Programmieren mit PyUSB 1.0

Eine kleine Einführung

PyUSB ist eine Python-Bibliothek, die einfachen USB-Zugriff ermöglicht. PyUSB bietet einige Vorteile:

100% Python-Code:
Anders als die 0.X Version, die in C geschrieben war, wurde 1.0 in Python programmiert. Das erlaubt Pythonprogrammierern mit wenig Kenntnissen inC, die Abläufe in PyUSB besser zu verstehen.

Plattformneutralität:
In der Version 1.0 wurde ein frontend-backend-Schema eingefügt.Damit wird das API (application programming interface) von dem System getrennt. Der Kitt zwischen den beiden Schichten ist die |IBackend| Schnittstelle. PyUSB 1.0 beinhaltet backends für libusb 0.1, libusb 1.0 und OpenUSB. Natürlich kann man auch sein eigenes backend schreiben.

Portabilität:
PyUSB läuft auf jeder Plattform mit Python ab 2.4, ctypes und einem der unterstützten backends.

Einfachheit:
Kommunikation mit einem USB-Gerät war noch nie so einfach. USB ist ein komplexes Protokoll, aber PyUSB bringt gute Standardvorgaben für die meisten Konfigurationen mit.

Isochrone Übertragung:
PyUSB unterstützt isochrone Übertragungen, wenn das Geräte-backend mitspielt.

Obwohl PyUSB die Programmierung der USB-Schnittstelle stark vereinfacht, wird für dieses Tutorial eine minimale Kenntnis des USB-Protokolls vorausgesetzt. Wir empfehlen z.B. das Buch "USB Complete" von Jan Axelson. Eine erste Einführung bietet auch der Wikipedia-Artikel "Universal Serial Bus".

Genug geredet, legen wir los

Wer ist wer?

Zuerst ein Blick auf die PyUSB-Bibliothek. Wir finden in dem |usb| Paket die folgenden Module:

Name :	Beschreibung:
core	Das Hauptmodul
util	Hilfsfunktionen
control	Standard-Steuerungsaufrufe
legacy	0.X kompatible Schicht
backend	Eingebaute backends

Um z.B. das |core| Module zu laden, tippen wir folgendes ein:

```
>>> import usb.core
>>> dev = usb.core.find()
```

Starten wir mit einem Programm

Hier ist das einfachste Script, das den String 'test' an den ersten gefundenen Endpunkt OUT unseres USB-Gerätes sendet:

```
import usb.core
import usb.util
# Finden wir unser Gerät
dev = usb.core.find(idVendor=0xfffe, idProduct=0x0001)
# Gefunden?
if dev is None:
    raise ValueError('Device not found')

# Setzen wir die aktive Konfiguration. Ohne Argumente, wird die erste
# Konfiguration des Gerätes aktiviert.
dev.set_configuration()

# Holen wir uns eine Endpunkt-Instanz
cfg = dev.get_active_configuration()
intf = cfg[(0,0)]

ep = usb.util.find_descriptor(
    intf,
    # Nehmen wir den ersten OUT Endpunkt
    custom_match =  \
    lambda e: \
        usb.util.endpoint_direction(e.bEndpointAddress) == \
        usb.util.ENDPOINT_OUT)

assert ep is not None
```

```
# Und schreiben wir die Daten
ep.write('test')
```

In den beiden ersten Zeilen werden das Hauptmodul |core| und die Hilfsfunktionen in |util| geladen. Das nächste Kommando sucht nach unserem Gerät und gibt eine Instanz zurück, falls es gefunden wurde. Ansonsten wird |None| generiert. Danach aktivieren wir die Konfiguration. Wir haben kein Argument angegeben. Wir werden sehen, dass viele PyUSB-Funktionen Standardvorgaben für die meisten Standadgeräte mitbringen. In diesem Fall wird die erste gefundene Konfiguration geladen.
Danach suchen wir im ersten Interface, das wir kennen nach dem Endpunkt, der uns interessiert. Haben wir ihn gefunden, senden wir die Daten dorthin.

