - RavKav cards used in Israel;
  - VIVA cards used in Lisbon;
- GSM SIM cards (without USIM data used in recent cards);

- Vitale 2, the French health card;
- Moneo, the French electronic purse;
- Driver tachograph cards;

It can also read the following cards with limited interpretation of data:

- Some Mifare Classic compatible cards (such as the Thalys card);

Some important card types are missing or need further development, 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™ PC TWIN) as well as contactless or dual-interface PCSC readers (such as the Omnikey™ 5321).

Chapter 2

Installation

CARDPEEK is designed to work under GNU/Linux with GTK+ and has been successfully ported under Windows.

CARDPEEK can be compiled from source using `configure` and `make`. It is being developed under Linux Debian and has been reported to work under Ubuntu, Centos, Fedora, Suse and FreeBSD, as well as Raspbian on the Raspberry Pi.

The Windows version is distributed as a self installing binary package. It can also be compiled under MinGW/MSYS with the custom Makefile provided in the source code (`Makefile.win32`).

Of course, a smart card reader is needed to take full advantage of this software.

2.1 Compiling and installing under Linux

Instructions:

1. Make sure you have the following development packages installed:
   - libgtk+ 3.0, version 3.5.4 or above (http://www.gtk.org)
   - libglib 2.0, version 2.32 or above.
   - liblua 5.2 (http://www.lua.org)
   - libpcsclite (http://pcsclite.alioth.debian.org/)
   - libssl (http://www.openssl.org/)
   - libcurl (http://libcurl.org)
   - libiconv
2. Unpack the source if needed and change directory to the source directory.
3. Type '`./configure`'
4. Type '`make`'
5. Type '`make install`' (usually as root) to install install CARDPEEK in the proper

system directories.

Notes:

1. On a Debian/Ubuntu system, these necessary packages are all available through package management tools such as `apt/aptitude`.
2. The last step (`make install`) is optional, as you can run CARDPEEK directly from the source directory.

2.2 Compiling and installing under Mac OS X

Starting from version 0.8, CARDPEEK should compile on Mac OS X following the same steps outlined for Linux. Note however that this requires setting up the correct development environment and an X11 server (Xquartz).

Cardpeek has notably been successfully compiled under Mac OS X Lion (10.7.5) with the following tools/environment:

1. Xcode 4.6.2 Command Line Tools.
2. Homebrew (http://mxcl.github.io/homebrew/), used to install libgtk+, liblua and libssl.
3. XQuartz (http://xquartz.macosforge.org/).

2.3 Installing under Windows

CARDPEEK works both on windows 32bits and 64bits platforms, in 32 bit compatibility mode.

Instructions:

1. Download the self installing binary setup program (`cardpeek-x.xx-win32-setup.exe` where `x.xx` is the version number of CARDPEEK).
2. Follow the instructions.

CARDPEEK can also be compiled from source with MinGW/MSYS, but this is a more complicated approach due to some current shortcomings of the windows port. You should use the dedicated Makefile (`make -f Makefile.win32`) and manually copy the contents of the directory `dot_cardpeek_dir/` from the source into the directory `%USERPROFILE%/.cardpeek`).

2.4 Related files and initial setup

2.4.1 Default installation

In the following discussion, the terms “home directory” will refer to the traditional home directory on a Unix system (as indicated by the `$HOME` environment variable) or the `%USERPROFILE%` directory on a MS-Windows systems. The first time CARDPEEK is run it will attempt to create the `.cardpeek/` directory in your home directory. This is normal. For Windows users this directory will be created during installation.

The `./cardpeek` directory will contain 5 elements: `cardpeekrc.lua`, `config.lua`, the `scripts/` and `replay/` directories and a `version` file. The `cardpeekrc.lua` allows you to run commands automatically when the program starts and the `config.lua` file is used to store configuration information. The `scripts/` directory contains all the scripts needed to explore smart cards. These scripts are LUA files (such as "`emv.lua`" or "`calypso.lua`") and all show up in the 'analyzer' menu of CARDPEEK (without their extension '`.lua`'). If you add any LUA file to this directory, it will therefore also appear in the menu. The `scripts/` directory contains three subdirectories: `lib/`, `etc/` and `calypso/`. `lib/` and `etc/` hold a few LUA files containing frequently used commands or data items that are shared among the card processing scripts. The directory `calypso/` holds country and region specific scripts for calypso cards. The `replay/` directory is used to save data for card emulation purposes.

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).

2.4.2 Changing the default installation directory

Under Unix-like systems, if you want to change the directory used by CARDPEEK to store and read scripts from `$HOME/.cardpeek` to something else, you may do so by setting the environment variable `CARDPEEK_DIR` to point to an existing alternative directory of your choice.

2.5 Choosing a smart card reader

There are many smart card readers available on the market, and their compatibility with different contact or contactless cards will depend on many parameters, such as:

- The OS you are using (Linux, Windows, 32bit or 64bit, etc.)
- The smart card driver on the OS.
- The firmware of the smart card reader.
- The smart card itself.

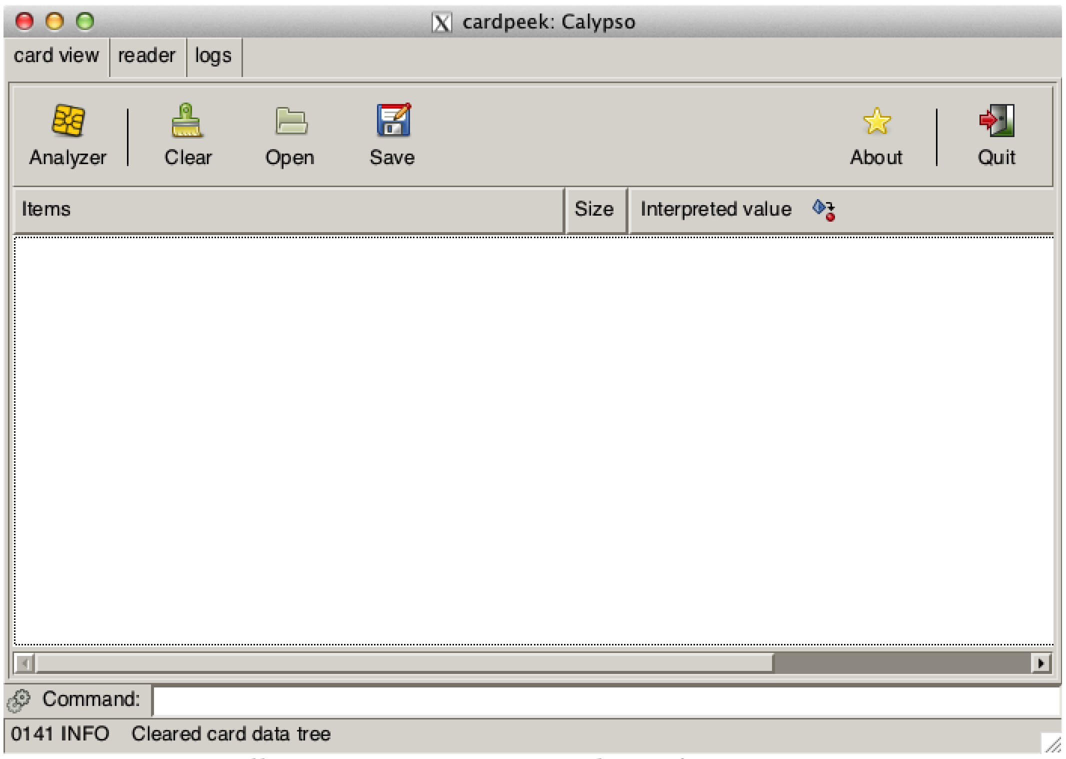
As an example, the OMNIKEY 5321 USB dual-interface reader comes into at least 2 firmware versions. Under MS Windows, a reader with firmware 5.10 fails to connect with some Calypso e-ticketing cards through the contact interface, but works under Linux with default PCSC drivers. However, on Linux, the contactless interface only seems to work with the OMNIKEY drivers and fail to operate with the PCSC standard drivers. Because of all these reasons, to our best knowledge there is no perfect smart card reader.

