FabLabKasse Documentation

FAU FabLab team and others

Aug 31, 2018

Contents

Contents:

CHAPTER 1

A Brief History of Cash-Acceptance

1.1 Prelude

The FAU FabLab is a universitarian grassroot FabLab at the Univeristy of Erlangen-Nuremberg in Germany, run by volunteers but with partial initial funding through the university. Students and any interested party can come by and make use of the FabLab in their free time or for class work and research. Since the upkeep of machines and tools, as well as the needed materials are not financed by the university, users have to pay for usage.

1.2 First Act

With a total of 1291 individual products for sale, you can imagine how messy it gets to handle and supervise finances and operations. We started with hand-labeling all products and trust users to pay the right amount into an opened cash box. But how could we know if somebody would steal money from the lab?

1.3 Second Act

Next, we added a handwritten cash journal and asked users to write down the paid amounts. But with this came extra work, since I had to copy that paper over to an excel sheet and check the sums, roughly twice a week. Obviously people made mistakes and payed a little more or less, but it always evened out over a couple of days and we did not notice any theft. But with 1.000+ products, how do we keep track of what is being sold, you might ask. We did not. You might also ask, did you not get tired of typewriting endless lists of numbers and counting cash? For sure I did!

Therefore, we started implementing a touchscreen-based sales-terminal, which would replace the handwritten lists and know about all the 1.000+ products. It took some time, lots of python code and caffeinated drinks, but finally we had it. No more typing, but still all the counting of coins and bills.

1.4 Third Act

As time progressed, we were quite happy by all the automated tabulation and statistics. Until we noticed that 100 Euros were missing… We never figured it out, so we must assume that it was stolen out of the open cash box. What to do?

1.5 Fourth Act

We already had the basic software at hand, but we needed hardware. Hardware to discourage people from stealing. Since weaponizing our cash box would not have been in line with the safety aspects of the Fab Charter nor the UN Declaration of Human Rights, we needed to take a look at less drastic measures. With a FabLab at hand, we began designing and building a FabATM, or for long-word-loving-germans: ein Besucherabkassiermaschinenautomat. For real: it is called kassenterminal, which means payment terminal.

1.6 Sixth Act

We have built an open-soure point-of-sale terminal, which has been in everyday operation for almost a year. It counts coins, bills, returns change, prints official receipts and is completely capable of self-service. It tracks sales and will soon accept electronic payments and tell us what to restock. No more theft has been detected and endless hours of counting coins and typing numbers have been abolished.

If you are interested in not reenacting our story, have a look at the following GitHub-projects, which contain all the information necessary to build your own:

- Software: https://github.com/fau-fablab/FabLabKasse
- Wooden case: https://github.com/fau-fablab/kassenautomat.CAD
- Interface circuit board: https://github.com/fau-fablab/kassenautomat.mdb-interface

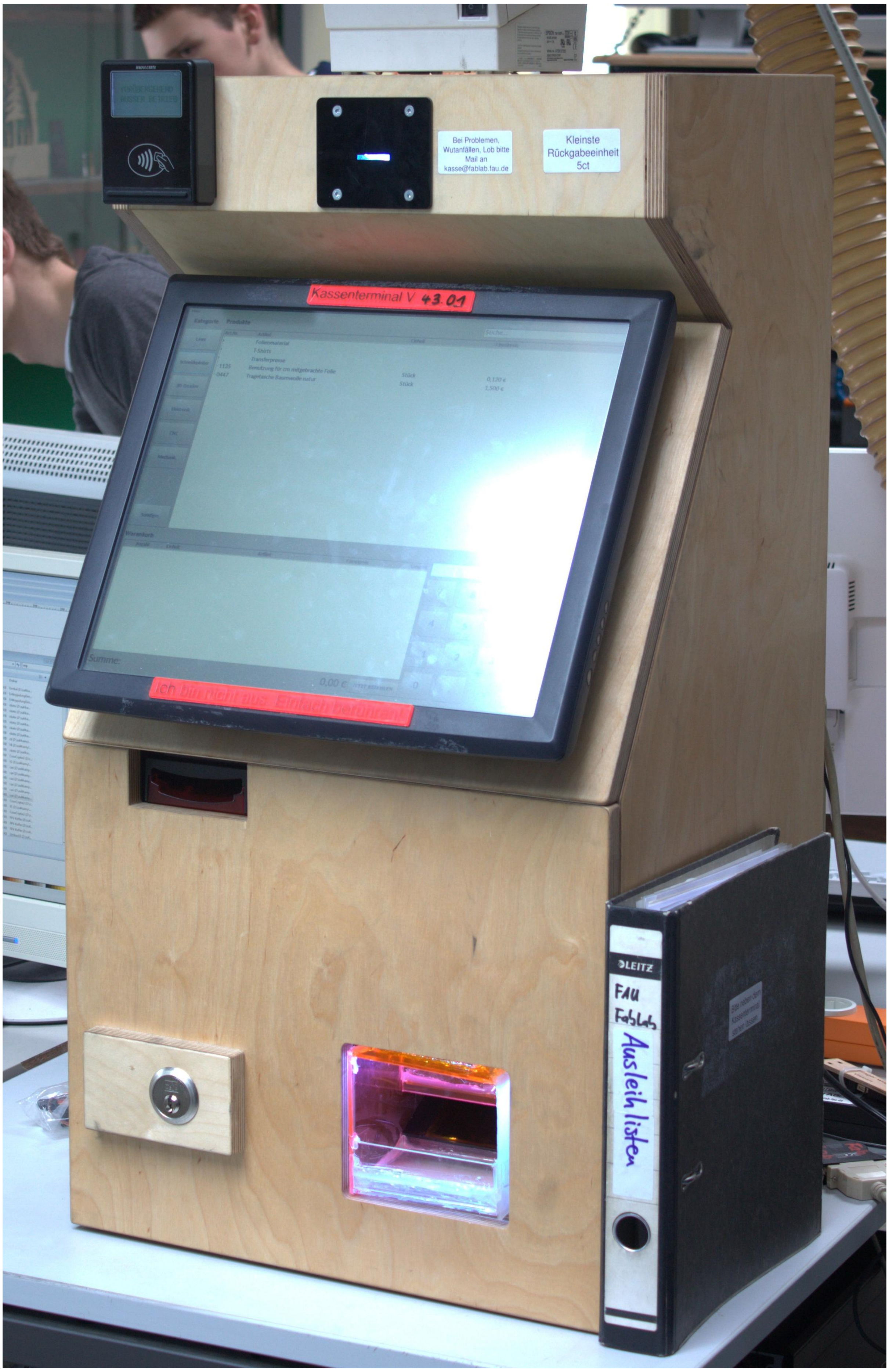
The cash-devices we used (for counting, verifying and returning bills and coins) are connected via an industry-standard interface and can be replaced with other such devices. They can also be ignored, if the use of an open cash box or a drop-in-only cash box (without change) is to be used. The software is already equipped for this. Have fun and keep your bean counters and bank accounts happy by using and helping in developing our automated cash system!

1.7 Sequel

At our university there is a class in which mobile apps are developed, called MAD (mobile application development). MAD developed for us an app that works together with the kassenterminal. You can select products that are stored in our ERP and then send them to the kassenterminal in a checkout process.

The app is released as iOS-Version, as Android-version and as an HTML-app. In the background works a server-software that handles the aggregation of the product database and the transfer to the kassenterminal.