Ist uns die Endpoint-Adresse bereits bekanntt, können wir die Write-Funktion gleich über das Geräteobjekt aufrufen:

```
dev.write(1, 'test' )
```

Hier senden wir 'test' direkt an die Endpoint-Adresse /1/. Diese Funktionen werden in den folgenden Abschnitten alle noch detailliert behandelt.

Und wenn etwas falsch läuft?

Jede PyUSB-Funktion meldet im Fehlerfall eine Ausnahme. Neben den Standardausnahmen von Python, definiert PyUSB |usb.core.USBError| bei USB speziellen Fehlern.
Man kann auch die PyUSB-Log-Funktion anwenden. Sie nutzt das Logging-Modul. Um es zu aktivieren, belegen wir die Environment-Variable |PYUSB_DEBUG| mit einem der folgenden Werte: |critical|, |error|, |warning |, |info| oder |debug|.
Normal werden die Meldungen an sys.stderr gesendet. Man kann sie auch in eine Datei umleiten, indem man in der Environment-Variablen |PYUSB_LOG_FILENAME| diese Datei definiert.

Wie weit sind wir?

Die Funktion |find()| im Modul |core| sucht und zählt die Geräte auf, die an das System angeschlossen sind. Nehmen wir z.B. an, dass bei unserem Gerät die Vendor ID gleich 0xffe und die Produkt ID gleich 0x001 sind, so können wir es auf folgendem Weg finden:

```
import usb.core
dev = usb.core.find(idVendor=0xfffe, idProduct=0x0001)
if dev is None:
   raise ValueError('Unser Gerät ist nicht angeschlossen')
```

Das war's, die Funktion gibt uns ein |usb.core.Device|-Objekt zurück, das unser Gerät beschreibt. Falls es nicht vorhanden ist, kommt |None| zurück. Ab jetzt können wir jedes Feld des Device-Descriptors benutzen. Wenn wir z.B. herausfinden wollen, ob ein Drucker angeschlossen ist, geht das ganz einfach:

```
# Das ist noch nicht die ganze Geschichte, bleib am Ball
if usb.core.find(bDeviceClass=7) is None:
    raise ValueError('Kein Drucker gefunden')
# Ansonsten gibt es einen Drucker
```

Die 7 ist der Code für die Druckerklasse laut USB-Spezfikation. Und was ist, wenn wir alle Drucker des Systems finden wollen? Kein Problem:

```
# Ich sagte doch, das ist noch nicht die ganze Story
printers = usb.core.find(find-all=True, bDeviceClass=7)

# Python 2, Python 3, was auch immer
import sys
sysstdout.write('Es gibt ' + len(printers) + ' im System\n.')
```

Was ist passiert? Also, es ist Zeit für eine kleine Exkursion.
|find| hat einen Parameter |find-all|, der standardmäßig auf False gesetzt ist. In diesem Fall gibt |find| das erste Gerät zurück, das den Kriterien entspricht. Setzt man |find-all| allerdings auf True, erhält man eine Liste mit allen passenden Geräten. Einfach, oder?
Alles klar? Noch nicht! Jetzt die ganze Geschichte: Viele Geräte legen ihre Class-Information im Interface-Descriptor und nicht im Device-Descriptor ab. Also, um wirklich alle Drucker zu finden, müssen wir sowohl alle Konfigurationen und dann auch alle Interfaces durchsuchen, ob das Feld |bDeviceClass| bzw. |bInterfaceClass| den Wert 7 hat.
Einen guten Programmierer sollte es wundern, wenn es da nicht einen einfacheren Weg gäbe. Gibt es, aber zuerst wollen wir einen Blick auf den Code werfen, mit dem wir alle an USB angeschlossenen Drucker finden können:

```
import usb.core
import usb.util
import sys

class find_class(objekt):
    def __init__(self, class_):
        self._class = class_
    def __call__(self, device):
        # Checken wir zuerst das Gerät
        if device.bDeviceClass == self._class:
```

```
            return True
        # Ok, transferieren wir alle Geräte, um ein Interface
        # zu finden, das unserer Klasse entspricht
        for cfg in device:
            # Finden wir den Deskriptor
            intf = usb.util.find_descriptor(
                                        cfg,
                                        bInterfaceClass=self._class
                                )
            if intf is not None:

        return False

printers = usb.core.find(find_all=1, custom_match=find_class(7)
```