The following smart card readers have been reported to work with CARDPEEK and are provided here as an indication (without any guaranty):

Reader	**Type**	**Notes**
BROADCOM BCM5880	Contact	*Only on Windows.*
GEMALTO PC Twin USB	Contact	-
OMNIKEY 5321 USB	Contact & contactless	*For Navigo cards, the Omnikey proprietary driver doesn't work. However the default PCSC driver seems to work on Linux.*
SPRINGCARD Prox'N'Roll PC/SC	Contactless	-
WATCHDATA W1981	Contact	*Offered for 7 euros in subway stations in Paris.*
TEO XIRING	Contact	*Reported to work on Linux*

Chapter 3

Using CARDPEEK



```
0141 INFO    Cleared card data tree
```

Illustration 1: Main Window of CARDPEEK

3.1 Quick start

To experiment with CARDPEEK, you 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 (scroll down to the “log data” in the card view).

3.2 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.
- **Logs:** displays a journal of application events, mainly useful for debugging purposes.

3.3 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 Chapter 4*). This tree can be saved and loaded in XML format (*see Chapter 7*) using the buttons in the toolbar.

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

Analyze	Clicking on this button spawns a menu from which a card analysis script can be chosen (*see next chapter).*
Clear	This button clears the card view.
Open	This button allows to load a previously saved card view from an XML file.
Save As	This button allows to save the current card view 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, where each node has a name, composed of a label and an ID. 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 interpreted “alternative” form, such as a text, or a date for example. By default, the card view will display node data in an interpreted “alternative” format if it exists. By clicking on the third column title, it is possible to switch between both “raw” and interpreted “alternative” data representations.

The **card view** tab has a right-click activated context menu featuring two commands:

expand all	This expands the contents of the tree structure starting from the currently highlighted node.
show raw value *or* show interpreted value	This is equivalent to clicking on on the third column title to switch between both “raw” and interpreted” data representations.

3.4 The reader tab

The reader tab displays the raw binary data exchanges between the card reader and the card itself. This data is composed of card command APDUs[1], card response APDUs and card reset indicators. 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 exchanges between the card reader and the smart card in a file that can later be used to emulate the card. Once this data is saved in a file (with the `.clf` extension) and placed in the `.cardpeek/replay/` folder, it will appear as a choice in the smart card reader selection window that appears when CARDPEEK is launched. The name of the file will be prefixed by “`replay://`” 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 smart card inserted in the reader. This is very useful for testing and debugging card scripts without relying on a real smart card inserted in the 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.

1 APDU: Application Protocol Data Unit, a sequence of bytes describing a message exchanged between the smart card and the reader.

Save as	This button allows to save the displayed APDU/response data, either Save as 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 useful to manually force the execution of these commands.

3.5 The logs tab

The **logs** 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.

3.6 The one-line command input field

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

3.7 Card-reader selection upon start-up

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 smart card APDU/response data:* This allows to replay a smart card transaction that was previously recorded by CARDPEEK, and is quite convenient for script debugging purposes. Each time an APDU is sent to the 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 `replay://`.
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 4

Card analysis tools

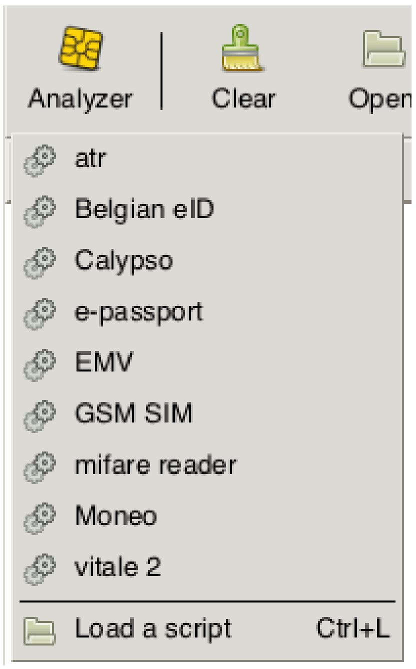


Illustration 2: The Analyzer menu

As shown on Illustration 2, 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. Though you are unlikely to damage a smart card with these tools, these scripts are provided WITHOUT ANY WARRANTY.

4.1 Atr

4.1.1 Overview

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

4.1.2 General notes

This is a very basic script that should always work. It attempts to identify the card with an internal ATR database (smartcard_list.txt), which is automatically updated online on a regular basis.