And since you managed to read until here, have a picture of the kassenterminal as a reward:


Bei Problemen, Wutanfällen, Lob bitte Mail an kasse@fablab.fau.de
Kleinste Rückgabeeinheit 5ct
Kassenterminal V 43.01
Ich bin nicht aus. Einfach berühren!
LEITZ
FAU FabLab
Ausleihlisten

(Unfortunately without banana for scale. However, the folder is DIN A4.)

CHAPTER 2

API reference

2.1 Subpackages

2.1.1 `UI`: User Interface components and dialogs

Subpackages

`uic_generated`: autogenerated UI code

FabLabKasse.UI.uic_generated.CheckCartAfterImportDialog module

FabLabKasse.UI.uic_generated.Kassenterminal module

FabLabKasse.UI.uic_generated.KeyboardDialog module

FabLabKasse.UI.uic_generated.LoadFromMobileAppDialog module

FabLabKasse.UI.uic_generated.PaymentMethodDialog module

FabLabKasse.UI.uic_generated.PayupCashDialog module

FabLabKasse.UI.uic_generated.PayupManualDialog module

FabLabKasse.UI.uic_generated.SelectClientDialog module

FabLabKasse.UI.uic_generated.StatisticsDialog module

FabLabKasse.UI.uic_generated.icons_rc module

Module contents

Submodules

FabLabKasse.UI.ClientDialogCode module

FabLabKasse.UI.CheckCartAfterImportDialogCode module

..automodule:: FabLabKasse.UI.CheckCartAfterImportDialogCode

members

undoc-members

show-inheritance

FabLabKasse.UI.GUIHelper module

`FabLabKasse.UI.GUIHelper.`**`resize_table_columns`**(*table*, *widths*)

resize Qt table columns by the weight factors specified in widths, using the whole width (excluding scrollbar width)

FabLabKasse.UI.KeyboardDialogCode module

FabLabKasse.UI.LoadFromMobileAppDialogCode module

FabLabKasse.UI.MyQLineEdit module

FabLabKasse.UI.CartTableView module

FabLabKasse.UI.PaymentMethodDialogCode module

FabLabKasse.UI.PayupCashDialogCode module

FabLabKasse.UI.PayupManualDialogCode module

FabLabKasse.UI.compile_all module

Module contents

2.1.2 `cash_payment`

Infrastructure for accepting real coins and banknotes, including logging.

Subpackages

FabLabKasse.cashPayment.client package

Subpackages

Submodules

FabLabKasse.cashPayment.client.PaymentDeviceClient module

Client for accessing a cash device driver.

`class` `FabLabKasse.cashPayment.client.PaymentDeviceClient.`**`PaymentDeviceClient`**(*cmd*, *options*)

Bases: `object`

Client for accessing a cash device driver. It starts a new python process ("server") for the specified device driver. It uses non-blocking communication and talks to the server process using stdin/stdout.

`accept`(*maximumPayin*)

accept up to maximumPayin money, until stopAccepting() is called

poll() must be called before other actions are taken

canAccept()
does the device support accept commands?

(If this function has not returned True/False once before, it may only be called while no operation is in progress and will raise an Exception otherwise.)

return values and usage:

- None: please call the function again later. The answer has not yet been received from the device. No other actions (dispense/accept/possibleDispense) may be called until a non-None value was returned! call poll() repeatedly until `canAccept() != None`
- **True/False: Does (not) support accepting. (Now the answer is cached and may the function may be called ag**
  always)

  Return type boolean | None

dispense(*amount*)
Dispense up to the requested amount of money (as much as possible)

- Wait until hasStopped() is true, then retrieve the paid out value with getFinalAmountAndReset()
- **An intermediate value (as a progess report) can be retrieved with getCurrentAmount, but the operation cann**
  be aborted.
- If you want to make sure that enough is available, see possibleDispense()

empty()
start service-mode emptying

The implementation of this modes is device specific:

- If the device has an inaccessible storage, it should move the contents to the cashbox so that it can be taken out for counting.
- If available, manual payout buttons are enabled.

usage:

- call empty()
- sleep, do something else, whatever you want…
- call poll() at least once before the next step:
- as soon as you want to stop, call stopEmptying()
- call hasStopped() until it returns True
- then call getFinalAmountAndReset()

getCurrentAmount()
how much has currently been paid in? (value is not always up-to-date, but will not be higher than the actual value)

getFinalAmountAndReset()
call this as soon as hasStopped() is true. this returns the final amount paid in/out (negative for payout)

hasStopped()
returns True as soon as the operation (accept/dispense) has finished

poll()
update internal status

call this regularly

Raise Exception if the device crashed - do not try to recover from this exception, or the result of any following calls will be undefined

possibleDispense()
how much can be paid out?

(function may only be called while no operation is in progress, will raise Exception otherwise)

return value:

- `None`: request in progress, please call the function again until it does not return None. No other actions (dispense/accept/canPayout) may be called until a non-None value was returned! Call poll() repeatedly until possibleDispense()!=None.
- **[maximumAmount, remainingAmount]: This one non-None response is not cached, another call will sen**
  - maximumAmount (int): the device has enough money to pay out any amount up to maximumAmount
  - remainingAmount (int): How much money could be remaining at worst, if canBePaid==True? This is usually a per-device constant. remainingAmount will be == 0 for a small-coins dispenser that includes 1ct.

Important: it can be still possible to payout more, but not any value above maximumAmount!

For example a banknote dispenser filled with 2*10€ and 5*100€ bills will return:

`possibleDispense() == [2999, 999]` which means "can payout any value in 0…29,99€ with an unpaid rest of <= 9,99€"

But it can still fulfill a request of exactly 500€!

Return type None | [int, int]

`stopAccepting``()`
stop accepting (does not work immediately - some payins may be possible!)

Return type None

`stopEmptying``()`
end the mode that was started by empty()

usage: see empty()

`updateAcceptValue``(`*maximumPayin*`)`
lower the amount of money that is accepted at maximum

this can be called while accept is active

example use case:

- Two payment devices should accept 50€ in total.
- 10€ were inserted into the first device -> update the second device to a maximum of 40€.

FabLabKasse.cashPayment.client.PaymentDevicesManager module

class `FabLabKasse.cashPayment.client.PaymentDevicesManager.`**`PaymentDevicesManager`**`(`*cfg*`)`
Bases: `object`

`abortPayin``()`

`canPayout``()`
returns values [totalMaximumRequest, totalRemaining]:

every requested amount <= totalMaximumRequest can be paid out, with an unpaid rest <= totalRemaining (the return value is only a conservative estimate, not the theoretical optimum)

Please warn the user if totalMaximumRequest is too low for the possible change if this function returns None, the value is still being fetched. In this case, sleep some time, then call poll() and then call the function again.

empty()
start service-empty mode

see PaymentDeviceClient.empty

Return type None

getCurrentAmount()
get intermediate amount, how much was paid in or out

getFinalAmount()
if stopped, return the final amount and reset to idle state

else, return None

payin(*requested*, *maximum*)

payout(*value*)

poll()
call repeatedly to update status

Return type None

startingUp()
return True if devices are still being started

No action methods may be called until this returns False

statusText()

stopEmptying()
exit service-empty mode

use getFinalAmount() afterwards

Return type None

class FabLabKasse.cashPayment.client.PaymentDevicesManager.**PaymentDevicesManagerTest**(*met*
Bases: `unittest.case.TestCase`

Test PaymentDevicesManager

test_canPayout_with_one_random_datapoint_on_example_server()
test the _canPayout_total() function with 10 random datapoints and the exampleserver (from example config)