Der |custom_match| Parameter akzeptiert jedes aufrufbare Objekt, welches das device object erreicht. Er muss wahr bei einem passenden Gerät zurückgeben und falsch bei einem unpassenden. Man kann |custom_match| auch mit anderen Feldern kombinieren:

```
# Finde alle Drucker eines bestimmten Anbieters:
printers = usb.core.find(find_all=1, custom_match=find_class(7),
idVendor=0xfffe)
```

Hier suchen wir nur nach Druckern des Anbieters mit der Vendor-ID 0xfffe.

Ok, wir haben unser Gerät gefunden, aber bevor wir mit ihm kommunizieren, wollen wir mehr über es herausfinden, also Konfigurationen, Schnittstellen, Endpunkte, Transfertypen....

Wenn wir ein Gerät haben, können wir auf jedes device descriptor -Feld als Objekt zugreifen:

```
>>> dev.bLength
>>> dev.bNumConfigurations
>>> dev.bDeviceClass
```

Um die vorhandenen Konfigurationen eines Gerätes zu finden, können wir über das Gerät iterieren:

```
for cfg in dev:
    sys.stdout.write(str(cfg.bConfigurationValue) + '\n')
```

Genau so können wir über Konfigurationen iterieren, um die Schnittstellen zu finden und über die Schnittstellen, um Endpunkte zu finden. Jede Art von

Objekten besitzt als Attribute die Felder des entsprechenden Deskriptors. Ein Beispiel:

```
for cfg in dev:
   sys.stdout.write(str(cfg.bConfigurationValue) + '\n')
   for intf in cfg:
      sys.stdout.write('\t' + \
                       str(intf.bInterfaceNumber) + \
                       ',' + \
                       str(intf.bAlternateSetting) + \
                       '\n')
      for ep in intf:
         sys.stdout.write('\t\t' + \
                          str(ep.bEndpointAddress) + \
                          '\n')
```

Wir können auch direkt mit dem Index des Eintrags in den jeweiligen Deskriptoren zugreifen, wie hier:

```
>>> # Suchen wir die zweite Konfiguration
>>> cfg = dev(1)
>>> # Oder die erste Schnittstelle
>>> intf = cfg[((0,0)]
>>> # Den dritten Endpunkt
>>> ep = intf(2)
```

Wir sehen, diese Indizes sind nullbasiert. Aber Achtung! Da ist etwas seltsam in den Beispielen oben. Genau, der Index für die Konfiguration akzeptiert zwei Werte. Der erste ist der Index für das Interface und der zweite für das alternative Setting. So müssen wir um das erste Interface, aber das zweite alternative Setting zu erhalten. folgendes schreiben: intf = cfg[(0,1)].

Nun ist es an der Zeit, einen mächtigen Weg zu lernen, um Deskriptoren zu finden, die Hilfsfunktion |find_descriptor|. Wir haben sie schon beim Suchen des Druckers gesehen. |find_descriptor| arbeitet genau so wie |find|, mit zwei Ausnahmen:

1) |find_descriptor| erhält als ersten Parameter den übergeordneten Deskriptor, nach dem gesucht wird.
2) Es gibt keinen |backend| Parameter.

Nehmen wir an. wir haben einen Konfigurations-Deskriptor |cfg| und suchen alle Alternativ-Settings der Schnittstelle 1, so machen wir folgendes:

```
import usb.util
```

alt = usb.util.find_descriptor(cfg, find_all=True, bInterfaceNumber=1)

Merke, |find_descriptor| befindet sich im |usb.util| Modul. Es akzeptiert auch den weiter oben beschriebenen |custom_match| Parameter.