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

4.2 Belgian eID

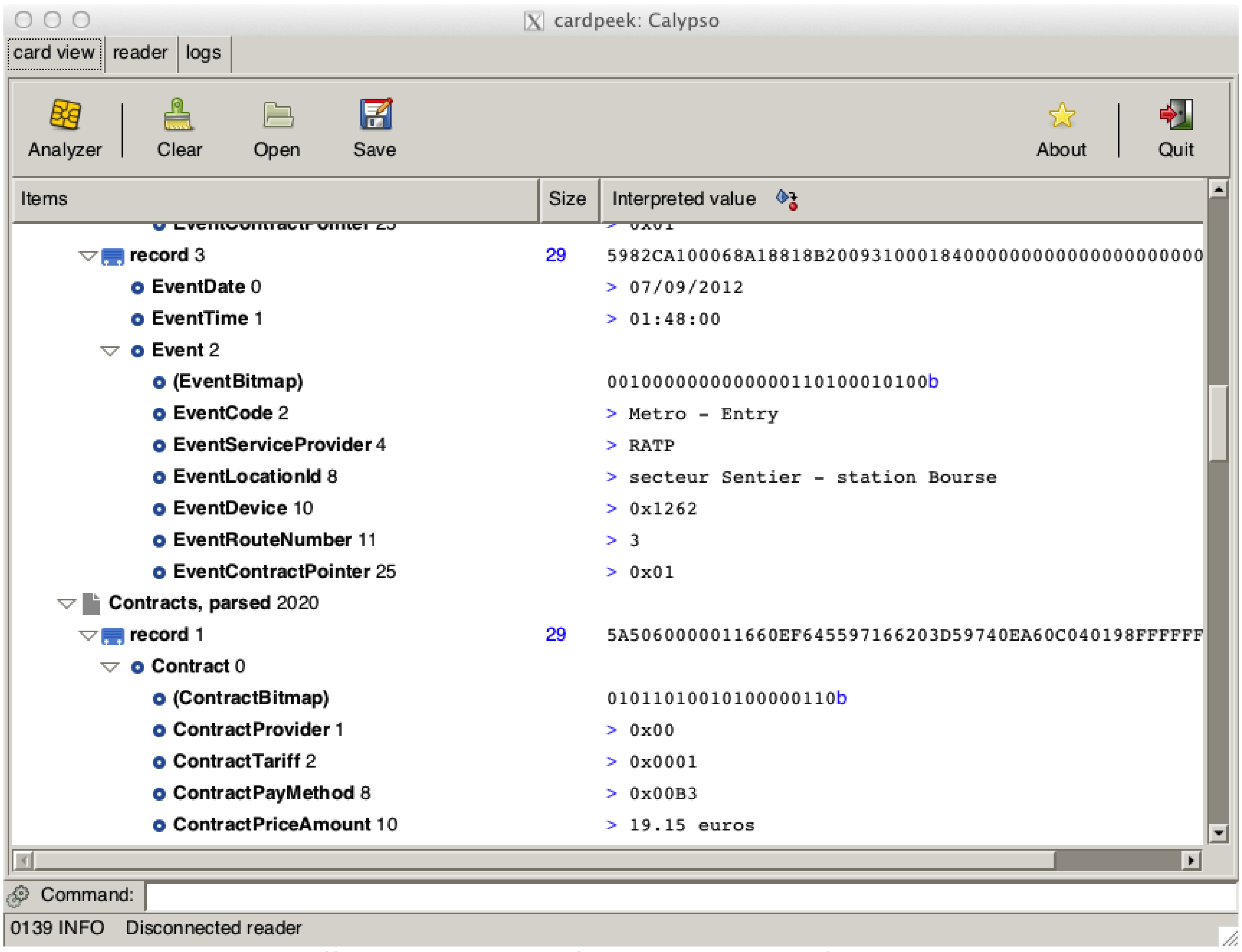
4.2.1 Overview

This script analyses the contents of the Belgian eID card.

4.2.2 General notes

The cardholder's picture is shown if available.

4.3 Calypso



```
Items                              Size  Interpreted value
    record 3                       29    5982CA100068A18818B200931000184000000000000000000000
      EventDate 0                        > 07/09/2012
      EventTime 1                        > 01:48:00
      Event 2
        (EventBitmap)                    00100000000000000110100010100b
        EventCode 2                      > Metro - Entry
        EventServiceProvider 4           > RATP
        EventLocationId 8                > secteur Sentier - station Bourse
        EventDevice 10                   > 0x1262
        EventRouteNumber 11              > 3
        EventContractPointer 25          > 0x01
  Contracts, parsed 2020
    record 1                       29    5A5060000011660EF645597166203D59740EA60C040198FFFFFF
      Contract 0
        (ContractBitmap)                 010110100101000001110b
        ContractProvider 1               > 0x00
        ContractTariff 2                 > 0x0001
        ContractPayMethod 8              > 0x00B3
        ContractPriceAmount 10           > 19.15 euros

0139 INFO    Disconnected reader
```

Illustration 3: Reading a Navigo card (Paris)

4.3.1 Overview

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

4.3.2 Implementation notes

The following calypso cards have been reported to work with this script: Navigo/Paris, MOBIB/Brussels, RavKav/Israel and Korrigo/Rennes (partial support).

You will notice that these transport cards keep an *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.

For Navigo cards, this script provides enhanced “event log” analysis notably with subway/train station names, as illustrated in Figure 4. It has been successfully tested on *Navigo Découverte, Navigo* and *Navigo Intégrale* cards.

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).

The script also reads MOBIB cards used in Brussels, with enhanced “event log” analysis. One unusual feature of the MOBIB card is the possibility to access the name and date of birth of the card holder. MOBIB cards are fully compatible with ISO 14443 card readers.

The calypso script reads all the files it can find on the card and extracts the raw binary data it finds. The interpretation of that binary data varies from country to country, and even from region to region.

Once the data is loaded, the script attempts to automatically detect the country and region the card comes from. The country is identified by a number following ISO 3166-1, but without leading zeros. The region code is also a numerical value. The script will then look into the `calypso` directory for a script called “`cXXX.lua`” where `XXX` represents the country code. If found, this extra script will be executed. Next the main script will look again in the `calypso` directory for a script called “`cXXXnYYY.lua`” where `XXX` represents the country code and `YYY` the region code. If found, this script will also be executed.

Currently country/region detection is based on some simple heuristics and **does not work for all calypso cards.**

Programmers wishing to tailor the behavior of the calypso script to their own country or region can thus add their own file in the `calypso` directory.

4.4 emv

4.4.1 Overview

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

4.4.2 Implementation 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 full access to all freely readable information in the card. As a side effect, issuing this command will increase an internal counter inside the card called ATC (Application Transaction Counter).

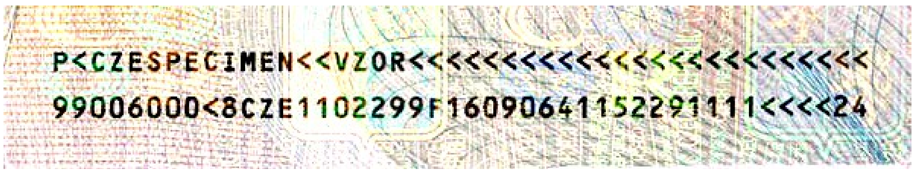
You will notice that many of these bank cards keep a “transaction log” of the last transactions you have made with your card. Some banks cards keep way over a hundred transactions that are freely readable, which brings up some privacy issues.

4.5 e-passport

4.5.1 Overview

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

You will need to enter the second lower line of the MRZ (Machine Readable Zone) data on the passport.



```
P<CZESPECIMEN<<VZOR<<<<<<<<<<<<<<<<<<<<<<<<<
99006000<8CZE1102299F16090641152291111<<<<24
```

4.5.2 Implementation 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 a minimum of 28 characters from the beginning of the second lower line 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. Normally however, fingerprint data is not accessible through BAC and requires EAC.

4.6 GSM/SIM

4.6.1 Overview

This script analyses the contents of GSM SIM cards, including SMS and phonebook contacts.

4.6.2 General notes

This script is limited to classic GSM/SIM card content. Most additional data contained on newer USIM/3G cards is not analyzed.

4.7 moneo

4.7.1 Overview

This scripts provide an analysis of MONEO electronic purse cards used in France.

4.7.2 Implementation notes

The provided output is only partially interpreted.

4.8 Tachograph driver cards

4.8.1 Overview

This script provides an analysis of tachograph smart cards used in Europe. The script offers the possibility to export data in a DDD file (also known as ESM or C1B files).

4.8.2 Implementation notes

The script is currently limited to driver cards. The generated DDD file does not contained signed data blocks.