FabLabKasse.cashPayment.client.PaymentDevicesManager.**demo**()
Simple demonstration using two exampleServer devices

Module contents

Server (device driver)

Subpackages

FabLabKasse.cashPayment.server.NV11 package

Submodules

FabLabKasse.cashPayment.server.NV11.NV11Device module

class `FabLabKasse.cashPayment.server.NV11.NV11Device.`**`ESSPDevice`**(*port*, *preshared-Key=81985526925837671*, *slaveID=0*)

Bases: `object`

low layer eSSP protocol - implements the network layer and all communication-related commands

class **`ByteStreamReader`**(*bytesList*)

Bases: `object`

read data values from a list of bytes

`assertFinished`()

`hasData`()

`readAscii`(*n*)

`readByte`()

`readData`(*n*)

`readUnsigned`(*numBytes*, *littleEndian*)

`readUnsigned24`()

`readUnsigned24BigEndian`()

`readUnsigned32`(*CheckOverrun=True*)

`readUnsigned32BigEndian`()

static **`unitTest`**()

class **`Helper`**

Bases: `object`

classmethod **`AsciiToBytes`**(*x*)

`CRC = <FabLabKasse.cashPayment.server.NV11.crc_algorithms.Crc object>`

classmethod **`Unsigned32ToBytes`**(*x*)

static **`Unsigned64ToBytes`**(*x*)

static **`UnsignedToBytes`**(*x*, *n*)

static **`byteArrayToString`**(*data*)

classmethod **`crc`**(*data*)

classmethod **`splitBytes`**(*uint16*)

static **`stringToByteArray`**(*string*)

static **`unitTest`**()

class Response(*data*)

Bases: `object`

status + data

getData()

getDataStream()

getStatus()

isEncrypted()

isHardFail()

isOkay()

isSoftFail()

statusString()

statusStrings = {-1: 'decoded response contains no data', 126: 'Encrypted Data

command(*data*, *allowSoftFail=False*, *encrypted=True*)

debug(*s*)

debug2(*s*)

decryptResponse(*r*)

encryptData(*data*)

error(*s*)

flushRead()

initCrypto(*presharedKey*)

log(*s*)

printDebug(*s*, *debugLevel*)

read()

resendLast()

reset()

send(*data*)

setEnabled(*enabled*)

unencryptedCommand(*data*, *allowSoftFail=False*)

warn(*s*)

class `FabLabKasse.cashPayment.server.NV11.NV11Device.`**NV11Device**(*port*, *presharedKey=81985526925837671*, *slaveID=0*)

Bases: *`FabLabKasse.cashPayment.server.NV11.NV11Device.ESSPDevice`*

Interface client for Innovative Technology NV11 banknote validator/changer with eSSP Protocol

empty()

getChannelValue(*channelId*, *reportedValue=False*)

getPayoutValues()

get values of notes on payout stack. The last one of these is on top of the stack and will be paid out by the payout-command.

poll()

setEnabledChannels(*enabledChannels=None*, *upTo=0*)
enable cash input for certain channels (denominations). Use either of the two parameters. If both are used, the result will be combined with logical `or`.

Parameters

- **enabledChannels** (*list[int]*) – IDs of channels to explicitly enable, even if their denominaion is below the value of `upTo`.
- **upTo** (*int*) – maximum allowed denomination, or 0

setRouteToPayout(*values*)
route all notes in the given list of values to the payout-store. others will be directly put to the cashbox and are not availble for return.

stackFromPayout()
move the current note from payout store to the cashbox, so that a smaller note that isn't on top of the stack can be paid out. does not check if this is useful, these checks need to be done at a higher level!

tryPayout(*value*)

class FabLabKasse.cashPayment.server.NV11.NV11Device.**NV11DeviceTest**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Test NV11Device class

test_ESSPDevice_ByteStreamReader()
unittest: test the ByteStreamReader of ESSSPDevice

test_ESSPDevice_crc()
unittest: check crc of ESSPDevice.Helper

FabLabKasse.cashPayment.server.NV11.crc_algorithms module

CRC algorithms implemented in Python. If you want to study the Python implementation of the CRC routines, then this is a good place to start from.

The algorithms Bit by Bit, Bit by Bit Fast and Table-Driven are implemented.

This module can also be used as a library from within Python.

Examples

This is an example use of the different algorithms:

```
>>> from crc_algorithms import Crc
>>>
>>> crc = Crc(width=16, poly=0x8005,
...           reflect_in=True, xor_in=0x0000,
...           reflect_out=True, xor_out=0x0000)
>>> print("0x%x" % crc.bit_by_bit("123456789"))
>>> print("0x%x" % crc.bit_by_bit_fast("123456789"))
>>> print("0x%x" % crc.table_driven("123456789"))
```

class FabLabKasse.cashPayment.server.NV11.crc_algorithms.**Crc**(*width*, *poly*, *reflect_in*, *xor_in*, *reflect_out*, *xor_out*, *table_idx_width=None*)
Bases: `object`

A base class for CRC routines.

bit_by_bit (*in_data*)
: Classic simple and slow CRC implementation. This function iterates bit by bit over the augmented input message and returns the calculated CRC value at the end.

bit_by_bit_fast (*in_data*)
: This is a slightly modified version of the bit-by-bit algorithm: it does not need to loop over the augmented bits, i.e. the Width 0-bits which are appended to the input message in the bit-by-bit algorithm.

gen_table ()
: This function generates the CRC table used for the table_driven CRC algorithm. The Python version cannot handle tables of an index width other than 8. See the generated C code for tables with different sizes instead.

reflect (*data*, *width*)
: reflect a data word, i.e. reverts the bit order.

table_driven (*in_data*)
: The Standard table_driven CRC algorithm.

Module contents

FabLabKasse.cashPayment.server.helpers package

Submodules

FabLabKasse.cashPayment.server.helpers.banknote_stack_helper module

helper for stack-based banknote payout systems. see *BanknoteStackHelper*

class `FabLabKasse.cashPayment.server.helpers.banknote_stack_helper.`**BanknoteStackHelper** (*a*

Bases: `object`

helper class for stack-based banknote payout systems. Such a system has a stack of banknotes from which the top one can be

- either paid out to the client (action "payout")
- or be irrevocably put away into a cashbox (action "stack"), from where it cannot be retrieved again for payout.

From the programmer's point of view, this stack is a list of banknotes, from which only the last one (stack.pop()) can be accessed.