Behandeln mehrerer identischer Geräte

Es kann vorkommen, dass wir zwei identische Geräte an unseren Computer angeschlossen haben. Wie können wir sie unterscheiden? |device| Objekte melden sich mit zwei zusätzlichen Attributen, die allerdings nicht in den USB-Spezifikationen stehen, aber sehr nützlich sind: |bus| und |address| Attribute. Zuerst ist es wichtig zu erwähnen, das diese Attribute vom backend stammen und das backend muss sie nicht unterstützen. In diesem Fall sind sie auf |None| gesetzt. Sie stehen für die Bus-Nummer und die Bus-Adresse des Gerätes und wir können sie nutzen, um zwei Geräte mit gleichen |idVendor| und |idProduct| Attributen zu unterscheiden.

Was sollen wir machen?

USB-Geräte müssen nach dem Anschluss durch ein paar Standard-Aufrufe konfiguriert werden. Die USB-Spezifikationen sind sehr komplex mit den Deskriptoren, Konfigurationen, Schnittstellen, Alternativ-Settings, Transfer-Typen und all dem Kram. Dummerweise kann man sie nicht einfach ignorieren, ein Gerät arbeitet nicht ohne das Setzen einer Konfiguration, selbst wenn es nur eine hat. PyUSB versucht, uns das Leben so leicht wie möglich zu machen. Nachdem wir z.B. das Device-Objekt gefunden haben, müssen wir einen |set_configuration| Aufruf starten, um mit dem Gerät kommunizieren zu können. Der Parameter für diesen Aufruf ist das |bConfigurationValue| der gewünschten Konfiguration. Die meisten Geräte besitzen nur eine Konfiguration und man kann sich das mühsame Suchen über das |bConfigurationValue| sparen. Darum gibt es in PyUSB den |set_configuration|-Aufruf ohne Argumente. In diesem Falle wird die erste gefundene Konfiguration eingestellt, und wenn das Gerät nur eine bereit stellt, müssen wir uns sowieso nicht um das |bConfigurationValue| kümmern. Angenommen, wir haben ein Gerät mit einem einzigen Konfigurations-Deskriptor, dessen bConfigurationValue auf 5 gesetzt ist, dann funktionieren die folgenden Aufrufe gleich:

```
>>> dev.set_configuration(5)
# oder in der Annahme, dass Konfiguration 5 die einzige ist
>>> dev.set_configuration ()
#Oder
>>> cfg = util.find_descriptor(dev, bConfigurationValue=5)
>>> cfg.set()
#Oder auch
>>> cfg= util.find_descriptor(dev, bConfigurationValue=5)
```

>>> dev.set_configuration(cfg)

Toll! Wir können ein Konfigurations-Objekt als Parameter für die |set_configuration| Funktion nutzen. Und genau so hat es eine Methode, sich selbst als die laufende Konfiguration einzustellen.

Ein Weiteres ist das Setzen der alternativen Schnittstelle. Für jedes Gerät kann nur eine Konfiguration aktiv sein, und jede Konfiguration kann mehr als eine Schnittstelle haben, die alle zur gleichen Zeit aktiv sein können. Man kann sich das besser erklären, wenn man die Schnittstellen als logische Geräte betrachtet. Stellen wir uns einen Multifunktions-Drucker vor, der sowohl Drucker als auch Scanner ist. Für das Gerät ist nur eine Konfiguration aktiv, aber intern besitzt es zwei Schnittstellen, eine für das Drucken und eine für das Scannen. Geräte mit mehr als einer Schnittstelle heißen Komposit-Geräte. Wenn wir diesen Multifunktions-Drucker an den Computer stecken, lädt das Betriebssystem zwei verschiedene Treiber, einen für jedes "logische" Gerät. (Genau genommen verläuft die Sache ein wenig komplizierter, aber für uns Programmierer reicht diese simple Betrachtungsweise)