4.9 vitale 2

4.9.1 Overview

This script provides an analysis of the second generation French health card called “Vitale 2”.

4.9.2 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.

4.10 Adding your own scripts

Adding or modifying a script in CARDPEEK is easy: simply add or modify a script in:

- the `$HOME/.cardpeek/scripts/` directory for Linux/Mac OS X, or
- the `%USERPROFILE%/.cardpeek/scripts/` for Windows users.

On Linux systems and Mac OS X, if you want to go further and make a script permanently part of the source code of CARDPEEK for further distribution, 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_dot_cardpeek.sh update`")
3. Run `make` to rebuild CARDPEEK.
4. The new created binary "CARDPEEK" will now contain your new scripts and can be distributed.

On MS Windows systems, you will need to manually copy your scripts form `%USERPROFILE%/.cardpeek/scripts/` to the `dot_cardpeek_dir` directory in the source code, before recompiling CARDPEEK.

You may provide some additional information in the LUA source code of your script by adding some specific keywords in comments that appear in the first 30 lines of your script. These keywords are identified by being prefixed with the `@` character and must be followed by a space. The following keywords are recognized:

- `@name` → This changes the default name of the script displayed in the menu.
- `@description` → This provides a human readable description of the script
- `@targets` → This provide an indication of the version of CARDPEEK this script was developed for.

Example:

```
-- in LUA comments are marked with a double dash '--'
--
-- @name EMV
-- @description Bank cards 'PIN and chip'
-- @targets 0.8
--
```

Chapter 5

Programming CARDPEEK scripts

The individual scripts that allow to process different types of smart cards are located in your `$HOME/.cardpeek/scripts/` directory or `%USERPROFILE%/.cardpeek/scripts/` for MS Windows. These scripts are written in LUA, a programming language that 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 provides an introduction to the CARDPEEK scripting facilities.

5.1 5.1 Hello world

The simplest CARDPEEK script is probably this one:

```
nodes.append(nodes.root(), {label="Hello world"})
```

You can directly type it in the "Command:" input zone at the bottom of the CARDPEEK GUI. Alternatively you can create a file called `hello_world.lua` in the script directory (indicated above), and copy that one line of script in that file. Once the file is saved, if you start CARDPEEK, "hello world" should appear in the "Analyzer" menu. One click on "hello world" in the menu will execute the script, providing the result shown in Illustration 5.

The above script is very simple and does not interact with any kind of smart card: it simply creates a node in the tree view area of CARDPEEK and assigns the label "Hello world" to it. The first parameter of the `append` command describes the parent node to which we add a new node. Since we are actually creating the first node in the tree, we use the special function `nodes.root()` which returns a reference to the default root node of the tree. The second value `{label="Hello world"}` that is passed to the `nodes.append()` function describes the attributes of the node. In this example, the node has only one attribute specified: a label, which is set to "Hello world".

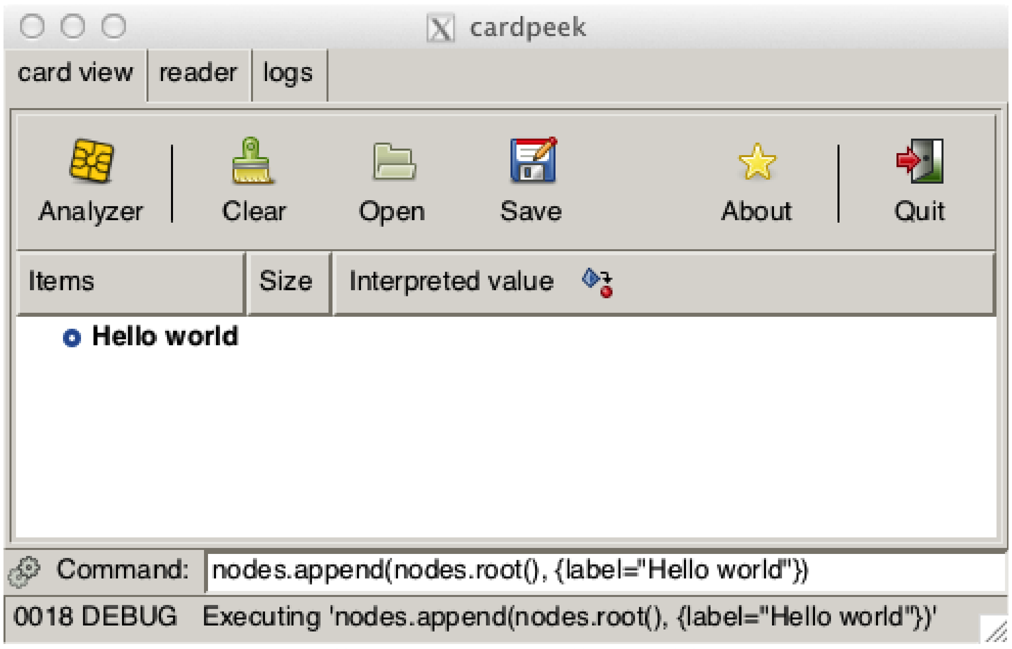
Node attributes can be accessed and modified at any time, with the `nodes.set_attribute()` and `nodes.get_attribute()` functions respectively. The above script could have been written in two lines with the same result as follows:

```
new_node = nodes.append(nodes.root())
nodes.set_attribute(new_node, "label", "Hello world")
```

Finally, it is worth noting that the function in the nodes library can use the Lua Object Oriented (OO) notation, providing a third and more concise way to write the "Hello world" script as follows:

```
nodes.root():append({label="Hello world"})
```

The choice of using the Lua OO notation instead of the traditional one is purely a matter of taste.



```
nodes.append(nodes.root(), {label="Hello world"})
0018 DEBUG   Executing 'nodes.append(nodes.root(), {label="Hello world"})'
```

Illustration 5: Hello world

5.2 Basic communication with a smart card

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

```
card.connect()

atr = card.last_atr()

if atr~=nil then

     root = nodes.root()
```

```
        mycard = root:append({ classname="card",
                               label="My card" })

        mycard:append({ classname="block",
                        label="Cold ATR",
                        size=#atr,
                        val=atr })
    end

    card.disconnect()
```

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 “ATR”, with a call to `nodes.append()`. This node will appear at the root of the card data tree-view and is constructed with two attributes:

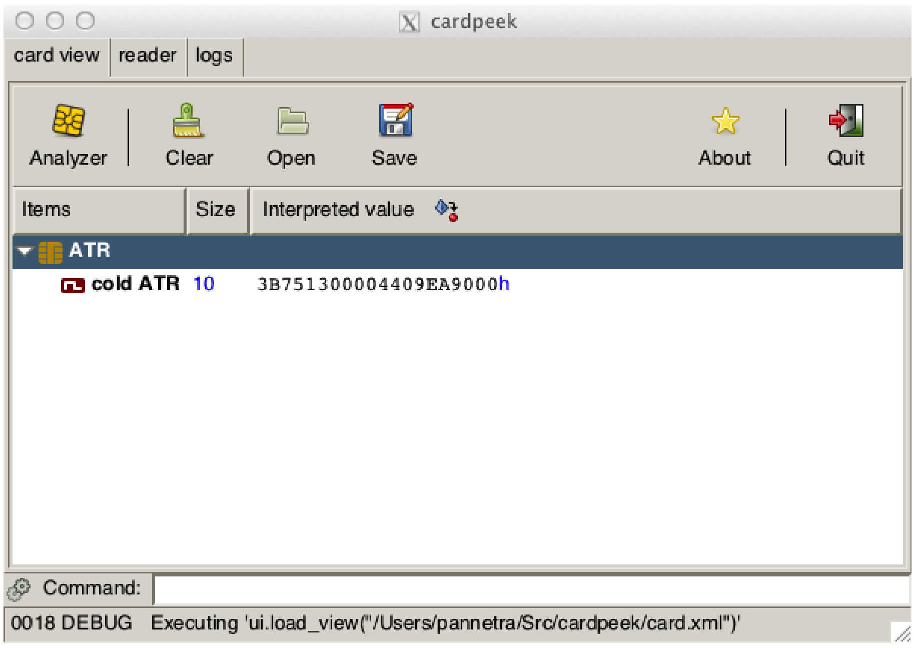
- A *classname*: this describes the icon used to display the node (here a “card” icon).
- A *label*: the text displayed with the node.

A child node called “cold ATR” is then added to the parent “ATR” node, with 4 attributes:

- A *classname*: the icon used to display the node (here a “block” icon).
- A *label*: the text displayed with the node (i.e. “Cold ATR”).
- A *size*: the text displayed in the second column of the tree view.
- A *val*: the bytestring value displayed in the third column of the tree view (i.e. the hexadecimal value of the ATR).

Finally, the card is powered down with the `card.disconnect()` function.

The final output of the script should have roughly the following structure (though the value of the ATR will likely be different):



```
ATR
  cold ATR  10   3B751300004409EA9000h
```

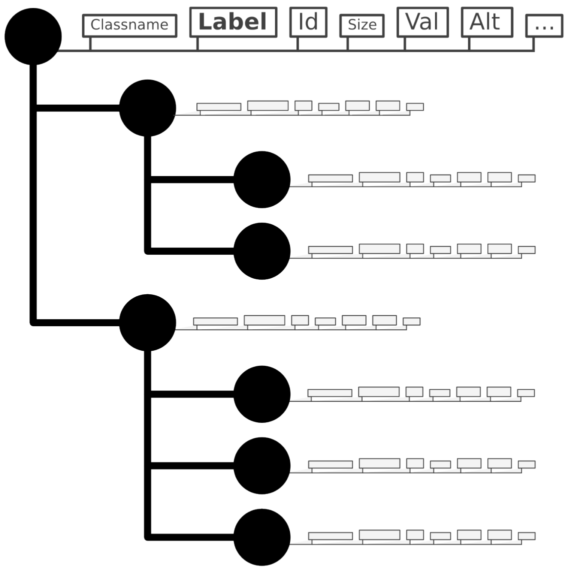
```
0018 DEBUG   Executing 'ui.load_view("/Users/pannetra/Src/cardpeek/card.xml")'
```

Illustration 6: Displaying the ATR

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.

5.3 Representing card data in a tree structure

The data displayed in the card view of CARDPEEK follows a tree structure, as sketched in Illustration 7.



Each node of the tree has the following attributes that influence the display of data in the card view:

- **A classname:** describes the icon that will be associated with the node in the first column of the card view.
- **A label:** describes the name of the node, shown in bold in the first column of the card
  view.
- **An id:** describes the Id associated with the node, following the label, in the first column of the card view.
- **A size:** the size of the data associated to the node, displayed in the second column of the card view.
- **A value (val):** the data associated to the node in raw binary format, displayed in the third column of the card view.
- **An alternative value (alt):** the data associated to the node in an interpreted format, displayed in the third column of the card view.
- **A mime-type**: this gives CARDPEEK a hint on how to render the value (val) on the screen, when an alternative value (alt) is not specified. Currently only images are supported.

All these attributes are optional. Moreover, script programmers can create new attributes as they wish for their own use, though only the ones above influence the display of CARDPEEK. Attributes are set through functions such as `nodes.set_attribute()` and `nodes.set_value()` as described in section 6.6.

The tree itself is built with functions such as `nodes.append()` already described in the previous examples in this chapter.

CARDPEEK provides many other functions to create, remove, alter and find nodes in a tree, all described in section 6.6.

Chapter 6

Development library

This chapter describes the LUA libraries of functions that are used in CARDPEEK scripts.

6.1 The bit library

Since LUA 5.1 does not have native bit manipulation functions, the following functions have been added. They all operate on integer numbers.

6.1.1 bit.AND

SYNOPSIS

```
bit.AND(A,B)
```

DESCRIPTION

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

6.1.2 bit.OR

SYNOPSIS

```
bit.OR(A,B)
```

DESCRIPTION

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

6.1.3 bit.XOR

SYNOPSIS

```
bit.XOR(A,B)
```

DESCRIPTION

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

6.1.4 bit.SHL

SYNOPSIS

```
bit.SHL(A,B)
```

DESCRIPTION

Shift the bits of A by B positions to the left. This is equivalent to computing $A*2^B$.

6.1.5 bit.SHR

SYNOPSIS

```
bit.SHR(A,B)
```

DESCRIPTION

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

6.2 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.

A bytestring is mainly used to represent binary data exchanged with the card reader in the application.

The elements in a *bytestring* array are most commonly bytes (8 bits), but it is also possible to construct arrays of nibbles (4 bit) 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 or modified with the `bytes.get` and `bytes.set` functions respectively. 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`). It is also possible to access elements in bytestring using array notation (e.g. `BS[1]=65`).

Like LUA arrays, if you create a bytestring `A` then the assignment `B=A` does not make `B` a full copy of `A` but only an alias of `A`: any modification of `A` applies to `B` and vice-versa. If you want to create a real copy of a bytestring, you should use the `bytes.clone()` function or the `bytes.new()` function instead.

The functions of the `bytes` library are described hereafter.

6.2.1 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. Writing `A..B` is equivalent to calling the function `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.

6.2.2 bytes.clone

SYNOPSIS

```
bytes.clone(BS)
```

```
BS:clone()
```

DESCRIPTION

Creates and returns a copy of `BS`.

RETURN VALUE

This function returns a bytestring upon success or `nil` if it fails.

6.2.3 bytes.concat

SYNOPSIS

`bytes.concat(`*$value_0$, $value_1$, ..., $value_N$*`)`

DESCRIPTION

Returns the concatenation of *$value_0$, $value_1$, ..., $value_N$* (from left to right). For the rules governing the processing of *$value_0$, $value_1$, ..., $value_N$,* see the `bytes.new()` function.

RETURN VALUE

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

6.2.4 bytes.convert

SYNOPSIS

```
bytes.convert(BS, w)
```

```
BS:convert(w)
```

DESCRIPTION

Converts `BS` to a new bytestring where each element has a width `w`. Depending on the value of `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.