This class makes the relevant decisions whether to pay out or stack away the current note. It also offers a matching implementation for `FabLabKasse.cashPayment.server.CashServer.getCanPayout()`

Parameters **accepted_rest** – see `FabLabKasse.cashPayment.server.cashServer.CashServer.getCanPayout()`

can_payout (*payout_stack*)
: implementation for CashServer.getCanPayout()

get_next_payout_action (*payout_stack*, *requested_payout*)
: which action should be taken next? (see the documentation for BanknoteStackHelper for more context information)

class `FabLabKasse.cashPayment.server.helpers.banknote_stack_helper.`**BanknoteStackHelperTe**

Bases: `unittest.case.TestCase`

Tests the banknote stack helper class

test_with_fixed_values ()

test_with_several_random_values()
: unittest: calls several integrated functions of banknote stack helper as test with several random numbers

class FabLabKasse.cashPayment.server.helpers.banknote_stack_helper.**BanknoteStackHelperTe**

Bases: *FabLabKasse.cashPayment.server.helpers.banknote_stack_helper.BanknoteStackHelper*

unittest methods for BanknoteStackHelper

classmethod get_random_payout_parameters(*random_generator*, *payout_stack=None*, *requested_payout=None*)
: determine parameters for payout_stack and requested_payout

Parameters **random_generator** (*random.Random*) – RNG instance for calculating pseudorandom test parameters

unittest_payout(*random_generator*)
: test one random set of parameters for BanknoteStackHelper.can_payout(), BanknoteStackHelper.get_next_payout_action()

Parameters **random_generator** (*random.Random*) – RNG instance for calculating pseudorandom test parameters

Return type None

Raise AssertionError if the test failed

unittest_payout_forced_stacking(*random_generator*)
: test one random set of parameters for BanknoteStackHelper._forced_stacking_is_helpful()

Parameters **random_generator** (*random.Random*) – RNG instance for calculating pseudorandom test parameters

Return type None

Raise AssertionError if the test failed

FabLabKasse.cashPayment.server.helpers.coin_payout_helper module

helper functions for multi-tube coin dispensers

FabLabKasse.cashPayment.server.helpers.coin_payout_helper.**get_possible_payout**(*coins*, *max_number_*

get possible amount of payout and remaining rest

This implementation returns a lower bound. This means that the theoretical maximum possible amount can be higher (and will be often).

Parameters

- **coins** (*list[(int, int)]*) – list of tuples (value, count), sorted descending by value. The function still works if a value occurs twice (e.g. if you have a dispenser with two separate tubes for 1€ coins).
- **max_number_of_coins** (*int*) – approximate upper limit for the number of coins - this limits the reporting of the maximum possible amount of payout, so that a device won't say it is able to pay out 50€, if that is only possible with 500x 0,10€ coins.

  Please note that if you limit this too much, the user will be warned that there is not enough change money. Some amount is necessary, especially in cooperation with other devices (e.g. banknotes + coins).

Returns as defined by `FabLabKasse.cashPayment.server.getCanPayout()`

FabLabKasse.cashPayment.server.helpers.test_coin_payout_helper module

tests for `coin_payout_helper`

class `FabLabKasse.cashPayment.server.helpers.test_coin_payout_helper.`**`CoinPayoutHelperTes`**

Bases: `unittest.case.TestCase`

Tests for `coin_payout_helper`

`test_get_possible_payout`(*coins=HypothesisProvided(value=st_coins()), requested_fraction=HypothesisProvided(value=floats(min_value=0, max_value=1)), coin_limit_fraction=HypothesisProvided(value=floats(min_value=0, max_value=1))*)

test `coin_payout_helper.get_possible_payout()` for a given state of available coins

`FabLabKasse.cashPayment.server.helpers.test_coin_payout_helper.`**`simulate_payout`**(*coins, requested*)

get payout amount for a very simple simulated payout strategy ('greedy strategy', just pay out largest coins first, without 'coin splitting' to get rid of smaller ones)

Parameters

- **coins** (*list[(int, int)]*) – see `coin_payout_helper.get_possible_payout()`
- **requested** (*int*) – requested amount

Module contents

FabLabKasse.cashPayment.server.mdbCash package

Submodules

FabLabKasse.cashPayment.server.mdbCash.mdb module

exception `FabLabKasse.cashPayment.server.mdbCash.mdb.`**`BusError`**

Bases: `exceptions.Exception`

exception `FabLabKasse.cashPayment.server.mdbCash.mdb.`**`InterfaceHardwareError`**

Bases: `exceptions.Exception`

class `FabLabKasse.cashPayment.server.mdbCash.mdb.`**`MdbCashDevice`**(*port, addr=1, extensionConfig=None*)

Bases: `object`

`ACK = 0`

`BUSY = 'busy'`

`CMD_COIN_TYPE = 4`

`CMD_DISPENSE = 5`

`CMD_EXPANSION = 7`

`CMD_POLL = 3`

`CMD_RESET = 0`

`CMD_SETUP = 1`

CMD_TUBE_STATUS = 2

ERROR = 'error'

IGNORE = 'ignore'

JUST_RESET = 'just reset'

NAK = 255

RET = 170

WARNING = 'warning'

checksum(*data*)

cmd(*command*, *data=None*)

dispenseCoin(*type*, *amount*)

dispenseValue(*maximumDispense*)

error(*s*)

extensionCmd(*data*)
in addition to the MDB commands, the interface hardware provides extension commands for other features (LEDs, hopper, …). Failure on these commands is not tolerated.

flushRead()

getPossiblePayout()

getSetup()

getSortedCoinValues()

getTubeStatus()

getValue(*type*)

log(*s*)

poll(*wasJustReset=False*)
get events from device. :param wasJustReset: set this to True at the first poll after the RESET command

printDebug(*s*, *debugLevel*)

read()

reset()

serialCmd(*text*)

setAcceptCoins(*acceptCoins*, *manualDispenseEnabled=False*)

setLEDs(*leds*)
set RGB-LED color via extension command, if it is enabled in the extensionConfig. :param leds: list of two LED color values. color value: RR GG BB in hex plus a mode of N (normal) or special modes B (blink) or T (timeout: switch off after 20 sec) e.g. "00FF00N" = green normal, "FF0000B" = red blink, "0000FFT" = blue with timeout (will switch off after 20sec or the next command)

statusEvents = {1: ['Escrow request', 'ignore'], 2: ['Payout Busy', 'busy'], 3:

tryDispenseCoinFromExternalHopper()
dispense a coin from an external non-MDB hopper connected directly to the interface board. :returns: False if it failed (or no external hopper is enabled), True if one coin was dispensed

warn(*s*)

exception `FabLabKasse.cashPayment.server.mdbCash.mdb.`**MissingResetEventError**
Bases: `exceptions.Exception`

Module contents

Device drivers

cashServer (abstract base class)

exampleServer

mdbCoinChanger

nv11

FabLabKasse.cashPayment.server.hex module

`FabLabKasse.cashPayment.server.hex.`**`hex`**`(`*x*`)`

Module contents

Submodules

`cashState`: logging

This module also has a command-line interface that can be accessed as `./cash` from the FabLabKasse directory.

FabLabKasse.cashPayment.listSerialPorts module

`FabLabKasse.cashPayment.listSerialPorts.`**`main`**`()`

Module contents

2.1.3 Libraries

FabLabKasse.libs

Subpackages

FabLabKasse.libs.escpos package

Submodules

FabLabKasse.libs.escpos.constants module

ESC/POS Commands (Constants)

FabLabKasse.libs.escpos.escpos module

@author: Manuel F Martinez <manpaz@bashlinux.com> @organization: Bashlinux @copyright: Copyright (c) 2012 Bashlinux @license: GPL

`class` `FabLabKasse.libs.escpos.escpos.`**`Escpos`**

ESC/POS Printer object

barcode (*code*, *bc*, *width*, *height*, *pos*, *font*)
Print Barcode

cashdraw (*pin*)
Send pulse to kick the cash drawer

control (*ctl*)
Feed control sequences

cut (*mode=''*)
Cut paper

device = None

hw (*hw*)
Hardware operations

image (*path_img*)
Open image file

set (*align='left'*, *font='a'*, *type='normal'*, *width=1*, *height=1*)
Set text properties

text (*txt*)
Print alpha-numeric text

FabLabKasse.libs.escpos.exceptions module

ESC/POS Exceptions classes

exception FabLabKasse.libs.escpos.exceptions.**BarcodeCodeError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

exception FabLabKasse.libs.escpos.exceptions.**BarcodeSizeError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

exception FabLabKasse.libs.escpos.exceptions.**BarcodeTypeError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

exception FabLabKasse.libs.escpos.exceptions.**CashDrawerError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

exception FabLabKasse.libs.escpos.exceptions.**Error** (*msg*, *status=None*)
Bases: exceptions.Exception