Was ist nun mit diesen Alternate Settings? Gute Frage. Eine Schnittselle hat eine oder mehrere alternative Settings. Auch wenn es seltsam klingt, aber eine Schnittstelle mit nur einem alternativen Setting wird behandelt, als ob es kein alternatives Setting gäbe. Alternative Settings sind für Schnittstellen das, was Konfigurationen für Geräte sind, für jede Schnittstelle kann nur ein alternatives Setting aktiv sein. Wenn z.B. die USB Specs sagen, dass ein Gerät keinen isochronen Endpunkt in seinem ersten alternativen Setting haben kann, so muss ein Streaming-Gerät also mindestens zwei alternative Settings haben, mit dem isochronen Endpunkt im zweiten. Aber im Gegensatz zu Konfigurationen müssen Schnittstellen mit nur einem alternativen Setting nicht gesetzt werden. Man wählt das alternative Setting für eine eine Schnittstelle durch die Funktion |set_interface_altsetting| aus:

>>> dev.set_interface_altsetting(interface = 0, alternate_setting = 0)

Warnung

Die USB Specs sagen, dass ein Gerät einen Fehler melden darf, wenn es einen SET_INTERFACE-Aufruf erhält für eine Schnittstelle, die keine weiteren alternativen Einstellungen hat. Falls wir also nicht sicher sind, ob die Schnittstelle entweder mehrere alternative Einstellungen hat, oder einen SET_INTERFACE-Aufruf akzeptiert, ist der sicherste Weg, den |set_interface_altsetting| Aufruf in einem try-except Block zu tätigen, vielleicht so:

try:

```
    dev.set_interface_altsetting(....)
except USBError:
    pass
```

Wir können auch ein Interface-Objekt als Parameter für die Funktion nutzen, die |interface| und die |alternate_setting| Parameter werden automatisch aus den Feldern |bInterfaceNumber| und |bAlternate_setting| genommen. Ein Beispiel:

```
>>> intf = find_descriptor(...)
>>> dev.set_interface_altsetting(intf)
>>> intf.set_altsetting()
```

Warnung

Das Interface-Objekt muss zum aktiven Konfigurations-Deskriptor gehören.

Unterhalten wir uns

Nun ist es an der Zeit zu lernen, wie die Kommunikation mit USB-Geräten funktioniert. USB hat vier Möglichkeiten des Transfers: bulk, interrupt isochronous und control. (Was soviel heißt wie Massentransfer, Interruptgesteuert, ungleichmäßiger Serientransfer und Steuerungstransfer). Mehr dazu in den USB-Specs.

Dabei ist der control-Transfer der einzige mit strukturierten Daten, beschrieben in den USB-Specs. Die anderen senden und empfangen aus Sicht des USB rohe Daten. Deswegen müssen wir mit dem control-Transfers gesondert umgehen, alle anderen Transfers werden in der gleichen Weise behandelt.

Wir starten einen control-Transfer mit der |ctrl_transfer| Methode. Sie wird sowohl für OUT- als auch für IN-Transfers benutzt. Die Richtung wird durch den |bmRequestType| Parameter festgelegt.

Die |ctrl_transfer| Parameter gleichen fast der control request-Struktur. Das folgende Beispiel zeigt, wie man einen control Transfer startet:

```
>>> msg = 'test'
>>> assert dev.ctrl_tranfer(0x40, CTRL_LOOPBACK_WRITE, 0, 0, msg ==
>>> len(msg)
>>> ret = dev.ctrl_transfer(0xc0, CTRL_LOOPBACK_READ, 0, 0, len(msg))
>>> sret = ''.join([chr(x) for x in ret])
>>> assert sret == msg
```

In diesem Beispiel gehen wir davon aus, dass unser Gerät zwei custom control Aufrufe bearbeitet, die wie eine Rückkopplungsschleife wirken. Was wir mit

der |CTRL_LOOPBACK_WRITE| Anweisung schreiben, können wir mit der |CTRL_LOOPBACK_READ| Anweisung lesen.