6.2.5 bytes.format

SYNOPSIS

```
bytes.format(BS, format_string)

BS:format(format_string)
```

DESCRIPTION

Converts the bytestring `BS` to various printable formats according to the `format_string` character string. This `format_string` can be composed of plain characters, which are simply copied to the resulting string, and format specifications which are replaced by the designated representation of `BS`.

As in `printf()` functions found in many programming languages, each format specification starts with the character "`%`" and has the following meaning:

- `%I` represent `BS` as an unsigned decimal integer.
- `%D` represent `BS` as the concatenation of each of its elements represented in hexadecimal or binary, starting from `BS[0]` to `BS[N-1]`.
- `%S` is equivalent to "`%w:%D`".
- `%P` represent `BS` where each element is converted to a printable character (*in 7 bit ascii format)*. Unprintable characters are escaped to octal notation (e.g. ascii 13 or carriage return becomes \015).
- `%C` represents `BS` as a LUA string, converting each element into a character of a string. This can notably be used to print a bytestring representing UTF8 data.
- `%w` represents the width of `BS`, that is 8, 4 or 1.
- `%l` represents the number of elements in `BS`, in decimal form (the length of `BS`).
- `%%` represent the "`%`" character itself.

RETURN VALUE

This function returns the resulting character string.

6.2.6 bytes.get

SYNOPSIS

```
bytes.get(BS,i [,j])

BS:get(i [,j])
```

DESCRIPTION

Returns the list of values of elements of BS with index `i` to `j` inclusive. When `j` is not specified, this function simply returns the `i`th element of BS (i.e it is equivalent to `bytes.get(BS,i,i)` or more simply `BS[i]`).

RETURN VALUE

This function returns a list of integer values upon success or nil otherwise.

6.2.7 bytes.ipairs

SYNOPSIS

```
for i,v in bytes.ipairs(BS) do
      ...
end

for i,v in BS:ipairs() do
      ...
end
```

DESCRIPTION

The `bytes.ipairs(BS)` function returns an iterator on the elements of a bytestring `BS`. This iterator returns both the index and the value of each element, much like the `ipairs()` iterator used for tables in LUA.

RETURN VALUE

This function returns an iterator (a LUA function) upon success or `nil` otherwise.

6.2.8 bytes.is_printable

SYNOPSIS

```
bytes.is_printable(BS)

BS:is_printable()
```

DESCRIPTION

Returns `true` if all elements in `BS` can be converted to printable 7 bit 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).

6.2.9 bytes.new

SYNOPSIS

bytes.new(width [, *value*$_1$*, ..., value*$_N$])

DESCRIPTION

Creates a new *bytestring,* where each element is `width` bits. `width` can be either 8, 4 or 1. A value can optionally be assigned to the *bytestring* by specifying one or several values *value*$_1$*, ..., value*$_N$ that will be concatenated together to form the content of the *bytestring.*

The optional assigned value is built as the concatenation of *value*$_1$*, ..., value*$_N$ (from left to right). Each *value*$_i$ can be either a bytestring, a string or a number. If *value*$_i$ is a bytestring, each element of *value*$_i$ is appended to `BS,` without any conversion. If *value*$_i$ is a string, it is interpreted as a text representation of the digits 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 *value*$_i$ is a number, it is converted into a single bytestring element and appended to `BS.`

RETURN VALUE

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

6.2.10 bytes.new_from_chars

SYNOPSIS

```
bytes.new_from_chars(string)
```

DESCRIPTION

Creates a new 8 bit width bytestring from `string.` Each ascii character in `string` is converted directly to an element of the resulting bytestring (e.g. “A” is converted to 65).

RETURN VALUE

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

6.2.11 bytes.pad_left

SYNOPSIS

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

```
BS:pad_left(block_length, value)
```

DESCRIPTION

This function returns a copy of `BS` padded on the left with the element `value` until the length of the resulting bytestring reaches a multiple of `block_length`.

If the size of `BS` is already a multiple of `block_length`, no additional padding is applied.

RETURN VALUE

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

6.2.12 bytes.pad_right

SYNOPSIS

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

```
BS:pad_right(BS, length, value)
```

DESCRIPTION

This function returns a copy of `BS` padded on the right with the element `value` until the length of the resulting bytestring reaches a multiple of `block_length`.

If the size of `BS` is already a multiple of `block_length`, no additional padding is applied.

RETURN VALUE

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

6.2.13 bytes.reverse

SYNOPSIS

```
bytes.reverse(BS)
```

```
BS:reverse()
```

DESCRIPTION

Returns a bytestring containing the elements of `BS` in reverse order.

RETURN VALUE

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

6.2.14 bytes.set

SYNOPSIS

```
bytes.set(BS, i, e0 [, e1, e2, ..., eN])

BS:set(i, e0 [, e1, e2, ..., eN])
```

DESCRIPTION

This function replaces the `i`th element of BS with `e0`. If additional elements `e1,e2,...,eN` are specified in the call then the `(i+1)`th element of `BS` is replaced by `e1`, the `(i+2)`th element of `BS` is replaced by `e2`, etc.

RETURN VALUE

This function modifies its main argument `BS` and returns it.

6.2.15 bytes.sub

SYNOPSIS

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

DESCRIPTION

Returns a copy of a substring from `BS` containing all elements between `start` and `end`. The returned value represents a bytestring containing a copy of all the elements of `BS` that have an index that verifies *index* ≥ `start` and *index* ≤ `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` respectively).

RETURN VALUE

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

6.2.16 bytes.tonumber

SYNOPSIS

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

```
BS:tonumber()
```

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.

6.2.17 bytes.width

SYNOPSIS

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

```
BS:width()
```

DESCRIPTION

Return the width of the elements in `BS.`

RETURN VALUE

This function may return the number 1, 4 or 8.

6.3 The `asn1` library

The ASN1 library[2] allows to manipulate bytestrings containing ASN1 TLV[3] data encoded in DER/BER[4] format. These bytestrings must be 8 bit wide.

When CARDPEEK reads TLV data from a card, it comes as a bytestring where the tag is encoded, followed by the length, and finally the value itself. For example CARDPEEK may receive the following string: `4F07A0000000031010,` where `4F` is actually the tag, `07` the length, and `A0000000031010` is the value. In some cases, the tag or the length follow more complex encodings, and some TLVs may be contained within other TLVs. The `asn1` library provides facilities to decode and encode TLV data.

2 For a quick tutorial on ASN1 see “A Layman's Guide to a Subset of ASN.1, BER, and DER” by B. S. Kaliski Jr.
3 TLV = Tag,Length,Value
4 DER/BER = Distinguished/Basic Encoding Rules

Normally, TLV values are composed of 3 elements: a tag number, a length, and the value itself. In LUA, we only need 2 items to represent a TLV: a tag number and a value represented as a bytestring. The length of the value is implicit and can computed by applying the # operator on the value.

The library provides the following functions.

6.3.1 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.

6.3.2 asn1.join

SYNOPSIS

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

DESCRIPTION

This function performs the opposite of `asn1.split()` (described in 6.3.3): it 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.