Base class for ESC/POS errors

exception FabLabKasse.libs.escpos.exceptions.**ImageSizeError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

exception FabLabKasse.libs.escpos.exceptions.**TextError** (*msg=''*)
Bases: *FabLabKasse.libs.escpos.exceptions.Error*

FabLabKasse.libs.escpos.printer module

@author: Manuel F Martinez <manpaz@bashlinux.com> @organization: Bashlinux @copyright: Copyright (c) 2012 Bashlinux @license: GPL

class FabLabKasse.libs.escpos.printer.**File** (*devfile='/dev/usb/lp0'*)
Bases: *FabLabKasse.libs.escpos.escpos.Escpos*

Define Generic file printer

open ()
Open system file

class FabLabKasse.libs.escpos.printer.**Network**(*host*, *port=9100*)

Bases: *FabLabKasse.libs.escpos.escpos.Escpos*

Define Network printer

open()

Open TCP socket and set it as escpos device

Module contents

FabLabKasse.libs.flickcharm package

Submodules

FabLabKasse.libs.flickcharm.flickcharm module

FabLabKasse.libs.flickcharm.main module

Module contents

FabLabKasse.libs.process_helper package

Submodules

FabLabKasse.libs.process_helper.asyncproc module

class FabLabKasse.libs.process_helper.asyncproc.**Process**(**params*, ***kw-params*)

Bases: `object`

Manager for an asynchronous process. The process will be run in the background, and its standard output and standard error will be collected asynchronously.

Since the collection of output happens asynchronously (handled by threads), the process won't block even if it outputs large amounts of data and you do not call Process.read*().

Similarly, it is possible to send data to the standard input of the process using the write() method, and the caller of write() won't block even if the process does not drain its input.

On the other hand, this can consume large amounts of memory, potentially even exhausting all memory available.

Parameters are identical to subprocess.Popen(), except that stdin, stdout and stderr default to subprocess.PIPE instead of to None. Note that if you set stdout or stderr to anything but PIPE, the Process object won't collect that output, and the read*() methods will always return empty strings. Also, setting stdin to something other than PIPE will make the write() method raise an exception.

closeinput()

Close the standard input of a process, so it receives EOF.

kill(*signal*)

Send a signal to the process. Raises OSError, with errno set to ECHILD, if the process is no longer running.

pid()

Return the process id of the process. Note that if the process has died (and successfully been waited for), that process id may have been re-used by the operating system.

read()

Read data written by the process to its standard output.

readboth()
Read data written by the process to its standard output and error. Return value is a two-tuple (stdout-data, stderr-data).

WARNING! The name of this method is ugly, and may change in future versions!

readerr()
Read data written by the process to its standard error.

terminate(*graceperiod=1*)
Terminate the process, with escalating force as needed. First try gently, but increase the force if it doesn't respond to persuassion. The levels tried are, in order:

- close the standard input of the process, so it gets an EOF.
- send SIGTERM to the process.
- send SIGKILL to the process.

terminate() waits up to GRACEPERIOD seconds (default 1) before escalating the level of force. As there are three levels, a total of (3-1)*GRACEPERIOD is allowed before the process is SIGKILL:ed. GRACEPERIOD must be an integer, and must be at least 1.

If the process was started with stdin not set to PIPE, the first level (closing stdin) is skipped.

wait(*flags=0*)
Return the process' termination status.

If bitmask parameter 'flags' contains os.WNOHANG, wait() will return None if the process hasn't terminated. Otherwise it will wait until the process dies.

It is permitted to call wait() several times, even after it has succeeded; the Process instance will remember the exit status from the first successful call, and return that on subsequent calls.

write(*data*)
Send data to a process's standard input.

FabLabKasse.libs.process_helper.asyncproc.**with_timeout**(*timeout*, *func*, **args*, ***kwargs*)
Call a function, allowing it only to take a certain amount of time.

param timeout The time, in seconds, the function is allowed to spend. This must be an integer, due to limitations in the SIGALRM handling.

param func The function to call.

param *args Non-keyword arguments to pass to func.

param **kwargs Keyword arguments to pass to func.

Upon successful completion, with_timeout() returns the return value from func. If a timeout occurs, the Timeout exception will be raised.

If an alarm is pending when with_timeout() is called, with_timeout() tries to restore that alarm as well as possible, and call the SIGALRM signal handler if it would have expired during the execution of func. This may cause that signal handler to be executed later than it would normally do. In particular, calling with_timeout() from within a with_timeout() call with a shorter timeout, won't interrupt the inner call. I.e., *with_timeout(5, with_timeout, 60, time.sleep, 120)*' won't interrupt the time.sleep() call until after 60 seconds.

exception FabLabKasse.libs.process_helper.asyncproc.**Timeout**
Bases: exceptions.Exception

Exception raised by with_timeout() when the operation takes too long.

FabLabKasse.libs.process_helper.nonblockingProcess module

`FabLabKasse.libs.process_helper.nonblockingProcess.`**`demo`**`()`

small example that communicates with bc, the commandline calculator. Note that readline() never blocks!

`class` `FabLabKasse.libs.process_helper.nonblockingProcess.`**`nonblockingProcess`**`(`*cmd, env=None*`)`

Bases: `object`

non-blocking subprocess, based on asyncproc

`hasLine``()`

`isAlive``()`

`readline``()`

read line from process stdout, if available, else return None

`write``(`*string*`)`

Module contents

FabLabKasse.libs.pxss package

Submodules

FabLabKasse.libs.pxss.pxss module

`class` `FabLabKasse.libs.pxss.pxss.`**`Display`**

Bases: `_ctypes.Structure`

`bitmap_bit_order`

Structure/Union member

`bitmap_pad`

Structure/Union member

`bitmap_unit`

Structure/Union member

`byte_order`

Structure/Union member

`db`

Structure/Union member

`default_screen`

Structure/Union member

`display_name`

Structure/Union member

`ext_data`

Structure/Union member

`fd`

Structure/Union member

`last_request_read`

Structure/Union member

`max_request_size`

Structure/Union member

nformats
Structure/Union member

nscreens
Structure/Union member

pixmap_format
Structure/Union member

private1
Structure/Union member

private10
Structure/Union member

private11
Structure/Union member

private12
Structure/Union member

private13
Structure/Union member

private14
Structure/Union member

private15
Structure/Union member

private2
Structure/Union member

private3
Structure/Union member

private4
Structure/Union member

private5
Structure/Union member

private6
Structure/Union member

private8
Structure/Union member

private9
Structure/Union member

proto_major_version
Structure/Union member

proto_minor_version
Structure/Union member

qlen
Structure/Union member

release
Structure/Union member

request
Structure/Union member

resource_alloc
Structure/Union member

screens
Structure/Union member

vendor
Structure/Union member

class `FabLabKasse.libs.pxss.pxss.`**IdleTracker**(*when_idle_wait=5000*, *when_disabled_wait=120000*, *idle_threshold=60000*)

Keeps track of idle times, screensaver state, and tells you when you to querying it for the next idle time. All times are in milliseconds. IdleTracker indicates a change in state when your idle time exceeds a certain threshold. See also XSSTracker.

check_idle()

suggested_time_till_next_check and idle_time is in milliseconds.

state_change is one of:

- None - No change in state
- "idle" - user is idle (idle time is greater than idle threshold)
- "unidle" - user is not idle (idle time is less than idle threshold)
- "disabled" - idle time not available

Note that "disabled" will be returned every time there is an error. :returns: tuple (state_change, suggested_time_till_next_check, idle_time)

class `FabLabKasse.libs.pxss.pxss.`**Screen**

Bases: `_ctypes.Structure`

backing_store
Structure/Union member

black_pixel
Structure/Union member

cmap
Structure/Union member

default_gc
Structure/Union member

depths
Structure/Union member

display
Structure/Union member

ext_data
Structure/Union member

height
Structure/Union member

mheight
Structure/Union member

min_maps
Structure/Union member

mwidth
Structure/Union member

ndepths
Structure/Union member

root
Structure/Union member

root_depth
Structure/Union member

root_input_mask
Structure/Union member

root_visual
Structure/Union member

save_unders
Structure/Union member

white_pixel
Structure/Union member

width
Structure/Union member

class FabLabKasse.libs.pxss.pxss.**XSSTracker**(*when_idle_wait=5000*, *when_disabled_wait=120000*)
Keeps track of idle times, screensaver state, and tells you when you to querying it for the next idle time. All times are in milliseconds. XSSTracker indicates a change in state when your screensaver activates. See also IdleTracker.

check_idle()
suggested_time_till_next_check and idle_time is in milliseconds.

state_change is one of:

- None - No change in state
- "idle" - screensaver has turned on since user is now idle
- "unidle" - screensaver has turned off since user is no longer idle
- "disabled" - screensaver is disabled or extension not present

Note that if the screensaver is disabled, it will return "disabled" every time. :returns: tuple (state_change, suggested_time_till_next_check, idle_time)

class FabLabKasse.libs.pxss.pxss.**XScreenSaverInfo**
Bases: _ctypes.Structure

eventMask
Structure/Union member

idle
Structure/Union member

kind
Structure/Union member

state
Structure/Union member

til_or_since
Structure/Union member

window
Structure/Union member

FabLabKasse.libs.pxss.pxss.**get_info**(*p_display=None*, *default_root_window=None*, *p_info=None*)

Module contents

Submodules

FabLabKasse.libs.random_lists module

randomly built lists with randomness taken from random.choice()

Warning: not cryptographically secure!

`FabLabKasse.libs.random_lists.`**`random_choice_list`**`(`*random_generator*, *possible_elements*, *number_of_elements*`)`

return a random list with len(list)==number_of_elements, list[i] in possible_elements (duplicates are possible)

Parameters

- **`random_generator`** (*`random.Random`*) – RNG instance
- **`possible_elements`** (*`list`*) – list elements to choose from
- **`number_of_elements`** (*`int`*) – length of resulting list

`FabLabKasse.libs.random_lists.`**`random_integer_list`**`(`*random_generator*, *integer_range*, *number_of_elements*`)`

return a list of length number_of_elements with elements in the range integer_range[0] <= element <= integer_range[1]

Parameters

- **`int) integer_range`** (*`(int,`*) – range (min, max) – ends are included
- **`random_generator`** (*`random.Random`*) – RNG instance
- **`number_of_elements`** (*`int`*) – length of resulting list

Module contents

2.1.4 FabLabKasse.scripts package

Submodules

FabLabKasse.scripts.logWatchAndCleanup module

a cronjob for cleaning up and gzipping old logfiles

- report errors and warnings in the newest logfiles (the current one, foo.log, and the archive files foo.log.2012-12-31 created after the last run of this script [i.e. the non-gzipped ones])
- gzip old logfiles and remove too old ones

run this script by starting logWatchAndCleanup.sh which lives in the same directory

# it is recommended to run this script before midnight, because the logs wrap over at midnight and might change in the middle of running this script

`FabLabKasse.scripts.logWatchAndCleanup.`**`main`**`()`

Module contents

2.1.5 `shopping`: backend for articles, payment etc.

Subpackages

FabLabKasse.shopping.backend package

Subpackages

FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools package

Submodules

FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.exportConsumptionMoney module

FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP module

class `FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP.`**`ca`**

Bases: `object`

`FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP.`**`importPr`**

`FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP.`**`main`**`()`

`FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP.`**`saveToDi`**

`FabLabKasse.shopping.backend.legacy_offline_kassenbuch_tools.importProdukteOERP.`**`str_to_i`**

Module contents

Submodules

FabLabKasse.shopping.backend.abstract module

abstract implementations of shopping and clients

base class and interface definition for specific implementations (see other files in this folder)

class `FabLabKasse.shopping.backend.abstract.`**`AbstractClient`**`(`*client_id=None, name=''*`)`

Bases: `object`

a client that can pay by pin

`get_debt``()`

how much is the current debt (<0 = client has pre-paid)

`get_debt_limit``()`

how much is the limit for the debt that may not be exceeded

test_pin(*pin*)
is the given pin (4-digit string) correct and the client enabled for paying?

class `FabLabKasse.shopping.backend.abstract.`**AbstractShoppingBackend**(*cfg*)
Bases: `object`

manages products, categories and orders (cart)

add_order_line(*prod_id*, *qty*, *comment=None*)
add product to cart

if not all values are allowed, `qty` is rounded *up* to the next possible amount.

The user should only be asked for a comment by the GUI if `self.product_requires_text_entry(prod_id) == True`

Parameters

- **prod_id** (*int*) – product
- **qty** (*Decimal*) – amount of product
- **comment** (*(basestring, None)*) – textual comment from the user, or None.

Raise ProductNotFound

create_order()
create a new order and return its id

delete_current_order()
delete currently selected order, implies set_current_order(None)

delete_order_line(*order_line_id*)
delete product from cart

format_money(*amount*)

format_qty(*qty*)

get_category_path(*current_category*)
return the category path from the root to the current category, *excluding* the root category

[child_of_root, . . . , parent_of_current, current_category]

Return type list(*Category*)

get_current_order()
get selected order (or return 0 if switching between multiple orders is not supported)

get_current_total()

Returns total sum of current order

Return type Decimal

Note: The internal rounding *must* be consistent, which is needed by :class: FabLabKasse.shopping.payment_methods. That means that x,xx5 € must always be rounded up or always down. “Fair rounding” like Decimal.ROUND_HALF_EVEN is not allowed.

For example:

- add article costing 1,015 € -> get_current_total == x
- add article costing 0,990 € -> get_current_total == x + 0,99

This would not be true with the fair strategy “round second digit to even value if the third one is exactly 5” (1,02€ and 2,00€).

get_order_line(*order_line_id*)
get order line of current order

get_order_lines()
return current order lines

Return type *OrderLine*

`get_products`(*current_category*)
return products in current category

Return type list(*Product*)

`get_root_category`()
return id of root category

`get_subcategories`(*current_category*)
return list(Category) of subclasses of the given category-id.

`list_clients`()
returns all selectable clients in a dict {id: Client(id), . . . }

`pay_order`(*method*)
store payment of current order to database :param method: payment method object, whose type is used to determine where the order should be stored in the database method.amount_paid - method.amount_returned is how much money was gained by this sale, must be equal to self.get_current_total()

`pay_order_on_client`(*client*)
charge the order on client's account

Parameters **`client`** – AbstractClient

Raises DebtLimitExceeded when the client's debt limit would be exceeded

`print_receipt`(*order_id*)
print the receipt for a given, already paid order_id

The receipt data must be stored in the backend, because for accountability reasons all receipt texts need to be stored anyway.