Die ersten vier Parameter der Standard control transfer Struktur sind die |bmRequestType|, |bmRequest|, |wValue| und |wIndex| Felder. Der fünfte Parameter sind entweder die Nutzdaten für einen OUT Transfer oder es ist die Anzahl der Bytes, die bei einem IN Transfer gelesen werden. Die Nutzdaten können jede Art von Sequenz sein, die wir als Parameter für die |__init__| Methode nutzen können. Falls es keine Nutzdaten gibt, muss der Parameter auf |NONE| gesetzt sein (oder 0 im Falle eines IN Transfers). Es gibt noch einen letzten, optionalen Parameter, der das Timeout der Operation bestimmt. Wenn wir ihn nicht belegen, kommt ein vorgegebener Timeout zum Tragen. Beim OUT Transfer gibt der Rückgabewert die Anzahl der Bytes an, die wirklich zum Gerät gesendet wurden. Beim IN Transfer ist der Rückgabewert ein Array Objekt mit den gelesenen Daten.

Für die anderen Transfers nutzen wir jeweils die |write| und die |read| Methode, um Daten zu schreiben bzw. zu lesen.Wir müssen uns nicht um den Transfer-Typ kümmer, er wird automatisch von der Endpunkt Adresse vorgegeben. Hier ist unser Rückkopplungsbeispiel in der Annahme, dass wir eine Rückkopplungsschleife am Endpunkt 1 haben:

```
>>> msg = 'test'
>>> assert len(dev.write(1, msg, 100)) == len(msg)
>>> ret = dev.read(0x81, len(msg), 100)
>>>sret = ''.join([chr(x) for x in ret])
>>> assert sret == msg
```

Der erste und der dritte Parameter sind für beide Methoden gleich. es sind die Endpunktadresse bzw. der Timeoutwert. Der zweite Parameter sind die Nutzdaten beim Schreiben oder es ist die Anzahl Bytes, die zu lesen sind. Die zurückgegebenen Daten sind entweder eine Instanz des array objects bei der |read| Methode oder die Anzahl der geschriebenen Bytes bei der |write| Methode.

Seit der beta 2 Version ist es möglich, anstelle der Anzahl Bytes an die |read| Funktion mittels |control_transfer| ein array object weiter zu geben, in das die Bytes gelesen werden.In diesem Falle bestimmt sich die Byteanzahl durch die Länge des arrays mal dem Wert |array.itemsize|.

Genau wie beim |control_transfer| ist der |timeout| Parameter optional. Machen wir keine Angabe, wird der |Device.default_timeout| Wert genommen.

Kontrolle ist alles

Neben den Transfer-Funktionen bietet das Modul |usb.control| weitere Funktionen, die Standard USB CONTROL Anforderungen erledigen und das Modul |usb.util| bietet die bequeme Funktion |get_string|, um speziell string descriptors zurück zu geben.

Wetere Punkte

Anfänglich gab es nur die libusb. Dann kam libusb 0.1 und 1.0. Dann schuf man OpenUsb, und jetzt haben wir eine Fülle von USB Bibliotheken. Wie geht PyUSB damit um? Nun, PyUSB ist sehr demokratisch, und so können wir jegliche libusb wählen. Schließlich können wir auch unsere eigene libusb von Grund auf schreiben und PyUSB anweisen, sie zu benutzen.

Die |find| Funktion kennt einen weiteren Parameter, der bisher noch nicht erwähnt wurde. Es ist der |backend| Parameter. Wenn man ihn nicht angibt, wird eine der vorgegebenen backends benutzt. Ein backend ist ein Objekt, das von |usb.backend.IBackend| übernommen wird, verantwortlich für die Bedienung des Betriebssystemabhängigen USB-Krams/Drumherum. Wie zu erwarten, sind diese Vorgaben die libusb 0.1, 1.0 und OpenUSB backends.

Wir können unser eigenes Backend schreiben und es benutzen. Am besten orientieren wir uns bei |IBackend| und implementieren die nötigen Methoden. Man kann sich bei der Dokumentation des |usb.backend| schlau machen.