6.3.3 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 number `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.

6.3.4 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.

6.3.5 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 {`T`, *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.

6.4 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.

According to ISO 7816-4, smart card command APDUs are composed as a series of bytes generally organized as follows:

Code	Length	Name
CLA	1	Class
INS	1	Instruction
P1	1	Parameter 1
P2	1	Parameter 2
Lc	1 (or 3)	Length of following data
Data	Variable	Data
Le	1 (or 3)	Maximum expected length of response

The card library defines a global value `card.CLA`, which is the value that most card commands will use as CLA when they exchange data with the card-reader (unless you use `card.send()` directly).

This library contains the following functions.

6.4.1 card.connect

SYNOPSIS

```
card.connect()
```

DESCRIPTION

Connect to the card currently inserted in the selected smart card reader or in proximity of a contactless smart card reader. This function will block until card is connected.

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

RETURN VALUE

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

6.4.2 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.

6.4.3 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 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 6.4.10).

6.4.4 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 ⇒ value)* pairs.

6.4.5 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.

6.4.6 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 in other words 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.

6.4.7 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 ≤ `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 6.4.10).

6.4.8 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 ≤ `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 6.4.10).

6.4.9 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 6.4.10).

6.4.10 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).

6.4.11 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

6.5 The `crypto` library

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

6.5.1 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 (so 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, which 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 (but is ignored for hash 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.

6.5.2 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 an 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.

6.5.3 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.

6.5.4 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 an 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.

6.5.5 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 MAC as a bytestring. The resulting MAC is not truncated.

6.6 The iconv library

The nodes library allows to convert strings from one encoding to another. This is useful for example to convert ISO-8859-1 character strings to UTF-8 for display in Cardpeek, since Cardpeek expects UTF-8 strings in the *treeview*.

This library is a wrapper around the `iconv` standard library provided on most unix platforms. See the `iconv_open()` and `iconv()` man pages on Linux/Unix for more details.

6.6.1 iconv.open

SYNOPSIS

```
converter = iconv.open(formcode, tocode)
```

DESCRIPTION

This function creates a converter object that can be used to convert text encoded in `fromcode` to text encoded in `tocode`.

Both `fromcode` and `tocode` are strings describing an encoding such as "ISO-8859-1", "UTF-8", "ASCII", "CP437", etc.

RETURN VALUE

Returns a "converter" object that can be used in subsequent calls to `iconv.iconv()` or `nil` in case of failure.

6.6.2 iconv.iconv

SYNOPSIS

```
converted_string = converter:iconv(orginal_string)
```

```
converted_string = original_string:iconv()
```

DESCRIPTION

This function converts a string from one format to another, using a converter object created previously with `iconv.open()`.

The function takes a string as input and returns the converted string.

RETURN VALUE

Returns the converted string in case or success or `nil` otherwise.

6.7 The nodes library

The nodes library allows to add, delete, modify and find nodes in the card view representation of the card content.

6.7.1 nodes.append

SYNOPSIS

```
nodes.append(parent [, attr_array])
```

```
parent:append([attr_array])
```

DESCRIPTION

This function adds a node in the card tree structure.

The new node will be appended to the children of the node identified by the *node reference* `parent`. If `parent` is `nodes.root()` the new node will be added at the top level (see 6.7.10).

The content of the new node can optionally be specified through `attr_array`, a LUA associative array of (key,value) pairs describing the content of the new node. For example, to create a node with the label “foo” with a “file” icon, you would pass the value `{classname="file", label="foo"}` as `attr_array`. While you may specify any (key,value) pairs, the nodes library gives a specific meaning to the following keys:

- `classname:` a 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 `classname` values are associated with a distinct icon: “application”, “block”, “card”, “file”, “record” and “item”. If `classname` is `nil` or unrecognized, it will be set to the default value “item”.
- `label:` a string that describes the data that is represented by the node in human readable form (such as a “file” or a “date of birth” for example).
- `id:` a string that identifies the node uniquely within a context (such as a

number or a unique name).

- `size:` is a number describing the length of the data element associated to the node. If set, it will be displayed in the second column of the card view in the UI.
- `Val:` a bytestring that describes the raw data presented in the third column of the card view in the UI.
- `alt` is an optional string that describes the interpreted data presented in the third column of the card view in the UI.
- `mime-type` is a string that is used to optionally indicate how to render visually the value specified by the `val` attribute (described above) when the `alt` attribute is absent. Currently this is only used to render images by specifying image data in the val attribute and setting mime-type to "image/jpeg" for example.

These attributes can be also added or modified in a node with the `node.set_attribute()` function (see 6.7.11).

RETURN VALUE

Upon success the function returns a *node reference* to the newly created node. If the function fails, it returns `nil`.

6.7.2 nodes.attributes

SYNOPSIS

```
for k,v in nodes.attributes(node_ref) do
    ...
end

for k,v in node_ref:attributes() do
    ...
end
```

DESCRIPTION

This function provides an iterator for the attributes of `node_ref`, a node reference. The attributes are represented as key/value pairs (in the example here `k,v`).

RETURN VALUE

Returns an iterator for use in a `for` loop.

6.7.3 nodes.children

SYNOPSIS

```
for child in nodes.children(parent) do
      ...
end

for child in parent:children() do
      ...
end
```

DESCRIPTION

This function provides an iterator for the child nodes of `parent`, a node reference. The iterator returns a reference to each child of the parent node (in the example here `child`).

RETURN VALUE

Returns an iterator for use in a `for` loop.

6.7.4 nodes.find

SYNOPSIS

```
for node_ref in nodes.find(root, attr_array) do
      ...
end

for node_ref in root:find(attr_array) do
      ...
end
```

DESCRIPTION

This function provides an iterator that will return all nodes in the card tree view rooted at `root`, which match the attributes defined in `attr_array`. The attribute array `attr_array` is a LUA associative array of key/value pairs as defined for the function `nodes.append()` (see 6.7.1).

A node is considered as matching `attr_array` if all key value/pairs defined in `attr_array` exist in the node and have the same value. For example if we use `{label="file"}` as `attr_array`, the iterator will go through all nodes that have the label "file", regardless of other attributes the nodes may have.

RETURN VALUE

Returns an iterator for use in a `for` loop.

6.7.5 nodes.find_first

SYNOPSIS

```
nodes.find_first(root, attr_array)
```

```
root:find_first(attr_array)
```

DESCRIPTION

This function returns the first node in the card tree view rooted at `root`, which matches the attributes defined in `attr_array`. The attribute array `attr_array` is a LUA associative array of key/value pairs that is used for matching following the same rules used for `node.find()` (see 6.7.4).

RETURN VALUE

A node reference or `nil` if no node is found.

6.7.6 nodes.from_xml

SYNOPSIS

```
nodes.from_xml(parent,xml_string)
```

```
parent:from_xml(xml_string)
```

DESCRIPTION

This function appends nodes described in the XML expression `xml_string` to parent. The expression `xml_string` describes a subtree of nodes, following the syntax defined in Chapter 7 .

RETURN VALUE

This function returns `true` in case of success and `false` otherwise.