`product_requires_text_entry`(*prod_id*)
when adding prod_id, should the user be asked for a text entry for entering comments like his name?

`static round_money`(*value*)
rounds money in Decimal representation to 2 places

Main purpose is shopping.backend.abstract.AbstractShoppingBackend.get_current_total(), since round() does behave weird. But maybe there are other applications too.

Parameters **`value`** (*`float | Decimal`*) – an amount of money to be rounded

Returns money, rounded to 2 digits

Return type Decimal

```
>>> AbstractShoppingBackend.round_money(Decimal('0.005'))
Decimal('0.01')
>>> AbstractShoppingBackend.round_money(Decimal('0.004'))
Decimal('0.00')
```

`search_from_text`(*searchstr*)
search searchstr in products and categories :return: tuple (list of categories, products for table)

Return type list(*Product*)

`search_product_from_code`(*code*)
search via barcode, PLU or similar unique-ID entry. code may be any string

Returns product id

Raises ProductNotFound() if nothing found

`set_current_order`(*order_id*)
switch to another order (when the backend supports multiple orders)

update_quantity(*order_line_id*, *amount*)
change quantity of order-line.

if not all float values are allowed, round upvalue to the next possible one

class FabLabKasse.shopping.backend.abstract.**AbstractShoppingBackendTest**(*methodName='runTest*
Bases: unittest.case.TestCase

test the AbstractShoppingBackend class

TODO extend this test

test_round_money_subcent_values()
test the money-rounding function

the test checks the rounding of subcent values

class FabLabKasse.shopping.backend.abstract.**Category**(*categ_id*, *name*, *parent_id=None*)
Bases: object

represents a category of Products

exception FabLabKasse.shopping.backend.abstract.**DebtLimitExceeded**
Bases: exceptions.Exception

exception raised by pay_order_on_client: order not paid because the debt limit would have been exceeded

class FabLabKasse.shopping.backend.abstract.**OrderLine**(*order_line_id*, *qty*, *unit*, *name*, *price_per_unit*, *price_subtotal*, *delete_if_zero_qty=True*)
Bases: object

one order line (roughly equal to a product in a shopping cart, although there may be multiple entries for one product)

Parameters

- **id** – id of order-line, *must be unique and non-changing* inside one Order() (if None: autogenerate id)
- **qty** (*Decimal*) – amount ("unlimited" number of digits is okay)
- **unit** (*unicode*) – product unit of sale
- **name** (*unicode*) – product name
- **price_per_unit** (*Decimal*) – price for one unit
- **price_subtotal** (*Decimal*) – price for qty * unit of this product
- **delete_if_zero_qty** (*boolean*) – if the qty is zero and the user starts adding something else, then remove this line

  [usually True, set to False for products that also may as comment limes costing nothing]

exception FabLabKasse.shopping.backend.abstract.**PrinterError**
Bases: exceptions.Exception

cannot print receipt

class FabLabKasse.shopping.backend.abstract.**Product**(*prod_id*, *name*, *price*, *unit*, *location*, *categ_id=None*, *qty_rounding=0*, *text_entry_required=False*)
Bases: object

simple representation for a product

Parameters

- **prod_id** (*int*) – numeric unique product ID
- **categ_id** (*int | None*) – category ID of product, or None if the product is not directly visible

  TODO hide these products from search, or a more explicit solution
- **name** (*unicode*) – Name of product
- **location** (*unicode*) – Location of product (shown to the user)
- **unit** (*unicode*) – Unit of sale for this product (e.g. piece, kilogram)
- **price** (*Decimal*) – price for one unit of this product
- **qty_rounding** (*int | Decimal*) – Product can only be bought in multiples of this quantity, user (GUI) input will be rounded/truncated to the next multiple of this.

  Set to 0 so that the product can be bought in arbitrarily small quantities.

  example: you cannot buy half a t-shirt, so you set qty_rounding = 1

  handling this is responsibility of the shopping backend

exception FabLabKasse.shopping.backend.abstract.**ProductNotFound**

Bases: exceptions.Exception

requested product not found

FabLabKasse.shopping.backend.abstract.**basicUnitTests**(*shopping_backend*)

FabLabKasse.shopping.backend.abstract.**float_to_decimal**(*number, digits*)

convert float to decimal with rounding and strict error tolerances

If the given number cannot be represented as decimal with an error within 1/1000 of the last digit, ValueError is raised.

Parameters

- **number** (*float | Decimal*) – a float that is nearly equal to a decimal number
- **digits** (*int*) – number of decimal places of the resulting value (max. 9)

Raise ValueError

```
>>> float_to_decimal(1.424, 3)
Decimal('1.424')
>>> float_to_decimal(0.7, 1)
Decimal('0.7')
```

FabLabKasse.shopping.backend.abstract.**format_money**(*amount*)

format float as money string

You should best use Decimal as input. TODO: make moneysign interchangeable

Parameters **amount** (*float|Decimal*) – amount of money

Returns amount formatted as string with Euro-Sign

Return type unicode

```
>>> format_money(1.23)
u'1,23 \u20ac'
>>> format_money(3.741)
u'3,741 \u20ac'
>>> format_money(42.4242)
u'42,424 \u20ac'
>>> format_money(5.8899)
u'5,89 \u20ac'
>>> format_money(Decimal('1.23'))
```

(continues on next page)

(continued from previous page)

```
u'1,23 \u20ac'
>>> format_money(Decimal('3.741'))
u'3,741 \u20ac'
>>> format_money(Decimal('42.4242'))
u'42,424 \u20ac'
>>> format_money(Decimal('5.8899'))
u'5,89 \u20ac'
```

`FabLabKasse.shopping.backend.abstract.`**`format_qty`**(*qty*)

format quantity (number) as string

Parameters **`qty`** – quantity in numbers

Returns string-representation of qty, decimal sep is dependent on locale

Return type unicode

```
>>> format_qty(5)
u'5'
```

`FabLabKasse.shopping.backend.abstract.`**`load_tests`**(*loader*, *tests*, *ignore*)

loader function to load the doctests in this module into unittest

FabLabKasse.shopping.backend.dummy module

FabLabKasse.shopping.backend.legacy_offline_kassenbuch module

FabLabKasse.shopping.backend.oerp module

`class` `FabLabKasse.shopping.backend.oerp.`**`Client`**(*client_id=None*, *name=”*)

Bases: *`FabLabKasse.shopping.backend.abstract.AbstractClient`*

oerp implementation of AbstractClient. do not instantiate this yourself, but please rather use Client.from_oerp or ShoppingBackend.list_clients

`classmethod from_oerp`(*client_id*, *oerp*)

read raw data from oerp record

`get_debt`()

how much is the current debt (<0 = client has pre-paid)

`get_debt_limit`()

how much is the limit for the debt that may not be exceeded

`class` `FabLabKasse.shopping.backend.oerp.`**`ShoppingBackend`**(*cfg*)

Bases: *`FabLabKasse.shopping.backend.abstract.AbstractShoppingBackend`*

OpenERP implementation of AbstractShoppingBackend

`add_order_line`(*prod_id*, *qty*, *comment=None*)

add product to cart

if not all values are allowed, `qty` is rounded *up* to the next possible amount.

The user should only be asked for a comment by the GUI if `self.product_requires_text_entry(prod_id) == True`

Parameters

- **`prod_id`** (*`int`*) – product
- **`qty`** (*`Decimal`*) – amount of product
- **`comment`** (*`(basestring, None)`*) – textual comment from the user, or None.