Nicht eigennützig sein

Python hat was wir ein "automatische Memory-Management" nennen. Damit ist gemeint, dass die virtuelle Maschine entscheidet, wann Objekte im Speicher gelöscht werden. PyUSB managt all die "low level" Resourcen, die es zur Arbeit benötigt(Schnittstellen-Bestimmung, Gerätebedienung usw.) im Hintergrund und die meisten Benutzer müssen sich darüber auch keine Gedanken machen, aber wegen der unvorhersehbaren Art des automatischen Löschens von Objekten in Python, können Benutzer nicht vorhersagen, wann die Resourcen freigegeben werden. Einige Programme müssen ihre Resourcen gezielt zuweisen und freigeben. Für diese Art von Applikationen besitzt das Modul |usb.util| einen ganzen Satz von Funktionen, um die Resourcen zu managen.

Um Schnittstellen manuell zuzuweisen bzw. freizugeben, können wir die Funktionen |claim_interface| und |release_interface| nutzen. |claim_interface| weist die spezielle Schnittstelle zu, falls das Gerät es nicht schon getan hat. Wenn das Gerät bereits zugewiesen ist, macht das gar nichts. In gleicher Weise gibt |release_interface| die Schnittstelle frei, falls sie zugewiesen ist, wenn nicht, macht es ebenfalls nichts. Näheres lässt sich im Manual "interface claim"

nachlesen, um das Problem der Konfigurations-Auswahl zu lösen,das in der libusb-Dokumentation beschrieben ist.

Um alle Resourcen, die vom Geräte Objekt (einschließlich der Schnittstelle) belegt sind frei zu geben, können wir die |dispose_resources| Funktion nehmen. Sie gibt alle Resourcen frei und bringt das Geräte-Objekt(allerdings nicht die Geräte-Hardware) in den Zustand wie zum Zeitpunkt nach der |find| Funktion.

Bibliotheken von Hand zuweisen

Allgemein ist ein backend eine Programmschicht um eine Bibliothek, die das USB-Zugriffs API bedient. Normalerweise benutzt das backend die find_library() ctypes Funktion. Unter Linux und anderen Unix-ähnlichen Sytemen, versucht |find_library| externe Programme(wie //sbin/ldconfig/, /gcc/ und /objdump/) zu nutzen, um die Bibliothek zu finden.

In Systemen, bei denen diese Programme fehlen, oder das Bibliotheks Cache abgeschaltet ist, kann diese Funktion nicht benutzt werden. Um diese Einschränkung zu überwinden, erlaubt PyUSB uns, eine eigene find_library() (Anfrage)Funktion an das backend zu richten.

Ein Beispiel für eine solches Szenario wäre:

```
>>> import usb.core
>>> import usb.backend.libusb1
>>> backend = usb.backend.libusb1.get_backend(find_library=lambda x:
      "/usr/lib/libusb-1.0.so")
>>> dev     = usb.core.find(..., backend=backend)
```

Wir müssen bedenken, bei dem find_library-Argument für die get_backend() Funktion handelt es sich um eine Funktion, die verantwortlich dafür ist, die korrekte Bibliothek für das backend zu finden.

Regeln der alten Schule

Wenn wir eine Applikation mit dem alten PyUSB API (0.irgendwas) geschrieben haben, ergibt sich natürlich die Frage, ob wir unseren Code updaten müssen, um das neue API zu nutzen. Nun, wir sollten es tun, aber es ist (noch) nicht notwendig. PyUSB 1.0 bringt das |usb.legacy| Kompatibilitätsmodul mit. Es legt das alte API über das neue API. Genau so müssen wir auch nicht die |import.usb| Anweisung durch eine |import usb.legacy as usb| Anweisung ersetzen. Die unveränderte Applikation muss laufen, denn die |import usb| Anweisung importiert alle öffentlichen Symbole von |usb.legacy|. Tauchen Probleme auf, deutet das auf einen Fehler im Programm hin.