6.7.7 nodes.get_attribute

SYNOPSIS

```
nodes.get_attribute(node_ref, attr_name)
```

```
node_ref:get_attribute(attr_name)
```

DESCRIPTION

Gets the value of an attribute in the node identified by `node_ref`. The name of the attribute to retrieve is identified by the string `attr_name`.

The attributes named “`classname`”, “`label`”, “`id`”, “`alt`”, “`val`”, “`mime-type`” and “`size`” refer to the parameters passed to the function `nodes.append()` as described in section 6.7.1.

RETURN VALUE

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

6.7.8 nodes.parent

SYNOPSIS

```
nodes.parent(node_ref)
```

```
node_ref:parent()
```

DESCRIPTION

Returns the parent node of the node referenced by `node_ref`. If the node referenced by `node_ref` has no parent the function return `nil`.

RETURN VALUE

Return a node reference upon success or `nil` otherwise.

6.7.9 nodes.remove

SYNOPSIS

```
nodes.remove(node_ref)
```

```
node_ref:remove()
```

DESCRIPTION

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

RETURN VALUE

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

6.7.10 nodes.root

SYNOPSIS

```
nodes.root()
```

DESCRIPTION

This function simply return the absolute root of the card view tree, which is invisible. This is typically used to create the first node in the tree in combination with `node.append()` (e.g. "`node.root():append(...)`")

RETURN VALUE

A node reference.

6.7.11 nodes.set_attribute

SYNOPSIS

```
nodes.set_attribute(node, attr_name, attr_value)
```

```
node:set_attribute(attr_name, attr_value)
```

DESCRIPTION

Sets an attribute in the node identified by `node_ref`. The attribute to set is identified by the string `attr_name` and takes the value indicated by the string `attr_value`.

The attributes named "`classname`", "`label`", "`id`", "`alt`", "`val`", "`mime-type`" and "`size`" refer to the parameters passed to the function `nodes.append()` as described in section 6.7.1.

The programmer can associate any arbitrary attribute with a node.

RETURN VALUE

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

6.7.12 nodes.to_xml

SYNOPSIS

```
nodes.to_xml(node)
```

```
node:to_xml()
```

DESCRIPTION

Returns an XML representation of the sub-tree that has `node` as a root. If `node` is `nodes.root()` the representation of the whole tree is returned.

RETURN VALUE

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

6.8 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 tree structure representing card data is composed of nodes, each represented on one row in the card tree view. Some function in the `ui` library are used to manipulate these nodes (or rows), allowing to add, remove or edit them. These functions identify each node by a node reference, which is an internal opaque type. The `ui` library functions are described in the following paragraphs.

6.8.1 ui.question

SYNOPSIS

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

DESCRIPTION

Asks the user a question requesting him to answer by selecting a response. The question is described in the string `text`, while the set of possible answers described in the LUA array `buttons`. Each element in the array `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 indices are usually numbers greater or equal to 1).

Upon failure the function returns 0.

EXAMPLE

```
ui.question("Quit the script?", { "yes", "no" } )
```

6.8.2 ui.readline

Synopsis

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

Description

Request the user to enter a text string. The user's input can optionally be limited to `len` characters and can also optionally hold a predefined value `default_value`.

Return Value

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

Example

```
ui.readline("Enter PIN code:", 4, "0000")
```

6.8.3 ui.load_view

Synopsis

```
ui.load_view(file_name)
```

Description

Loads the tree from the XML file named `file_name`. See Chapter 7 for a description of the file format.

Return Value

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

6.8.4 ui.save_view

Synopsis

```
ui.save_view(file_name)
```

Description

Saves the tree in XML format inside the file named `file_name`. See Chapter 7 for a description of the file format.

Return Value

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

6.9 The `log` library

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

6.9.1 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.

6.10 Other libraries

6.10.1 The treeflex library

As of version 0.8, the `treeflex` library is deprecated. All functions previously provided by this library are now available through the `nodes` library.

6.10.2 The country_codes and currency_codes libraries

These library provide convenience functions to translate currency and country codes in human readable names.

6.10.3 The en1545 library

This library provides tools to manipulate data used in Calypso cards that follow CEN/ISO 1545.

6.8.4 The strict library

This library forces LUA variables to be explicitly declared, and thus reduces programming errors.

6.8.5 The tlv library

This library is built upon the `asn1` library and provides automated tools to analyze and display complex ASN1 TLV data objects in CARDPEEK.

Chapter 7

File format

The card view presented in CARDPEEK can be save or imported in XML format. This format is quite straightforward, as shown in the following example, which was created with the `atr` script:

```
<?xml version="1.0"?>
<cardpeek>
  <version>0.8</version>
  <node>
    <attr name="classname">card</attr>
    <attr name="label">ATR</attr>
    <node>
      <attr name="classname">block</attr>
      <attr name="label">cold ATR</attr>
      <attr name="size">10</attr>
      <attr name="val" type="bytes">8:3B751300004409EA9000</attr>
    </node>
  </node>
</cardpeek>
```

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

- The root element of the XML structure is **`<cardpeek>`**, which contains one **`<version>`** element followed by one or more **`<node>`** elements.
- The **`<version>`** element contains the file format version number (currently 0.8).
- A **`<node>`** element may contain both **`<node>`** and **`<attr>`** elements.
- A **`<attr>`** element has one mandatory XML attribute "name" which describes the name of a "node attribute" associated with a node in the tree view, while the text inside the **`<attr>`** element describes the value associated with that "node attribute". The **`<attr>`** element has one optional XML attributes "`type`" that is used to specify the format of the content data, when it is not a standard string. Currently, the

attribute `type` can only take one value "`bytes`" to indicate that the content data is an encoded bytestring.

A node can have any number of "node attributes" defined by an **`<attr>`** element. However, some "node attributes" have a specific meaning for CARDPEEK (see 6.7.1):

- **`<attr`** `name="classname">` describes the type of node (a file, an application, a data block, a data item, etc.) and its value will determine the icon used to represent the node on the screen in the application.
- **`<attr`** `name="label">` describes the label given to the node on the screen (first column).
- **`<attr`** `name="id">` describes the id of the node displayed on the screen (first column).
- **`<attr`** `name="size"> ` describes the size that is displayed on the screen (second column).
- **`<attr`** `name="val" type="bytes">` describes the value of the bytestring associated with a node (and represented in the third column on the screen). This bytestring is represented as a width value followed by ":" and the digits representing the bytestring (this is equivalent to the `%S` output format of the `bytes.format()` function).
- **`<attr`** `name="alt">` describes an alternative representation of the value associated with a node and represented in simple text format.
- **`<attr`** `name="mime-type">` is an indicator of how to render the data described by the `val` attribute in the absence of the `alt` attribute (see 6.7.1).

Node attribute names stating with a minus sign (example: "`-makup-val`") are considered as temporary attributes and are not exported or saved in XML format. They are used internally by CARDPEEK or script programs.

Note: the CARDPEEK XML format has changed in CARDPEEK version 0.8 and is not compatible with previous versions.

Chapter 8

License