Raise ProductNotFound

`create_order`()
create a new order and return its id

`delete_current_order`()
delete currently selected order, implies set_current_order(None)

`delete_order_line`(*order_line_id*)
delete product from cart

`get_category_path`(*current_category*)
return the category path from the root to the current category, *excluding* the root category

[child_of_root, . . . , parent_of_current, current_category]

Return type list(*Category*)

`get_current_order`()
get selected order (or return 0 if switching between multiple orders is not supported)

`get_current_total`()

Returns total sum of current order

Return type Decimal

Note: The internal rounding *must* be consistent, which is needed by :class: FabLabKasse.shopping.payment_methods. That means that x,xx5 € must always be rounded up or always down. “Fair rounding” like Decimal.ROUND_HALF_EVEN is not allowed.

For example:

- add article costing 1,015 € -> get_current_total == x
- add article costing 0,990 € -> get_current_total == x + 0,99

This would not be true with the fair strategy “round second digit to even value if the third one is exactly 5” (1,02€ and 2,00€).

`get_order_line`(*order_line_id*)
get order line of current order

`get_order_lines`()
return current order lines

Return type *OrderLine*

`get_products`(*current_category*)
return products in current category

Return type list(*Product*)

`get_root_category`()
return id of root category

`get_subcategories`(*current_category*)
return list(Category) of subclasses of the given category-id.

`list_clients`()
returns all selectable clients in a dict {id: Client(id), . . . }

`pay_order`(*method*)
store payment of current order to database :param method: payment method object, whose type is used to determine where the order should be stored in the database method.amount_paid - method.amount_returned is how much money was gained by this sale, must be equal to self.get_current_total()

`search_from_text`(*searchstr*)
search searchstr in products and categories :return: tuple (list of categories, products for table)

Return type list(*Product*)

`search_product_from_code`(*code*)
search via barcode, PLU or similar unique-ID entry. code may be any string

Returns product id

Raises ProductNotFound() if nothing found

`set_current_order`(*order_id*)
switch to another order (when the backend supports multiple orders)

`update_quantity`(*order_line_id*, *amount*)
change quantity of order-line.

if not all float values are allowed, round upvalue to the next possible one

FabLabKasse.shopping.backend.offline_base module

Module contents

`cart_from_app` - Loading carts via smartphone app

This module supports using your smartphone to enter (or scan) which products you would like to pay and transfer the cart (list of products and amounts) to the terminal.

The smartphone app https://github.com/FAU-Inf2/fablab-android has a java-based backend server https://github.com/FAU-Inf2/fablab-server which is used for communication by both the smartphone and the terminal.

You can use the HTML-based simulator as a replacement for FabLabKasse and also as a reference implementation:
`my-server.com/checkoutDummy/`

Workflow

- Fetch a random number from the appserver (e.g. 12345)

  `HTTP GET server/checkout/createCode`

- **show a QR code including this number to the user** (this is to guard against DOS or collisions with other people also sending a cart at the same time)
- User has his cart in the app, and pushes "send to cashdesk", scans code
- app sends cart to server, authenticating with the random number
- cashdesk polls for a cart:

  `HTTP GET server/checkout/12345`

  – -> If a cart was sent, the response is a JSON object containing the cart
- ask for payment, process payment
- When finished or aborted, send back the status to the server to notify the application:
  – success: `HTTP POST server/checkout/paid/12345`
  – aborted: `HTTP POST server/checkout/canceled/12345`

FabLabKasse.shopping.cart_from_app.cart_gui module

FabLabKasse.shopping.cart_from_app.cart_model module

Module contents

Submodules

`payment_methods`: select and handle a payment method

2.2 gui

Main window

2.3 scriptHelper: various utilities

2.4 Module contents

2.5 Files that are here for legacy reasons

Some parts of `FabLabKasse.shopping.backend.legacy_offline_kassenbuch` are in this folder for historical reasons

2.5.1 Kassenbuch

this file is here for legacy reasons

2.5.2 produkt

this file is here for legacy reasons

class `FabLabKasse.produkt.`**Produkt**(*plu*, *name*, *basiseinheit*, *basispreis*, *verkaufseinheiten=None*, *input_mode='DECIMAL'*)

Bases: `object`

add_verkaufseinheit(*verkaufseinheit*, *preis*, *basismenge=None*, *input_mode='DECIMAL'*)

Fügt eine neue Verkaufseinheit zum Produkt hinzu.

Parameters

- **verkaufseinheit** (`basestr`) – ist ein string, welcher die Einheit beschreibt, z.B. "Platte (600x300mm)"
- **preis** – ist der Preis für _eine_ solche Einheit
- **basismenge** – (optional) ein Umrechnungsfaktor: eine Basisheinheit mal Basismenge entspricht einer Verkaufseinheit
- **input_mode** – (optional) kann DECIMAL, INTEGER oder MINUTES sein. Ändert nichts an dem gespeicherten Wert, dieser ist immer Decimal.

gesamtpreis(*menge*, *einheit=None*)

Berechnet den Gesamtpreis für *menge einheit*. Wenn *einheit* nicht gegeben ist wird die Basiseinheit verwenden.

classmethod load_from_dir(*path*)

classmethod load_from_file(*filename*)

CHAPTER 3

Code structure overview

- `run` is the launcher, it starts `FabLabKasse.gui`
- the rest of the code is in folder FabLabKasse
- kassenbuch.py (currently still german) accounting CLI for legacy_offline_kassenbuch shopping backend
- produkt.py is directly in FabLabKasse-folder for legacy reasons
- cashPayment: automated cash payment - coin and banknote acceptors. see README_cashPayment.md
  - client: interface towards the GUI that connects with the device drivers
    - PaymentDevicesManager: manages all (multiple) payment devices, used by the GUI (as self.cashPayment)
    - PaymentDeviceClient: one device used by PaymentDevicesManager, launches a 'server' process and communicates with it via the protocol specified in cashPayment/protocol.txt
  - server: device drivers. they run as a standalone process
    - cashServer: abstract base class
    - exampleServer: simulated hardware for first tests
    - nv11, mdbCoinChanger: real hardware
  - protocol.txt: specification of how client (GUI) and server (individual device-driver process) communicate
  - cashState: database backend + CLI for accounting cash (individual pieces of money) inside the devices. This accounting is for auditing and error-finding purposes and therefore separate from the shopping backend, which has its own accounting (that does not look at individual coins, but just at sums). management tool can be started as ./cash from the main directory
  - listSerialPorts: tool to list all ports that can be found with pyserial, useful for configuration of all (usb-)serial connecting device drivers
- shopping:
  - backend: backends that provide connection to a webshop, ERP system, database etc and manages products, categories, carts and financial accounting (storage of payments)
    - abstract: abstract base class

    * offline_base: abstract base class for backends that read products only once at the start and keep the cart in memory; as opposed to a always-online system that has its whole state somewhere in the cloud
    * dummy: has some fake products, just silently accepts all payments without storing them somewhere
    * oerp: OpenERP / odoo implementation, still needs testing.
    * legacy_offline_kassenbuch: backend with product importing from a python script, SQLite based double-entry bookkeeping, contains many german database field names and is therefore marked as legacy. With some re-writing it would make a decent SQLite backend. Has a management CLI kassenbuch.py in the main folder.
  - payment_methods: different methods of payment like manual cash entry, automatic cash in+output, charge on client account, . . .
- libs: some helping libraries
- produkte: empty directory for local caching of product data (TODO rename)
- scripts: some helping cronjobs - TODO

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index